

**TEMPLATE  
PROJECT WORK**

<b>Corso di Studio</b>	INFORMATICA PER LE AZIENDE DIGITALI (L-31)
<b>Dimensione dell'elaborato</b>	Minimo 6.000 – Massimo 10.000 parole ( <i>pari a circa Minimo 12 – Massimo 20 pagine</i> )
<b>Formato del file da caricare in piattaforma</b>	PDF
<b>Nome e Cognome</b>	Vito di Venosa
<b>Numero di matricola</b>	0312200363
<b>Tema n.</b> (Indicare il numero del tema scelto):	1
<b>Titolo del tema</b> (Indicare il titolo del tema scelto):	La digitalizzazione dell'impresa
<b>Traccia del PW n.</b> (Indicare il numero della traccia scelta):	6
<b>Titolo della traccia</b> (Indicare il titolo della traccia scelta):	Sviluppo di una applicazione full-stack API-based per un'impresa del settore finanziario
<b>Titolo dell'elaborato</b> (Attribuire un titolo al proprio elaborato progettuale):	FinHub Credit Bank

**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):

Diverse sono state le abilità e le competenze tecniche che mi hanno supportato nel portare a termine questo progetto. Per descriverle con chiarezza, ritengo opportuno partire in ordine seguendo le fasi tipiche di sviluppo e progettazione. La prima competenza rilevante è stata lo studio e l'analisi della funzione, che mi ha permesso di progettare in modo ordinato le varie fasi di sviluppo, massimizzando la qualità del risultato e assicurandomi di coprire tutti i punti richiesti dalla traccia; inoltre mi ha permesso anche di riuscire a fare mente locale delle idee e di trovare la giusta strada da intraprendere per lo sviluppo della piattaforma. Tale competenza è stata inoltre utile per definire una timeline di riferimento, precisa e strutturata, che ha guidato l'intero processo e che mi ha permesso di creare task checkabili mediante una app chiamata 'todoist' portando a massimizzare il tempo impiegato. In seguito, si sono rivelate fondamentali le conoscenze acquisite nel corso di studi in merito alle basi di dati e ai database, indispensabili per progettare la struttura del database, le tabelle e le relazioni tra entità, nonché per scegliere in maniera appropriata le tipologie di dati da archiviare.

Altrettanto importante è stata la padronanza degli algoritmi e delle strutture dati, che ha reso possibile l'ottimizzazione del codice, in particolare nella gestione e manipolazione dei dati estratti dal database durante le richieste di fetch.

Dal punto di vista pratico, le competenze nello sviluppo web hanno avuto un ruolo essenziale:

- l'utilizzo di HTML e CSS è stato determinante per progettare un'interfaccia utente intuitiva e professionale;
- JavaScript è stato impiegato per interagire con le API, estrarre i dati e renderizzarli dinamicamente nelle pagine.

Un ulteriore contributo significativo è derivato dalle competenze relative alle API (Application Programming Interface), che hanno consentito un'interazione sicura con il database e favorito la suddivisione logica delle sezioni, garantendo una struttura scalabile e moderna. Le conoscenze del linguaggio Python hanno rappresentato un'ottima base di partenza per l'apprendimento e l'utilizzo del framework Django, impiegato nella realizzazione del progetto, assicurando una gestione

completa ed efficace sia del lato back-end sia del front-end. Infine, è doveroso citare le competenze sviluppate nell'ambito aziendale e gestionale, che mi hanno permesso di comprendere con maggiore profondità la richiesta della traccia e di ideare un'impresa fittizia denominata *Finhub Credit Bank*, utile a contestualizzare il progetto e ad avvicinarmi a dinamiche tipiche del settore finanziario.

#### **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):

##### *Fase 1 Studio e analisi della funzione:*

Durante questa fase ho analizzato la traccia e i requisiti richiesti e ho realizzato il file dello studio e analisi della funzione che mi è servito come base di riferimento al fine di creare una timeline precisa andando a sottolineare i requisiti di ogni fase di sviluppo. Il tempo di creazione dello studio della funzione è durato circa 3 giorni dedicando 2h ore giornaliere al progetto, durante la quale ho analizzato la traccia e approfondito ogni suo aspetto cercando di capire al meglio i requisiti da sviluppare. Durante questa fase non ho avuto particolari problemi.

##### *Fase 2 Creazione timeline progetto:*

In questa fase ho creato una timeline basandomi sullo studio della funzione creata nella fase 1 e questo mi ha permesso di avere ben chiaro cosa avrei dovuto sviluppare e il tempo da rispettare. La timeline mi ha portato via 1 giorno sempre da 2h ore e avendo già lo studio della funzione con me è stato molto semplice capire le tempistiche.

##### *Fase 3 Creazione e inizializzazione repository:*

In questa fase ho creato i set up iniziali della cartella necessari al fine dello sviluppo:

- Creazione cartella root mediante il comando da terminale 'django-admin startproject <nome\_progetto>'.
- Installazione docker, dfr\_spectacular, rest\_framework e configurazione.
- Creazione file docker-compose.yml, dockerfile per la configurazione necessaria al fine di utilizzare l'immagine di docker con le varie dipendenze installate all'interno e configurato il database all'interno con le credenziali e le coordinate per utilizzare PostgreSQL.
- .github/workflows/ci.yml per le git actions.
- Creato file .env per tenere al sicuro le credenziali di accesso al database
- Creato il file requirements.txt necessario per far sì che docker riconosca tutte le dipendenze da scaricare durante la creazione del container del progetto.
- Modificati i settings della cartella root per utilizzare rest\_framework e dfr\_spectacular.
- Creata la repository github con i vari branch.

Questa fase è stata completata in circa 4 giorni dal 12/06/ al 16/06 e durante la quale non ho avuto particolari problemi poiché avevo già avuto modo di fare abbastanza pratica nella configurazione di un progetto django.

##### *Fase 4 Creazione login, register e profile:*

Questa è una delle prime fasi corpose in quanto partendo dalla configurazione iniziale sono andato a creare tutto il necessario per gestire il login e register in sicurezza e i dati del profilo e questa è la scaletta delle implementazioni:

- creazione app user nella cartella progetto.
- creazione models per la tabella dell'users.
- creazione dell'utente superuser per accedere alla sezione admin.
- creazione serializers.py e views.py per l'users.
- creazione frontend per visualizzazione dei template necessari per il login e register.
- creazione logica che gestisce le richieste per il register e per il login.
- configurazione dell'email nella cartella root e di smtp lib per l'invio di email automatiche nel progetto.
- creazione di email personalizzate con i template per diversificare e rendere professionali i vari tipi di email tra cui email di attivazione account e reset password.
- creazione logica che permette di utilizzare il token jwt per l'autenticazione della sessione.
- creazione logica che consente di sfruttare la sessione con il token jwt per l'autenticazione tramite email durante la fase di registrazione.
- creazione logica di rest password mediante l'invio dell'email e il link sempre con sessione protetta da token jwt.
- creazione logica che consente di salvare l'utente quando dopo la registrazione la richiesta risponde con uno status

code 200 e creata la gestione dei relativi errori con messaggi.

- creata la logica che gestisce l'attivazione dell'account dopo che l'utente ha cliccato sul link inviato tramite email inserita durante la registrazione protetto con il token jwt.
- creata la logica di verifica utente durante il login e di verifica se l'account è attivo e al corrispettivo redirect. - creazione tabella per l'inserimento dei dati personali e dell'account collegato all'user loggato.
- creazione serializers.py per la tabella dei dati personali.

Questa fase è durata dal 16/06/2025 al 03/07 e durante questa fase le uniche difficoltà riscontrate sono state l'utilizzo delle email personalizzate in base alla verifica dell'account o al reset della password e l'utilizzo della sessione autenticata con il token jwt per autorizzare e attivare l'account una volta registrato.

#### *Fase 6 Creazione sezione profilo e homepage:*

Durante questa fase ho sviluppato la sezione dei dati del profilo implementando l'inserimento dei dati personali e la modifica dei dati e la fase si divide in queste sottofasi:

- creazione frontend per la visualizzazione della sezione profilo.
- creazione del frontend per la visualizzazione della sezione dati account e dati personali.
- creazione della logica che gestisce le richieste mediante api per il fetch dei dati personali dal database.
- creazione della logica che gestisce le richieste api per il salvataggio dei dati personali.
- creazione della logica che gestisce le richieste api per la modifica dei dati personali e relativo salvataggio nel database.
- creazione logica che gestisce la modifica dell'email di accesso all'account mediante validazione tramite email.
- aggiunta la logica che gestisce la creazione del codice univoco da mostrare all'assistenza qual'ora l'email inserita non corrispondesse ad un email valida (poichè una volta avviata la richiesta di modifica email e confermata la modifica si verrà disconnessi dall'account)
- creazione navbar.
- creazione template homepage.
- creazione della logica per il fetch dei dati all'interno della homepage.

Questa fase è stata completata in 7 giorni dal 03/07 al 10/07 senza particolari difficoltà in quanto gran parte del lavoro era prettamente frontend.

#### *Fase 7 Implementazione sezione bonifico:*

Durante questa fase ho sviluppato interamente la sezione del bonifico sia lato backend che frontend seguendo questi step:

- creazione template della sezione bonifico
- creazione delle tabelle necessarie nel database al fine di gestire i dati dei soggetti coinvolti.
- creati i serializers.py per le tabelle che ospiteranno i dati dei soggetti del bonifico.
- create le api necessarie per gestire le richieste di bonifico.
- creata la logica che durante la registrazione crea in automatico un IBAN fittizio e lo collega all'account registrato con relativo cvv. L'IBAN viene salvato senza spazi e tutti gli input vengono normalizzati per garantire la corrispondenza durante i bonifici.
- creazione della logica che verifica le coordinate dei dati prima di fare il bonifico e l'operazione di bonifico verrà confermata soltanto se l'utente inserisce il pin corretto.
- creazione della logica che gestisce la possibilità di salvare le coordinate del beneficiario in rubrica così da poterle riutilizzare con un click.
- creazione logica che permette la stampa del movimento di bonifico direttamente dalla modale del successo.
- creazione logica che gestisce gli errori e i messaggi di successo o non del bonifico.

Questa fase è stata completata in 6 giorni a partire dal 10/07 fino al 16/07, in questa fase l'unico intoppo che ha portato via un po di tempo è stato il problema di sincronizzare l'IBAN creato durante la registrazione nel bank account e nell'account dell'utente e dunque la possibilità di generare l'iban e cvv che per scopo di test ha un algoritmo semplice ma comunque univoco, mentre il saldo parte da zero e bisogna versare manualmente il saldo e per uso di test viene fatto dalla sezione admin; infine il pin viene generato in automatico durante la fase di registrazione e inviato tramite email una sola volta.

#### *Fase 8 Sezione Elenco movimenti, Estratto conto, Elenco transazioni:*

In questa fase sono state implementate tre sezioni che rappresentano rispettivamente l'elenco degli estratti conto, l'elenco dei movimenti e l'elenco delle transazioni seguendo queste sottofasi:

- Creazione della sezione elenco movimenti che comprende sia gli accrediti che gli addebiti.
- Creazione delle tabelle del database e dei serializers.py per i dati dei movimenti associati all'account.
- Creazione delle views e degli endpoint per le api che gestiscono le richieste di fetch e rendering dei dati nella tabella dell'elenco dei movimenti.
- Creazione della logica per la ricerca, i filtri e la paginazione della tabella.
- Creazione della logica che permetta la stampa in pdf dei singoli movimenti.
- Creazione del template della sezione elenco transazioni.
- Riutilizzata la logica del fetch e del rendering della tabella con l'elenco di tutte le transazioni fatte con il conto.
- Riutilizzate la logica di filtri, ricerca e paginazione per la sezione elenco transazioni.
- Creazione della logica che permette la stampa in pdf delle singole transazioni.
- Creazione della sezione elenco estratto conto.
- Creata la logica che permette di generare in automatico l'estratto conto ogni primo del mese con l'elenco di tutti i movimenti.
- Creata la logica che permette di mostrare gli estratti conti nella tabella in ordine di data e mostrare la variazione del saldo (saldo iniziale a saldo finale).
- Creata la logica che permette di scaricare il pdf con l'elenco di tutti i movimenti presenti all'interno del mese preso in considerazione.
- Creazione sezione statistica che permette di vedere un resoconto delle operazioni fatte e due grafici che rappresentano rispettivamente l'andamento delle spese / entrate e categorie di spese.

Questa fase mi ha portato via due settimane, dal 16/07 al 29/07, e l'unica difficoltà significativa è stata il rendering del saldo iniziale e finale all'interno dei vari estratti conto. Per risolvere questo problema ho dovuto progettare e implementare un algoritmo basato sulla ricorsione, capace di calcolare in maniera accurata i valori mese per mese. Questa scelta è stata guidata dalla volontà di aderire a uno scenario realistico: in contesti bancari, infatti, i saldi iniziali e finali sono tipicamente considerati dati derivati, o addirittura "dati spazzatura", in quanto calcolabili in qualunque momento a partire dalle transazioni memorizzate. Di conseguenza, non è necessario archivarli direttamente nel database. L'adozione di questa strategia mi ha permesso di ridurre la ridondanza dei dati e di mantenere il modello più pulito ed efficiente, pur rispettando la coerenza contabile richiesta dal dominio.

#### *Fase 9 creazione sezione suddivisione spese e goals saving:*

In questa fase ho creato le due sezioni innovative che possono con future implementazioni differenziare il progetto dalle classiche piattaforme di gestione dei conti e seguendo questi passi:

- creazione template delle due sezioni.
- creazione della logica che gestisce il salvataggio dei piani di suddivisione spesa, l'allocazione del capitale e la possibilità di cambiare colori per rendere meglio la suddivisione delle categorie.
- creazione della logica che permette di fetchare e salvare i piani di suddivisione spese e mostrarli mediante un grafico ad aereogramma.
- creazione logica che gestisce la creazione dei goals savings e che permette di personalizzarli attraverso nome categoria, colore e versamento mensile.
- creazione della logica che gestisce il versamento automatico mensilmente inserito durante la creazione del goals savings.
- creazione della logica che gestisce il versamento manuale nel goal saving.

Questa fase è stata completata in 4 giorni che va dal 29/07 al 02/08 e non ho avuto particolari difficoltà di sviluppo.

#### *Fase 10 fix bugs e creazione unit test*

In questa fase ho corretto i bugs che durante lo sviluppo ho segnato in un file a parte, successivamente ho scritto e creato gli unit test necessari per testare il corretto funzionamento delle api implementate. Questa fase è durata circa 4 giorni, dal 02/08 al 06/08.

#### *Fase 11 fix bugs emersi dagli unit test*

In questa fase ho corretto tutti i bugs emersi dopo il run dei vari unit test creati e ho testato la creazione del container e il

download delle dipendenze scaricando e iniziando la repository del progetto su un altro computer. Successivamente ho creato il file read me per assicurarmi che il progetto venga avviato nella maniera corretta una volta scaricato. Questa fase è stata completata in 5 giorni circa dal 06/08 al 11/08

### *Fase 12 report e fase finale*

In questa fase conclusiva, che si è svolta dal 11/08 al 03/09, ho nuovamente ricontrollato che tutte le funzionalità implementate fossero operative e prive di anomalie. Per validare ulteriormente la documentazione, ho chiesto anche ad un amico di seguire il README da me redatto per scaricare, configurare e avviare il progetto su un computer diverso dal mio. Questo test ha avuto lo scopo di verificare se le istruzioni fossero sufficientemente chiare e se coprissero tutte le possibili situazioni che possono verificarsi durante l'inizializzazione dell'ambiente. Successivamente ho provveduto a rileggere e migliorare il README, ho aggiunto la cartella docs contenente screenshot della piattaforma e di Swagger in funzione, e ho incluso tutti i file richiesti dalla traccia (schema UML, schema ER e file OpenAPI YAML). Infine, ho redatto il presente report seguendo fedelmente i punti previsti dalla traccia e ho utilizzato il tempo restante per rileggerlo più volte, assicurandomi che ogni sezione fosse completa e coerente con gli obiettivi richiesti, così da garantire un elaborato finale accurato e allineato agli standard richiesti.

### **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):

#### *Software e linguaggi*

- Python 3 + Django 5.2: framework principale per il backend, con approccio OOP e admin integrato utile per i dati di prova.
- Django REST Framework 3.16: esposizione delle API REST (serializers, viewset/APIView, paginazione, filtri).
- Autenticazione JWT con djangorestframework-simplejwt; Djoser per endpoint auth standard.
- Documentazione API con drf-spectacular (+ sidecar per asset Swagger/ReDoc): genera lo schema OpenAPI dalla codebase.
- PostgreSQL (driver psycopg): database transazionale per garantire integrità e performance.
- docker-compose + Dockerfile: ambienti replicabili (servizi web, db, mailhog).
- MailHog: SMTP finto + web UI per testare attivazione via email e invii senza dipendere da provider reali.

#### *Strumenti di sviluppo e test*

- Swagger UI / ReDoc (serviti da drf-spectacular) per provare le API direttamente dal browser.
- Django admin per amministrare utenti/conti e inizializzare saldi in ambiente di prova.
- Git/GitHub per versionamento del codice.
- Diagrammi (Mermaid/draw.io) per ER/UML inseriti nel report.

#### *Modelli e principi adottati*

- Stile REST con risorse e verbi HTTP coerenti; codici d'errore standard (400/401/403/404).
- Transazioni ACID per operazioni critiche (bonifico/aggiornamento saldi) e movimenti append-only (storni come contro-movimenti) per auditabilità.
- JWT Bearer per sessioni stateless e uso multi-canale (web/mobile).

#### *Motivazioni delle scelte*

- Django/DRF: maturità, documentazione ampia, rapidità di sviluppo con solide best practice.
- PostgreSQL: affidabilità e vincoli di integrità forti; facile gestione in Docker.
- SimpleJWT: integrazione semplice con DRF e supporto a refresh e blacklist.
- drf-spectacular: evita discrepanze tra implementazione e documentazione, facilitando la consegna dello YAML OpenAPI.
- MailHog: test end-to-end dell'onboarding via email senza costi/limiti dei servizi reali.

Durante la fase di selezione delle tecnologie ho valutato anche alternative come Flask e FastAPI per il backend, o MySQL come database. Tuttavia, la scelta è ricaduta su Django e PostgreSQL perché garantiscono maggiore solidità nelle transazioni e strumenti integrati che riducono i tempi di sviluppo. Flask o FastAPI avrebbero offerto leggerezza e velocità, ma avrebbero richiesto più configurazione manuale, mentre PostgreSQL si è distinto per le funzioni avanzate



rispetto a MySQL (integrità referenziale, tipi avanzati, CTE). Inoltre, durante gli studi universitari ho seguito corsi su Django e sviluppato piccoli progetti pratici, che mi hanno permesso di apprezzarne la completezza e le performance. Ho trovato particolarmente utile la possibilità che offre di strutturare il source code in app indipendenti, isolando le diverse funzionalità durante lo sviluppo e collegandole in fase di esecuzione tramite gli endpoint definiti nel progetto root.

### *Come ho individuato e reperito le risorse*

- Per lo studio di Django, Django REST Framework e l'approfondimento di Python ho seguito alcuni corsi sulla piattaforma *Udemy*. Questi percorsi formativi mi hanno permesso di costruire una base solida e non solo di portare a termine il presente progetto, ma anche di acquisire le fondamenta per affrontare progetti futuri.
- Per *drf-spectacular* e Swagger ho consultato la documentazione ufficiale e seguito tutorial specifici, approfondendo in particolare l'uso di pattern comuni come l'autenticazione JWT, la paginazione e i filtri.
- Per l'utilizzo di PostgreSQL ho seguito un corso sempre su *Udemy* che trattava in modo approfondito i concetti già visti nello studio dei database. Il corso includeva inoltre un confronto tra i principali database (MySQL, MongoDB e PostgreSQL), permettendomi di comprendere meglio i punti di forza di ciascuno e di motivare la scelta più adatta al progetto.
- Infine, ho confrontato alternative come *drf-yasg* e *drf-spectacular*, optando per quest'ultimo in quanto meglio supportato e più aggiornato a livello di manutenzione.

### *Difficoltà incontrate e come le ho superate*

Una delle principali sfide è stata il calcolo del saldo iniziale e finale per ciascun estratto conto mensile. Per affrontarla ho progettato un algoritmo basato sulla ricorsione: partendo dal saldo finale dell'ultimo estratto conto, l'algoritmo somma progressivamente le uscite e sottrae le entrate di ogni mese, ricostruendo così il saldo iniziale prima che i movimenti vengano contabilizzati e così itera e ripete per tutti gli estratti conto generati. Questo approccio ha garantito coerenza con lo scenario reale, evitando di dover archiviare dati derivati ridondanti nel database.

Un'altra difficoltà ha riguardato la generazione dello schema OpenAPI: alcune *APIView* non avevano la *serializer\_class* e generavano errori in fase di build. Ho risolto aggiungendo la *serializer\_class* mancante e, per le view che filtravano su *request.user*, gestendo il caso *swagger\_fake\_view* o impostando *queryset = Model.objects.none()*.

Ho incontrato anche problemi con i campi calcolati, come le percentuali dei *goals*, che non venivano tipizzate correttamente in Swagger. La soluzione è stata dichiararli esplicitamente con *@extend\_schema\_field*, così da mostrare il tipo corretto. Allo stesso modo, i parametri di path (UUID/int) venivano interpretati come string generiche: li ho specificati manualmente nell'URL o annotati nel codice, migliorando l'accuratezza dello schema.

In fase di containerizzazione ho ricevuto un warning in docker-compose relativo a version: "3.9", deprecata in Compose v2. Ho mantenuto comunque il file funzionante, preparando una versione alternativa senza il campo per eventuali richieste d'esame.

Infine, ho curato la gestione di segreti e variabili d'ambiente: le credenziali di PostgreSQL e la configurazione dell'SMTP fittizio (MailHog) sono state centralizzate in un file *.env*, migliorando sicurezza e portabilità.

### *Modalità d'uso nel progetto*

Le API sono state progettate, testate e documentate all'interno di un flusso unico e integrato: la definizione in Django/DRF viene seguita dalla generazione automatica della documentazione tramite *drf-spectacular* e dalla successiva esportazione del file *openapi.yaml* presente all'interno della cartella *docs/* della repository. In questo modo si garantisce coerenza tra implementazione e documentazione, riducendo il rischio di discrepanze.

L'intero ambiente di esecuzione è inoltre avviabile tramite Docker, includendo i servizi web, database e MailHog. Ciò consente al valutatore di riprodurre facilmente tutte le principali funzionalità — login, consultazione dei movimenti, esecuzione di bonifici e gestione dei goals — senza dover effettuare configurazioni manuali, a beneficio della rapidità di test e valutazione.

## **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):

L'obiettivo è realizzare una piccola piattaforma di banking digitale API-based che consenta a un utente di registrarsi, autenticarsi e operare sul conto corrente (consultazione movimenti, esecuzione bonifici, gestione rubrica e obiettivi di risparmio), con backend RESTful in Python (Django + DRF) e un'interfaccia web essenziale in HTML/CSS/JS.

### *Obiettivi specifici raggiunti — funzionali.*

- Onboarding sicuro: registrazione con attivazione via email e login JWT (access/refresh).
- Conto e movimenti: visualizzazione del conto dell'utente (IBAN, saldo) e storico transazioni ordinato per data.
- Bonifico/trasferimento: esecuzione di trasferimenti verso IBAN interni con controllo del saldo e registrazione dei movimenti.
- Rubrica beneficiari: creazione/lista contatti (nome, IBAN, email).
- Obiettivi di risparmio (Goals): creazione, modifica, cancellazione e versamenti sul goal, con progresso calcolato.
- Estratti conto: consultazione movimenti per statement mensile.

### *Obiettivi specifici raggiunti — tecnici/non funzionali.*

- Architettura API-first: esposizione REST coerente (verbi HTTP, codici 4xx/5xx/2xx).
- Sicurezza: password hashate, autenticazione JWT Bearer, permessi object-level (un utente vede solo i propri dati).
- Consistenza: operazioni critiche (bonifico, saldi) dentro transazioni ACID; movimenti append-only per audit.
- Documentazione: Swagger / OpenAPI generato con drf-spectacular e allegato come openapi.yaml.
- Replicabilità: ambiente Docker (web, Postgres, MailHog) con avvio da docker-compose.
- Manutenibilità: modelli OOP chiari (User, BankAccount, Transaction, GoalsSaving, ecc.) e separazione tra serializer, view e servizi applicativi.

### *Allineamento con la traccia.*

- Settore finanziario: caso d'uso bancario reale (gestione conto, movimenti, bonifici, risparmio).
- Full-stack API-based: backend REST in Python (OOP) + interfaccia web; integrazione e test via Swagger UI.
- Rapporto richiesto: inclusi contesto, UML/ER, documentazione API, link al repository, descrizione degli snippet chiave e test funzionali (con screenshot).

### *Indicatori di esito (verifica).*

- Accesso alle API protette con token JWT valido.
- Bonifico con saldo insufficiente → 400 con messaggio chiaro; con saldo sufficiente → 201 e saldi aggiornati.
- Lista transazioni restituita paginata e ordinata per data.
- Creazione di un goal e incremento dell'importo attuale tramite versamento.
- Avvio locale riproducibile via Docker e consultazione della Swagger UI.

## **Contestualizzazione**

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

### *Contesto teorico*

L'elaborato si inserisce nel tema della digitalizzazione dell'impresa, con particolare attenzione ai servizi finanziari. L'approccio scelto è API-first: le funzionalità di business vengono esposte tramite API REST stateless, consumabili da applicazioni web. Il backend è sviluppato in Python in ottica object-oriented, utilizzando Django e Django REST Framework, con una netta separazione tra modello dati, serializer, viste e, quando necessario, servizi applicativi.

I principali principi architetturali adottati sono:

**Sicurezza:** password hashate, autenticazione tramite JWT Bearer e autorizzazioni a livello di oggetto, così che ogni utente possa accedere esclusivamente alle proprie risorse.

**Consistenza:** le operazioni critiche (ad esempio i bonifici o l'aggiornamento dei saldi) sono racchiuse in transazioni ACID, mentre i movimenti vengono registrati in modalità append-only per garantire auditabilità e tracciabilità.

**Chiarezza dell'interfaccia:** utilizzo coerente di risorse, verbi HTTP e codici di stato standard; la documentazione OpenAPI/Swagger è generata automaticamente per mantenere allineamento costante con il codice.

**Manutenibilità e replicabilità:** adozione di Docker per garantire ambienti coerenti, PostgreSQL come database transazionale e MailHog per simulare i flussi email senza dipendere da servizi esterni.

Sono stati inoltre considerati aspetti pratici come l'idempotenza (tramite external\_id nelle transazioni importate) e la gestione di campi calcolati, ad esempio il progresso dei goals di risparmio.

### Contesto applicativo

Il progetto riproduce un caso d'uso bancario essenziale: un cliente crea un account, si autentica e utilizza servizi di base. In particolare:

**Onboarding:** registrazione con attivazione via email e login tramite JWT.

**Conto e movimenti:** consultazione del conto corrente (IBAN, saldo) e dello storico delle transazioni ordinate per data.

**Trasferimenti (bonifici):** invio verso IBAN interni, con validazioni sul saldo disponibile e sulla titolarità del conto.

**Rubrica beneficiari:** salvataggio e riutilizzo dei destinatari per agevolare i futuri pagamenti.

**Risparmio personale:** creazione di goals con target economico e versamenti progressivi, con calcolo della percentuale di avanzamento.

**Estratti conto:** accesso ai movimenti aggregati per mese e anno.

L'applicazione è stata progettata per essere facilmente avviabile e verificabile tramite Docker Compose (web, database e MailHog), così da mostrare concretamente come le scelte teoriche ( REST, sicurezza, transazioni ) si riflettano in un servizio funzionante. Alcuni aspetti più avanzati (KYC completo, pagamenti istantanei, multivaluta, AML) sono rimasti fuori dal perimetro didattico, ma il design è stato mantenuto estendibile in vista di possibili evoluzioni future.

### Descrizione dei principali aspetti progettuali

(Sviluppare l'elaborato richiesto dalla traccia prescelta):

#### Architettura generale.

Il progetto è stato sviluppato adottando un approccio API-first: il backend, realizzato in Python con Django e Django REST Framework, espone una serie di servizi REST che vengono consumati dall'interfaccia web. La struttura del codice segue l'organizzazione a più applicazioni indipendenti (*users, accounts, transactions, api*), così da favorire modularità e manutenibilità.

Ogni applicazione presenta una chiara separazione tra modelli, serializer e view, mentre per le operazioni critiche come l'esecuzione di un bonifico è stato introdotto un service layer leggero, con l'obiettivo di isolare la logica di business e rendere il codice più leggibile e facilmente estendibile. Questa impostazione consente di mantenere un'architettura pulita e coerente, facilitando sia lo sviluppo che l'eventuale evoluzione futura della piattaforma.

#### Modello dati.

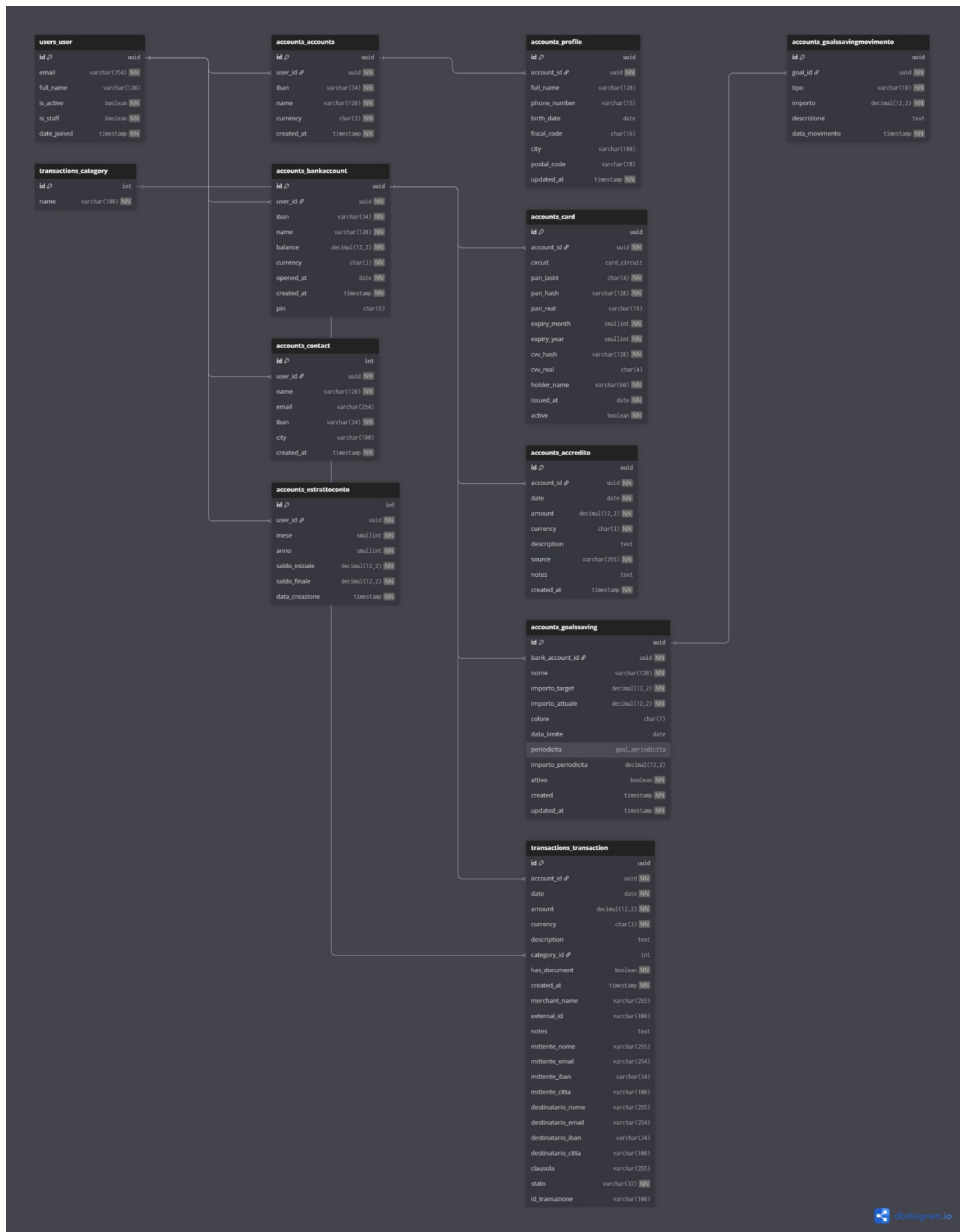
Le entità principali sono: User, BankAccount (IBAN, saldo, valuta), Transaction (importo firmato, categoria, controparte), Contact (rubrica per utente), GoalsSaving e GoalsSavingMovimento, più Accounts/Profile per i dati anagrafici, Card per le carte, Accredito ed EstrattoConto.

Vincoli importanti: IBAN univoco (viene memorizzato normalizzato ovvero senza spazi e gli input sono normalizzati in validazione per garantire il match univoco); Contact unico per (user, iban); GoalsSaving unico per (bank\_account, nome). Le transazioni sono append-only (storni come movimenti inversi). In Transaction uso external\_id UNIQUE per l'idempotenza negli import o nei retry.

#### Schema ER

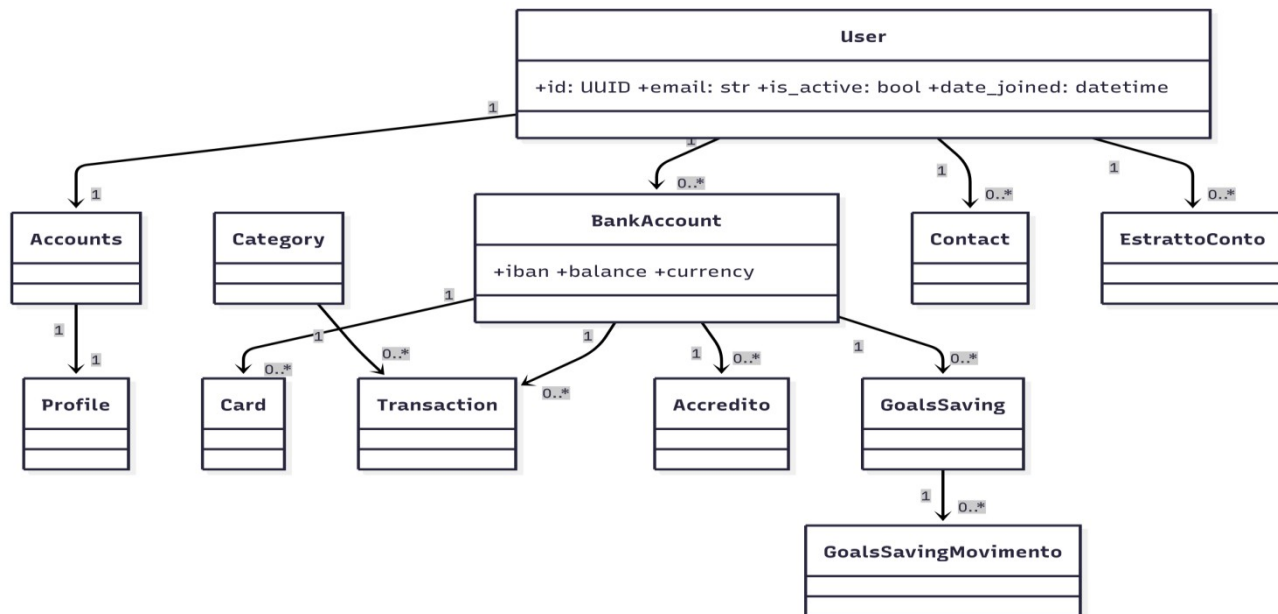
Lo schema er è stato generato utilizzando dbdiagram.io (per una visione più nitida dello schema si consiglia di visionarlo all'interno della cartella docs della repository, il link è presente alla fine del file)





## Schema UML

Lo schema uml è stato generato utilizzando mermaid (diagramming and charting tool)



### Logica applicativa.

- **Onboarding:** registrazione utente, attivazione via email (testata con MailHog), login con JWT.

```

from rest_framework_simplejwt.serializers import TokenObtainPairSerializer
from rest_framework_simplejwt.views import TokenObtainPairView
from django.utils.timezone import now

```

```

class CustomTokenObtainPairSerializer(TokenObtainPairSerializer):
    def validate(self, attrs):
        data = super().validate(attrs)
        self.user.last_login = now()
        self.user.save(update_fields=['last_login'])
        return data

```

```

class CustomTokenObtainPairView(TokenObtainPairView):
    serializer_class = CustomTokenObtainPairSerializer

```

Perché è utile: consolida l'onboarding lato auth, l'endpoint JWT produce access/refresh token e aggiorna in modo tracciabile last\_login, utile per audit e UX (es. "ultimo accesso"). L'approccio è stateless e adatto a un'architettura API-first web/mobile.

- **Conti:** ogni utente vede solo i propri conti (attualmente la piattaforma è in grado gestire un unico conto per account ma facilmente estendibile a più conti con un futuro aggiornamento).

```

from rest_framework.generics import ListAPIView
from rest_framework.permissions import IsAuthenticated
from accounts.models import BankAccount
from accounts.serializers import BankAccountSerializer

```

```

class UserBankAccountListView(ListAPIView):
    serializer_class = BankAccountSerializer
    permission_classes = [IsAuthenticated]
    pagination_class = None
    ordering = ['-created_at']

    def get_queryset(self):
        return BankAccount.objects.filter(user=self.request.user)

```

Perché è utile: applica un object-level scoping lato queryset: anche se l'utente prova a passare parametri arbitrari, il server restituisce solo i conti del request.user. È semplice da testare e difende dall'esfiltrazione di dati.

- **Bonifico/trasferimento:** tramite un servizio applicativo eseguo controlli (proprietà conto, saldo > importo, IBAN valido) e aggiorno i saldi dentro una transazione DB: addebito (importo negativo) al mittente e, se l'IBAN è interno, accredito (positivo) al destinatario.

#### # VALIDAZIONI BUSINESS

```
class TransferSerializer(serializers.Serializer):
    amount = serializers.DecimalField(max_digits=12, decimal_places=2)
    description = serializers.CharField(allow_blank=True, required=False)
    category = serializers.CharField(allow_blank=True, required=False)
    clause = serializers.CharField(allow_blank=True, required=False)
    pin = serializers.CharField(max_length=6)
    to_name = serializers.CharField(allow_blank=True, required=False)
    to_email = serializers.EmailField(allow_blank=True, required=False)
    to_iban = serializers.CharField(max_length=34)
    to_city = serializers.CharField(allow_blank=True, required=False)

    def validate(self, data):
        user = self.context['request'].user
        from_account = BankAccount.objects.filter(user=user).first()
        if not from_account:
            raise serializers.ValidationError("Conto mittente non trovato.")
        try:
            to_account = BankAccount.objects.get(iban=data['to_iban'])
        except BankAccount.DoesNotExist:
            raise serializers.ValidationError("Conto destinatario non trovato.")

        amount = data['amount']
        if amount <= 0:
            raise serializers.ValidationError("L'importo deve essere positivo.")
        if from_account.balance < amount:
            raise serializers.ValidationError("Saldo insufficiente sul conto mittente.")
        if from_account.pin != data['pin']:
            raise serializers.ValidationError("PIN errato.")

        data['from_account'] = from_account
        data['to_account'] = to_account
        return data
```

#### # ESECUZIONE ATOMICA + DOPPIO MOVIMENTO (uscita/entrata)

```
from django.utils import timezone
from django.db import transaction as db_transaction

def create(self, validated_data):
    from_account = validated_data['from_account']
    to_account = validated_data['to_account']
    amount = validated_data['amount']
    description = validated_data.get('description', '')
    clause = validated_data.get('clause', '')
    category_name = validated_data.get('category', '')
    date = timezone.now().date()

    category_obj = None
    if category_name:
        category_obj, _ = Category.objects.get_or_create(name=category_name)

    try:
        mittente_citta = Accounts.objects.get(user=from_account.user).profile.city
    except Exception:
        mittente_citta = ''
```

```

with db_transaction.atomic():
    tx_out = Transaction.objects.create(
        account=from_account, date=date, amount=-amount, currency='EUR',
        description=description, category=category_obj, notes=clause,
        merchant_name=str(to_account),
        mittente_nome=from_account.name, mittente_email=from_account.user.email,
        mittente_iban=from_account.iban, mittente_citta=mittente_citta,
        destinatario_nome=to_account.name, destinatario_iban=to_account.iban,
        clausola=clause, stato='Completata', id_transazione="",
    )

    tx_in = Transaction.objects.create(
        account=to_account, date=date, amount=amount, currency='EUR',
        description=description, category=category_obj, notes=clause,
        merchant_name=str(from_account),
        mittente_nome=from_account.name, mittente_email=from_account.user.email,
        mittente_iban=from_account.iban, mittente_citta=mittente_citta,
        destinatario_nome=to_account.name, destinatario_iban=to_account.iban,
        clausola=clause, stato='Completata', id_transazione="",
    )

    from_account.balance -= amount
    to_account.balance += amount
    from_account.save(); to_account.save()

return {'tx_out': tx_out, 'tx_in': tx_in}

```

Perché è utile: il serializer impone le regole di dominio (conto mittente dell'utente, saldo sufficiente, PIN corretto, IBAN valido). La create() esegue il bonifico in modo ACID, registrando due transazioni speculari (addebito/accredito) e aggiornando i saldi nella stessa transazione DB.

- **Goals:** creo un obiettivo con `importo_target`; i versamenti creano movimenti dedicati e aggiornano `importo_attuale`; percentuale e rimanente sono calcolati.

```

# accounts/models.py
class GoalsSaving(models.Model):
    # ... campi omessi
    def aggiungi_versamento(self, importo, descrizione="Versamento"):
        from decimal import Decimal
        if not isinstance(importo, Decimal):
            importo = Decimal(str(importo))
        if importo <= 0:
            raise ValueError("L'importo deve essere positivo")

        movimento = GoalsSavingMovimento.objects.create(
            goal=self,
            tipo=GoalsSavingMovimento.TipoMovimento.VERSAMENTO,
            importo=importo,
            descrizione=descrizione
        )
        self.importo_attuale += importo
        self.save(update_fields=['importo_attuale', 'updated_at'])
        return movimento

```

```

# api/views.py
from rest_framework import permissions
from rest_framework.views import APIView
from rest_framework.response import Response

```

```

from django.db import transaction

class GoalsSavingAddMoneyView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def post(self, request, pk):
        try:
            goal = GoalsSaving.objects.get(pk=pk, bank_account__user=request.user)
        except GoalsSaving.DoesNotExist:
            return Response({'detail': 'Obiettivo non trovato.'}, status=404)

        importo = request.data.get('importo')
        descrizione = request.data.get('descrizione', 'Versamento manuale')
        # ...validazioni su importo...

        with transaction.atomic():
            movimento = goal.aggiungi_versamento(importo, descrizione)
            goal.refresh_from_db()
            return Response({
                'movimento': GoalsSavingMovimentoSerializer(movimento).data,
                'goal_aggiornato': GoalsSavingSerializer(goal).data
            }, status=201)

```

Perché è utile: il metodo di dominio incapsula la regola di aggiornamento (crea movimento, incrementa importo\_attuale), la view garantisce consistenza con transaction.atomic() e restituisce lo stato aggiornato (utile al frontend per percentuale e rimanente).

### Progettazione delle API.

Le rotte seguono convenzioni REST (plurali, verbi HTTP, codici di stato coerenti). Esempi:

- Auth: POST /auth/jwt/create, .../refresh, .../verify
  - Accounts: GET /api/accounts, GET /api/accounts/me
  - Transazioni: GET /api/transactions, GET /api/transactions/{id}
  - Trasferimento: POST /api/transfer
  - Rubrica: GET/POST /api/accounts/contacts, DELETE /api/accounts/contacts/{id}
  - Goals: GET/POST /api/goals-saving, GET/PATCH/DELETE /api/goals-saving/{id}
- La documentazione è generata con drf-spectacular (Swagger UI/ReDoc) ed esportata in OpenAPI (openapi.yaml).

### Documentazione delle API.

La documentazione è generata automaticamente con drf-spectacular (OpenAPI 3) ed è pubblicata sui seguenti endpoint:

- Swagger UI: /docs/
- ReDoc: /redoc/
- Schema OpenAPI (JSON/YAML): /api/schema/

Nota per Swagger UI: bisogna cliccare su Authorize → incollare solo l'access token (la UI premette "Bearer " automaticamente). In un client esterno usare l'header:

Authorization: Bearer <access\_token>

### Autenticazione (Djoser + SimpleJWT) — prefisso /auth/

Metodo	Path	Descrizione	
POST	/auth/jwt/create	Login → access, refresh	
POST	/auth/jwt/refresh	Refresh token	
POST	/auth/jwt/verify	Verifica token	
POST	/auth/users/	Registrazione	



POST	/auth/users/activation/	Attivazione via token
POST	/auth/users/reset_password/	Avvio reset password

### Accounts e profilo — prefisso /api/accounts/

Metodo	Path	Descrizione	Auth
GET	/api/accounts/	Lista conti dell'utente	Bearer
GET	/api/accounts/{id}	Dettaglio conto (owner)	Bearer
GET	/api/accounts/me	Dati utente + conti collegati	Bearer
GET/POST	/api/accounts/contacts	Rubrica: lista/crea contatto	Bearer
DELETE	/api/accounts/contacts/{id}	Elimina contatto	Bearer

### API applicative varie — prefisso /api/

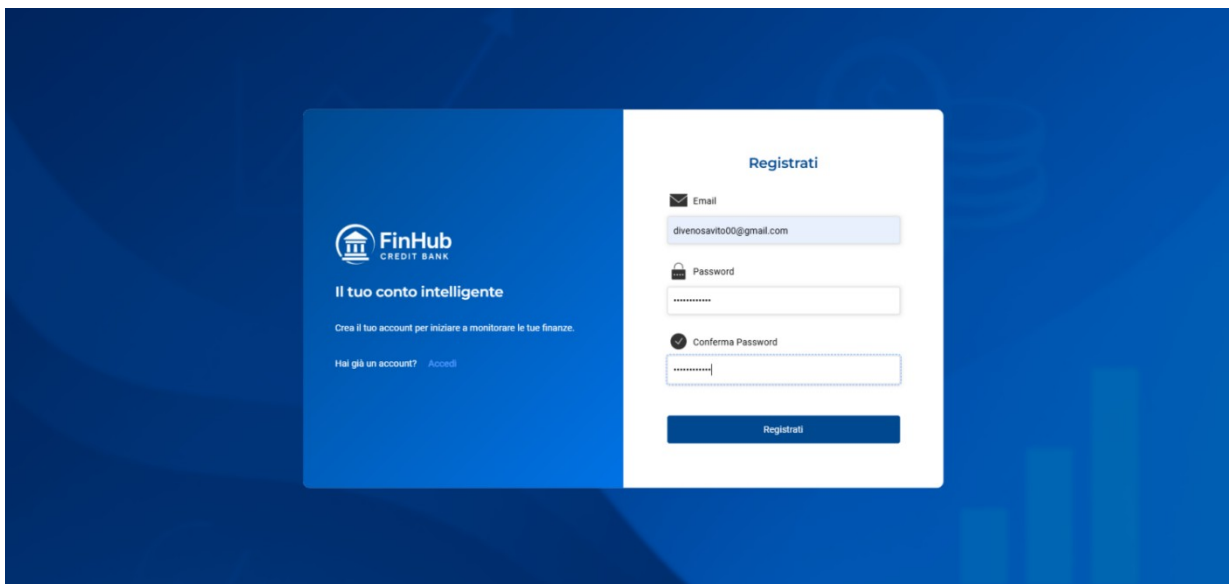
Metodo	Path	Descrizione
GET	/api/transactions	Lista transazioni (paginata, ordinate)
GET	/api/transactions/{id}	Dettaglio transazione
		Bonifico atomico
POST	/api/transfer	{from_account,to_iban,amount,pin,..}
		..}
GET/POST	/api/goals-saving	Lista/crea obiettivi
GET/PATCH/DELETE	/api/goals-saving/{id}	Dettaglio/aggiorna/elimina goal
POST	/api/goals-saving/{id}/add-money	Versamento sul goal
GET	/api/estratto-conto/	Estratti conto mensili
GET	/api/estratto-conto/{estratto_id}/movimenti/	Estratti conto movimenti

### Allegati

- docs/openapi.yaml incluso nel repository.
- Screenshot di Swagger UI e ReDoc inclusi nella repository nel percorso docs/".

### Test Funzionale

Questa sezione mostra alcune delle funzionalità principali durante l'utilizzo di routine mediante alcuni screenshot



Attiva il tuo account su FinHub Credit Bank [Posta in arrivo](#) x

 finhubemailservice@gmail.com

Ciao [divenosavito00@gmail.com](#).

Grazie per esserti registrato su FinHub Credit Bank.

Per attivare il tuo account, clicca sul pulsante qui sotto:


[Attiva il tuo account](#)



Cordiali Saluti

Il team leader di FinHub Credit Bank, Vito

Se non hai richiesto questa email, ignorala pure.


[← Rispondi](#) [→ Inoltra](#) [🗑](#)


 **FinHub**  
CREDIT BANK


7 Help Center  


Bentornato,  
**Vito di venosa**


Operazioni


 **Bonifico Sepa**


 **Giroconto Interno**


 **Ricarica prepagata**

 **Goal Saving**

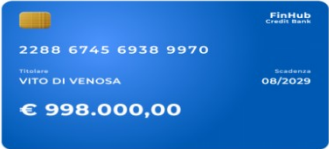
 **Bolletta veloce (QR)**

 **Blocco carta**

 **Cambio PIN online**

 **Piano PAC**


Credit Card



2288 6745 6938 9970

VITO DI VENOSA

€ 998.000,00




View Transactions


888

Il Mondo FinHub


Elenco Movimenti

 [Vedi di più](#)


Elenco Spese e transizioni

 [Vedi di più](#)


Suddivisione Spese

 [Vedi di più](#)


Statistiche

 [Vedi di più](#)


Assicurazioni



Finanziamenti






Investimenti Eco



## Bonifico SEPA

[Form Bonifico](#)

 **FinHub**  
CREDIT BANK

7 Help Center  


Checkout

[Rubrica](#)

Dati del mittente

Nome entità / persona

Email:




divenosavito00@gmail.com


Dati del destinatario

Nome entità / persona:

Email:




Mario rossi




mariorossi@gmail.com

IBAN:

Città di residenza:




IT60 0542 8111 0197 9502 2210 18




IBAN:

Città di residenza:



4539148803436467




Milano

Altre specifiche


Clausola:

Categoria:


Importo:



preventivo sito web



Web design



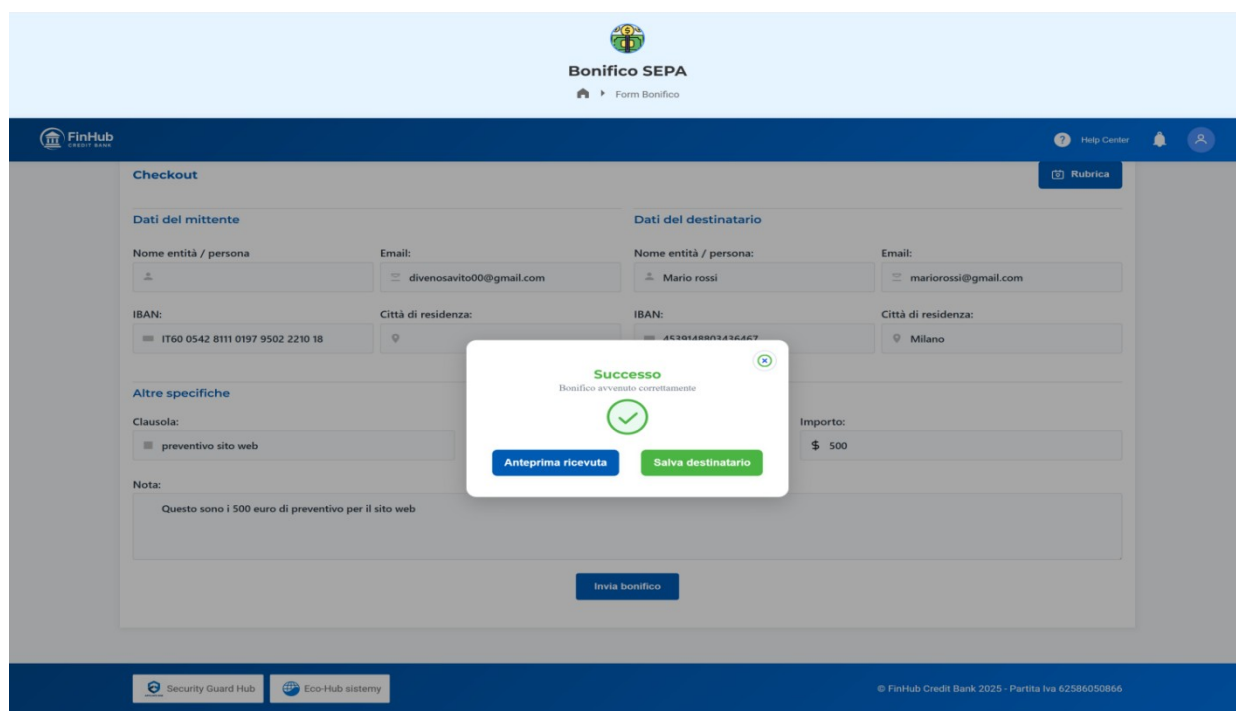
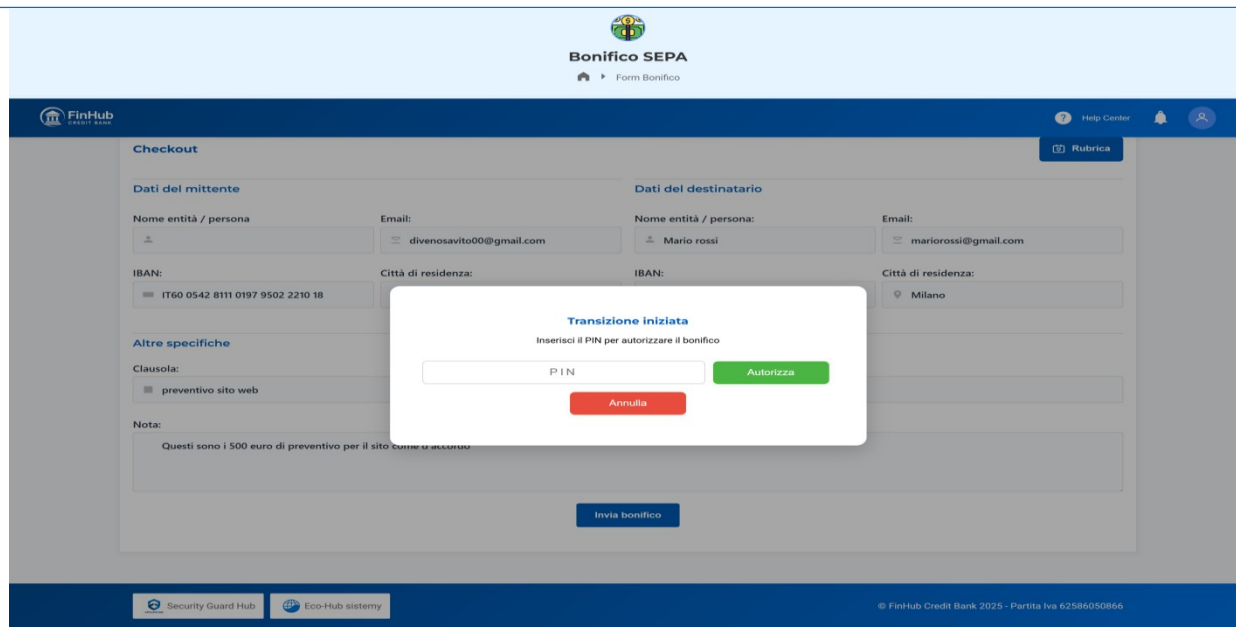
\$ 500

Nota:

Questi sono i 500 euro di preventivo per il sito come d'accordo

Invia bonifico





L'elenco degli screensht completi è presente nella repository di github del progetto esattamente nella cartella docs/screenshot/

### Sicurezza

Il sistema implementa meccanismi di sicurezza basati su password hashate tramite AbstractBaseUser e autenticazione JWT Bearer per l'accesso alle rotte protette. Sono stati inoltre configurati controlli a livello di oggetto per impedire accessi incrociati tra utenti.

Per quanto riguarda le politiche CORS, nel contesto attuale il frontend e l'API risiedono sullo stesso host e non è quindi necessaria alcuna configurazione specifica; in caso di deployment su domini differenti è comunque possibile abilitare il pacchetto *django-cors-headers*.

### Interfaccia utente

È stata realizzata una UI web essenziale in HTML/CSS/JS che copre le principali funzionalità: registrazione e login, dashboard con conti e movimenti, form per l'esecuzione dei bonifici (con selezione dell'IBAN dalla rubrica), schermate dedicate alla creazione e gestione dei goals di risparmio e alla consultazione degli estratti conto, oltre a una sezione per la suddivisione delle spese e la visualizzazione di statistiche.

### Ambiente e deploy

L'intero ambiente è avviabile tramite Docker, includendo il servizio web (Django), il database PostgreSQL e MailHog per la gestione delle email di attivazione. L'uso di docker-compose garantisce un setup ripetibile e portabile, con la

configurazione centralizzata in un file .env per la gestione delle variabili sensibili.

### *Qualità e test*

I flussi principali dell'applicazione sono stati verificati tramite Swagger UI, e in alcuni casi con Postman. Sono stati testati il login, la lettura dei conti, la consultazione delle transazioni, i bonifici sia con saldo sufficiente che insufficiente, e le operazioni sui goals (creazione e versamenti). Gli screenshot delle prove sono allegati alla sezione dei test funzionali del report.

### *Scelte e trade-off*

È stata volutamente adottata una logica di dominio essenziale ma estendibile. Alcuni aspetti avanzati come KYC completo, pagamenti istantanei o multivaluta sono stati esclusi in quanto fuori perimetro didattico. Tuttavia, il design scelto (service layer, uso delle transazioni a livello di database, idempotenza) è stato concepito per permettere un'eventuale estensione futura senza dover stravolgere l'architettura esistente.

## **Campi di applicazione**

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

### *Ambiti d'uso principali*

Il progetto trova collocazione in diversi scenari applicativi. In ambito bancario retail o per una neobank rappresenta un MVP in grado di gestire conti, movimenti e bonifici attraverso un'interfaccia web, facilmente estendibile al mobile mantenendo la stessa logica API. Nel settore fintech/PFM (Personal Finance Management) le funzionalità di analisi delle spese e degli obiettivi di risparmio risultano riutilizzabili in applicazioni di budgeting o di consulenza finanziaria. L'approccio API-first consente inoltre l'integrazione con terze parti in contesti di open banking, dove la documentazione OpenAPI agevola l'onboarding di partner e l'interfacciamento con gateway, core banking o provider di pagamento. Grazie a Docker e MailHog, la piattaforma si presta bene anche come ambiente didattico e di testing, utile per studiare flussi di onboarding, autenticazione JWT e transazioni ACID senza dipendenze esterne. Infine, nelle PMI può fungere da strumento per digitalizzare processi base, grazie a funzionalità come rubrica beneficiari, estratti conto e tracciabilità dei movimenti.

### *Vantaggi per l'organizzazione*

Dal punto di vista organizzativo, i benefici sono molteplici. Il time-to-market risulta ridotto grazie alla coerenza delle API REST e all'uso di Swagger, che permettono lo sviluppo parallelo di front-end e mobile. L'integrazione è facilitata dallo schema OpenAPI, che standardizza i contratti e riduce il rischio di errori tra team e sistemi. L'affidabilità è garantita dalla logica append-only dei movimenti e dall'uso delle transazioni per l'aggiornamento dei saldi, assicurando uno storico verificabile e la consistenza contabile. La sicurezza è progettata by-design tramite password hashate, autenticazione JWT, permessi object-level e separazione dei dati sensibili. Infine, la scalabilità è supportata dall'adozione di servizi stateless e dalla containerizzazione, che rendono riproducibili gli ambienti di sviluppo, staging e produzione.

### *Vantaggi per l'utente finale*

Dal punto di vista dell'utente finale la piattaforma offre trasparenza e controllo, attraverso una dashboard con conto e storico delle transazioni, e la possibilità di consultare estratti conto mensili. L'operatività risulta semplice e rapida: la rubrica evita errori di digitazione dell'IBAN e i messaggi di errore sono chiari (ad esempio saldo insufficiente). Sul piano dell'educazione finanziaria, gli strumenti di risparmio (goals) consentono di monitorare il progresso e l'importo residuo, aiutando a pianificare spese e accantonamenti.

### *Estensioni naturali*

Il design è stato concepito per rimanere estendibile. Tra le possibili evoluzioni figurano i pagamenti istantanei, le notifiche push, la gestione multivaluta, procedure di KYC avanzato/AML, l'introduzione di limiti dinamici di operatività e l'integrazione con sistemi di esportazione contabile. Tutte queste funzionalità possono essere aggiunte senza dover riscrivere l'architettura esistente.

Oltre al valore didattico, il progetto può quindi trovare applicazione concreta come prototipo per piccole realtà finanziarie o startup fintech interessate a sperimentare architetture API-first. La modularità del design lo rende facilmente adattabile a scenari di open banking o a soluzioni di contabilità interna per PMI, fungendo da laboratorio sicuro per testare e introdurre nuove funzionalità.

## Valutazione dei risultati

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

### *Punti di forza (potenzialità)*

Il progetto presenta diversi elementi di solidità:

**Aderenza alla traccia:** backend REST in Python (Django + DRF), interfaccia web, dominio finanziario reale (conti, movimenti, bonifici, goals), documentazione Swagger/OpenAPI e repository completo e front-end con Html,Css e javascript.

**Architettura pulita e replicabile:** approccio API-first, separazione tra modelli, serializer e view, ambienti Docker (web, Postgres, MailHog) avviabili rapidamente.

**Sicurezza di base corretta:** password hashate, JWT Bearer per le rotte protette, permessi object-level che garantiscono isolamento dei dati.

**Coerenza e auditabilità:** movimenti append-only e aggiornamento dei saldi all'interno di transazioni DB, così da evitare stati incoerenti.

**Documentazione:** Swagger UI/ReDoc e OpenAPI esportabile, a supporto di test, valutazione e integrazioni.

**Estendibilità funzionale:** domain model già predisposto per funzionalità aggiuntive (rubrica, goals saving, estratti conto) e per integrazioni esterne (gateway di pagamento, open banking).

### *Limiti attuali (criticità)*

Sono stati individuati alcuni limiti che rappresentano al contempo aree di miglioramento:

**UI essenziale:** front-end minimale (HTML/CSS/JS), funzionale ma privo di validazioni client avanzate e grafici interattivi.

**Sicurezza avanzata assente:** mancano rate limiting, blocco tentativi di login/bruteforce, audit log amministrativi, policy di password configurabili.

**Bonifico esterno simulato:** corretta scrittura dei movimenti, ma senza integrazione reale con circuiti SEPA/instant né riconciliazione automatica.

**Gestione dati sensibili:** esistono campi hashati corretti (pan\_hash, cvv\_hash), ma l'uso di dati reali va evitato o comunque gestito con cifratura e mascheramento.

**Osservabilità limitata:** assenza di log strutturati, metriche e alerting (APM, healthcheck estesi).

**Test automatici parziali:** presenti prove manuali e alcuni unit test, ma la copertura potrebbe essere ampliata su bonifici, permessi e edge case.

### *Rischi e impatto*

**Incoerenze contabili** (bassa probabilità, medio impatto): mitigazione → uso costante di transaction.atomic() e external\_id univoci.

**Sovraccarico/abusi sugli endpoint** (medio): mitigazione → aggiunta di throttling DRF, captcha o blocco progressivo sul login.

**Fuga di informazioni sensibili** (bassa probabilità ma impatto critico): mitigazione → cifratura campi, controlli granulari e logging sicuro.

### *Evidenze del funzionamento*

**Onboarding completo:** registrazione → attivazione via email (MailHog) → login JWT.



**Operatività base:** dati conto, storico transazioni ordinato, bonifico con gestione saldo e risposte HTTP coerenti.

**Goals saving:** creazione obiettivi, versamenti, calcolo percentuale/rimanente e suddivisione spese.

**Documentazione:** Swagger UI navigabile e OpenAPI allegata al report.

### *Migliorie rapide (roadmap breve)*

- Abilitare throttling DRF su /auth e /api/transfer con blocco su login multipli.
- Ampliare la copertura dei test automatici su bonifici, permessi cross-utente e idempotenza.
- Migliorare la sicurezza (cifatura campi sensibili, revisione CORS per produzione).
- Potenziare la UI (feedback immediati, filtri, paginazione visibile).

### *Sintesi*

Il progetto risponde pienamente alla traccia (full-stack API-based nel settore finanziario) e presenta una base solida, sicura e replicabile. Le principali aree di miglioramento riguardano la sicurezza avanzata (rate limiting, audit) e l'esperienza utente del front-end. Questi interventi sono incrementali e non richiedono di modificare l'architettura, che è già predisposta a evoluzioni future.

### *Collegamento ai criteri di valutazione*

#### *Pianificazione delle fasi*

Il progetto è stato gestito attraverso una pianificazione dettagliata, che ha previsto fasi distinte (analisi, creazione repository, login/registrazione, bonifici, movimenti, goals saving, test). Ogni fase è stata descritta con tempistiche precise (giorni/ore dedicate), garantendo una chiara roadmap di sviluppo. Questo livello di dettaglio corrisponde allo standard "avanzato" in termini di pianificazione.

#### *Organizzazione delle risorse*

Sono state utilizzate risorse aggiornate, attuali e appropriate: Django 5.2, Django REST Framework, PostgreSQL, Docker, MailHog, drf-spectacular per la documentazione API, GitHub per versionamento e CI/CD. La scelta è stata guidata da affidabilità, attinenza al dominio finanziario e possibilità di replicare l'ambiente in maniera innovativa (container e workflow automatizzati). Inoltre, va citato l'utilizzo dell'intelligenza artificiale per creare il design della carta di credito del conto a livello grafico.

#### *Individuazione degli obiettivi*

Gli obiettivi sono stati chiari e misurabili sin dall'inizio: registrazione e autenticazione sicura, gestione conti e movimenti, bonifici, rubrica, estratti conto, goals saving. Ogni obiettivo ha guidato lo sviluppo delle funzionalità e la validazione finale, con allineamento diretto ai requisiti della traccia.

#### *Originalità*

Il progetto non si limita a riprodurre una semplice piattaforma bancaria, ma introduce elementi innovativi come la suddivisione delle spese in categorie e la gestione degli obiettivi di risparmio personalizzati (Goals Saving), funzionalità tipiche di app fintech moderne e non sempre presenti nei sistemi didattici di banking simulato. Ciò conferisce al lavoro un grado di originalità superiore.

#### *Accuratezza del prodotto*

Il progetto rispecchia in modo fedele la traccia, con un backend RESTful object-oriented, una UI funzionante, documentazione OpenAPI, schema ER e UML, e test funzionali. Alcuni aspetti sono stati addirittura ampliati (gestione rubrica, estratti conto PDF, goals saving), dimostrando una comprensione superiore delle richieste.

#### *Valutazione dei risultati*

Sono stati evidenziati con chiarezza punti di forza (architettura API-first, sicurezza con JWT, auditabilità dei movimenti, estendibilità del modello dati) e limiti attuali (UI essenziale, sicurezza avanzata non completa, copertura test ampliabile). Questa analisi riflette un pensiero critico articolato, con proposte di roadmap migliorative.

### *Ricchezza lessicale e padronanza del linguaggio disciplinare*

L'elaborato utilizza terminologia tecnica coerente e avanzata (RESTful, API-first, transazioni ACID, JWT, containerizzazione, idempotenza), dimostrando padronanza del linguaggio disciplinare informatico-gestionale e capacità di applicarlo al dominio finanziario.

### *Conclusioni*

Il progetto ha raggiunto con successo gli obiettivi definiti dalla traccia: è stata realizzata un'applicazione full-stack API-based per il settore finanziario, con autenticazione sicura, gestione del conto, trasferimenti, obiettivi di risparmio e generazione di estratti conto.

Sotto il profilo tecnico, il lavoro ha permesso di consolidare competenze in architetture RESTful, autenticazione JWT, gestione di transazioni ACID, containerizzazione con Docker e documentazione OpenAPI. Sul piano critico, ha evidenziato punti di forza (sicurezza, modularità, estensibilità) e alcune limitazioni (UI essenziale, test automatizzati ancora migliorabili, assenza di antifrode e KYC). Questi aspetti costituiscono al tempo stesso aree di miglioramento e spunti per sviluppi futuri.

L'esperienza ha consentito non solo di rispettare i requisiti accademici, ma anche di simulare un contesto vicino a casi d'uso reali nel settore fintech, favorendo l'acquisizione di competenze trasversali tecniche, organizzative e critiche di valore per il proseguimento del percorso formativo e professionale. Inoltre, il progetto si è rivelato particolarmente stimolante, permettendomi di mettere a frutto capacità creative e analitiche.

Dal punto di vista personale, lo sviluppo mi ha insegnato l'importanza della pianificazione accurata e della gestione dei task: ogni errore o bug affrontato è diventato occasione di crescita tecnica e metodologica, consolidando competenze preziose anche in ambito professionale.

### *Repository del progetto*

Il codice sorgente completo, comprensivo di file `source_code`, configurazioni Docker, test e documentazione Swagger/OpenAPI, è disponibile al seguente link GitHub:

<https://github.com/Vito290500/FinHubCreditBank>