|  |  |
| --- | --- |
| **Immagine che contiene Carattere, testo, bianco, logo  Il contenuto generato dall'IA potrebbe non essere corretto.** | |
|  |  |
|  |  |
|  |  |
|  | Vito di Venosa |
|  | 0312200363 |
| Immagine che contiene testo, Carattere, bianco, tipografia  Il contenuto generato dall'IA potrebbe non essere corretto. | 1 |
| Immagine che contiene testo, Carattere, bianco, tipografia  Il contenuto generato dall'IA potrebbe non essere corretto. | La digitalizzazione dell’impresa |
| Immagine che contiene testo, Carattere, bianco, tipografia  Il contenuto generato dall'IA potrebbe non essere corretto. | 6 |
| Immagine che contiene testo, Carattere, bianco, tipografia  Il contenuto generato dall'IA potrebbe non essere corretto. | Sviluppo di una applicazione full-stack API-based per un’impresa del settore finanziario |
| **Immagine che contiene testo, Carattere, bianco  Il contenuto generato dall'IA potrebbe non essere corretto.** | FinHub Credit Bank |
|  | |
|  | |
| 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 dalle fasi iniziali, precedenti alla codifica. 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. Tale competenza è stata inoltre utile per definire una timeline di riferimento, precisa e strutturata, che ha guidato l’intero processo. 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. | |
|  | |
| *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 è durato circa 3 giorni dedicando 2h ore giornaliere al progetto, in questa fase non ho avuto particolari difficoltà.  *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. * Creazione file docker.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/worflows/ 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 si che docker riconosca tutte le dipendeze 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 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 smtplib 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’authenticazione 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 rest della password e l’utilizzo della sessione authenticata 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 modifca 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.  *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 fittizzio e lo collega all’account registrato con relativo cvv. * 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 cosi 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.   *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 partendo dal 16/07 e terminando il 29/07 e l’unica difficoltà incontrata è stata il rendering del saldo iniziale e finale all’interno dei vari estratti conto e ho dovuto creare un algoritmo sfruttando la ricorsione per poter ottenere i valori del saldo iniziale e finale di ogni estratto conto mensile in maniera accurata.   *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 metiante 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 versamente automatico mensilmente inserito durante la creazione del goals savings. * creazione della logica che gestisce il versamente 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 e poi ho scritto e creato gli unitest necessari per testare il corretto funzionamento di tutte le api. Questa fase è durata circa 4 giorni, dal 02/07 al 06/08.  *Fase 11 fix bugs emersi dagli unitest*  In quetsa fase ho corretto tutti i bugs emersi dopo il run dei vari unitest creati e ho testato l’installazione su un altro compiuter della repository. Successivamente ho creato il filre 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 | |
|  | |
| *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 (incluso token\_blacklist); 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.   *Come ho individuato e reperito le risorse*   * Documentazione ufficiale (Django, DRF, SimpleJWT, drf-spectacular, PostgreSQL) e tutorial focalizzati su pattern comuni (login JWT, paginazione, filtri). * Confronto tra alternative (es. drf-yasg vs drf-spectacular), scegliendo quest’ultimo per supporto e manutenzione.   *Difficoltà incontrate e come le ho superate*   * Generazione schema OpenAPI: alcune APIView non avevano serializer\_class/queryset deducibile → ho aggiunto serializer\_class dove mancava e, per le view che filtrano su request.user, ho gestito il caso swagger\_fake\_view o impostato queryset = Model.objects.none() così da evitare errori in build. * Campi calcolati (es. percentuali dei goals) non tipizzati in Swagger → dichiarati con @extend\_schema\_field per mostrare il tipo corretto. * Parametri di path non tipizzati (UUID/int) → specificati esplicitamente nell’URL o annotati, così lo schema non “indovina” string generico. * docker-compose (warning su version:): il file usava version: "3.9"; Compose v2 la considera obsoleta → mantenuto il file funzionante e pronta la rimozione del campo se richiesto dall’ambiente d’esame. * Gestione segreti/variabili: configurate via .env (es. credenziali Postgres, host SMTP fittizio), con .env.example per chi corregge.   *Modalità d’uso nel progetto*   * Le API vengono progettate, testate e documentate nello stesso flusso: definizione in Django/DRF → generazione Swagger con drf-spectacular → esportazione openapi.yaml da allegare al report. * L’ambiente è avviabile via Docker (web, db, mailhog), così il valutatore può riprodurre login, consultazione movimenti, bonifico e goals senza configurazioni manuali. | |
|  | |
|  | |
| Realizzare una piccola piattaforma di banking digitale API-based che permetta a un utente di registrarsi, autenticarsi e operare su conti correnti (consultazione movimenti, esecuzione bonifici, gestione rubrica e obiettivi di risparmio), con backend RESTful in Python (Django + DRF) e una UI web essenziale in HTML/CSS/JS.  *Obiettivi specifici raggiunti — funzionali.*   * Onboarding sicuro: registrazione con attivazione via email e login JWT (access/refresh). * Conti e movimenti: visualizzazione dei conti dell’utente (IBAN, saldo) e storico transazioni ordinato per data. * Bonifico/trasferimento: esecuzione di trasferimenti verso IBAN interni/esterni con controllo saldo e registrazione dei movimenti. * Rubrica beneficiari: creazione/lista/eliminazione 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 (risorse al plurale, 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 conti, 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. | |
|  | |
| *Contesto teorico.*  L’elaborato nasce nel tema della digitalizzazione dell’impresa con particolare riferimento ai servizi finanziari. L’approccio seguito è API-first: le funzionalità di business sono esposte tramite API REST stateless, consumabili da web e mobile. Il backend è sviluppato in Python (OOP) con Django + Django REST Framework, con separazione chiara fra modello dati, serializer, viste e (dove serve) servizi applicativi.  I principi architetturali adottati sono:   * Sicurezza: password hashate, JWT Bearer per l’autenticazione, autorizzazioni object-level (un utente vede solo le proprie risorse). * Consistenza: operazioni critiche (es. bonifici e aggiornamento saldi) racchiuse in transazioni ACID; movimenti append-only per garantire audit e tracciabilità. * Chiarezza dell’interfaccia: risorse, verbi HTTP e codici di stato standard; OpenAPI/Swagger generato automaticamente per evitare discrepanze tra codice e documentazione. * Manutenibilità e replicabilità: uso di Docker per ambienti coerenti, PostgreSQL come database transazionale, e MailHog per testare i flussi email senza servizi esterni.  Sono stati considerati anche temi pratici come l’idempotenza (es. external\_id sulle transazioni importate) e la gestione di campi calcolati (progresso dei goals).   *Contesto applicativo.*  Il progetto modella un caso d’uso bancario essenziale: un cliente apre un account, si autentica e utilizza servizi di base. In particolare:   * Onboarding: registrazione con attivazione via email e login JWT. * Conti e movimenti: consultazione dei conti (IBAN, saldo) e dello storico delle transazioni ordinato per data. * Trasferimenti (bonifici): invio verso IBAN interni/esterni con validazioni (proprietà del conto, saldo sufficiente). * Rubrica beneficiari: salvataggio/riuso dei destinatari. * Risparmio personale: creazione di goals con target e versamenti, con percentuale di avanzamento. * Estratti conto: accesso ai movimenti per mese/anno.   L’applicazione è pensata per essere facile da avviare e verificare (compose: web, db, mailhog), e per mostrare in modo concreto come le scelte teoriche (REST, sicurezza, transazioni) si riflettano in un servizio funzionante. Alcuni aspetti avanzati (KYC completo, pagamenti istantanei, multivaluta, AML) sono fuori perimetro didattico ma il design resta estendibile verso tali evoluzioni. | |
|  | |
| *Architettura generale.*  Ho adottato un approccio API-first: il backend in Python/Django + Django REST Framework espone i servizi REST consumati dall’interfaccia web. La struttura del codice è a più app (users, accounts, transactions, api), con separazione tra modelli, serializer, view e un service layer leggero per le operazioni critiche (es. bonifico).  *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; 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    *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 (permessi object-level).   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/  *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 | | POST | /api/transfer | Bonifico atomico {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/statements | Estratti conto mensili |  *Allegati*  * docs/FibHub\_Credit\_bank\_API\_schema.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        L’elenco degli screensht completi è presente nella repository di github del progetto esattamente nella cartella docs/screenshot/  *Sicurezza.*  Password hashate (AbstractBaseUser), JWT Bearer per l’accesso alle rotte protette, controlli a livello di oggetto per impedire accessi incrociati, CORS configurato per il frontend. Per i dati carta uso solo hash (niente PAN/CVV in chiaro).  *Interfaccia utente.*  Una UI web semplice in HTML/CSS/JS che copre: login/registrazione, dashboard conti e movimenti, form per il bonifico (con IBAN da rubrica), schermate per i goals (creazione e versamenti) e per la consultazione degli estratti conto.  *Ambiente e deploy.*  Tutto è avviabile con Docker: web (Django), db (PostgreSQL), mailhog per le email di attivazione. Il setup è ripetibile via docker-compose, con variabili in .env.  *Qualità e test.*  Ho verificato i flussi chiave via Swagger UI (e Postman dove utile): login, lettura conti, elenco transazioni, bonifico OK/KO (saldo insufficiente), creazione goal e versamento. Gli screenshot sono inclusi nei test funzionali del report.  *Scelte e trade-off.*  Ho preferito un dominio essenziale ma estendibile: niente KYC avanzato, pagamenti istantanei o multivaluta (fuori perimetro didattico), ma il design (service layer, transazioni DB, idempotenza) permette di aggiungerli senza stravolgere l’architettura. | |
|  | |
| *Ambiti d’uso principali*   * Banca retail / neobank (MVP): gestione base di conti, movimenti e bonifici con UI web; * ideale come prototipo da estendere a mobile mantenendo le stesse API. * Fintech/PFM (Personal Finance Management): funzionalità di analisi spese e obiettivi * di risparmio riusabili in app di budgeting o advisor finanziario. * Open Banking / integrazioni terze parti: architettura API-first facilmente integrabile con   gateway, core banking o provider di pagamento; documentazione OpenAPI a supporto dell’onboarding partner.   * Ambiente didattico e di testing interno: grazie a Docker e MailHog, è un banco prova per flussi di onboarding, sicurezza JWT e transazioni ACID senza dipendenze esterne. * PMI / pagamenti interni semplificati: rubrica beneficiari, estratti conto e tracciabilità movimenti utili per piccole realtà che vogliono digitalizzare processi base.   *Vantaggi per l’organizzazione*   * Time-to-market ridotto: API REST coerenti + Swagger → sviluppo front-end e mobile in parallelo. * Integrazione facilitata: lo schema OpenAPI standardizza contratti e riduce errori tra team/sistemi. * Affidabilità e audit: movimenti append-only e aggiornamenti saldo in transazione → storico verificabile e consistenza contabile. * Sicurezza by-design: password hashate, JWT Bearer, permessi object-level, separazione dati sensibili (hash PAN/CVV). * Scalabilità organizzativa: servizi stateless e containerizzazione → ambienti riproducibili in sviluppo, staging e produzione.   *Vantaggi per l’utente finale*   * Trasparenza e controllo: dashboard conti, storico transazioni ordinato, estratti conto per mese/anno. * Operatività semplice e rapida: rubrica per evitare errori di IBAN, feedback chiari sugli errori (es. saldo insufficiente). * Educazione finanziaria e risparmio: goals con progresso e importo rimanente per pianificare spese/accantonamenti.   *Estensioni naturali*   * Pagamenti istantanei e notifiche push, multivaluta, KYC avanzato/AML, limiti dinamici di operatività, esporti contabili (CSV/MT940) → tutte evoluzioni compatibili con il design attuale senza riscrivere l’architettura. | |
|  | |
| *Punti di forza (potenzialità)*   * Aderenza alla traccia: backend REST in Python (Django + DRF), interfaccia web, dominio finanziario reale (conti, movimenti, bonifici, goals), documentazione Swagger/OpenAPI e repository completo. * Architettura pulita e replicabile: approccio API-first, separazione modelli/serializer/view, ambienti con Docker (web, Postgres, MailHog) avviabili rapidamente. * Sicurezza di base corretta: password hashate, JWT Bearer per le rotte protette, permessi object-level (l’utente vede solo i propri dati). * Coerenza e auditabilità: movimenti append-only; aggiornamento dei saldi all’interno di transazioni DB → riduce stati incoerenti “a metà”. * Documentazione viva: Swagger UI/ReDoc e OpenAPI esportabile → facilita test, valutazione e integrazioni future. * Estendibilità funzionale: domain model già pronto per feature aggiuntive (rubrica beneficiari, goals con movimenti dedicati, estratti conto), e per integrazioni (gateway pagamenti, open banking).   *Limiti attuali (criticità)*   * Doc API incompleta in alcuni punti: per alcune APIView lo schema richiesta/risposta non è esplicitato (es. POST /api/transfer/); la generazione OpenAPI può richiedere annotazioni (serializer\_class, @extend\_schema). * UI essenziale: front-end minimale (HTML/CSS/JS); UX funzionale ma non ottimizzata (mancano grafici avanzati, validazioni client, feedback in tempo reale). * Controlli di sicurezza avanzati non presenti: assenti rate limiting, blocco tentativi di login/bruteforce, audit log amministrativi, policy password configurabili. * Bonifico esterno simulato: la scrittura dei movimenti è corretta, ma non c’è integrazione reale con circuiti SEPA/instant; nessuna riconciliazione automatica con statement bancari. * Dati sensibili carte: esistono campi hashati corretti (pan\_hash, cvv\_hash), ma va evitato l’uso dei campi “reali” se presenti; se necessari, cifrarli e oscurarli sempre nella UI/API. * Osservabilità limitata: non ci sono log strutturati/metriche/alerting (niente APM, healthcheck estesi). * Test automatici parziali: presenti test e prove manuali via Swagger, ma la copertura automatica potrebbe essere ampliata (unit/integration su bonifici, permessi e edge case).   *Rischi e impatto*   * Incoerenze contabili (bassa probabilità, medio impatto) se si aggiorna il balance fuori da transazioni o senza idempotenza su retry: mitigazione → usare sempre transaction.atomic() e un external\_id UNIQUE sui movimenti importati. * Sovraccarico/abusi sugli endpoint auth/transfer (medio): mitigazione → throttling DRF, captcha o “lockout” progressivo sul login. * Fuga di informazioni sensibili (bassa ma critica): mitigazione → rimuovere/cifrare campi carta non necessari, controlli di autorizzazione granulari e logging sicuro.   *Cosa funziona già (evidenze)*   * Onboarding completo: registrazione → attivazione via email (MailHog) → login JWT. * Operatività base: elenco conti, storico transazioni ordinato, bonifico con gestione saldo e risposte HTTP coerenti. * Risparmio: creazione goal, versamento, calcolo percentuale/rimanente. * Documentazione: Swagger UI navigabile ed OpenAPI allegabile al report.   *Migliorie rapide (roadmap breve)*   1. Annotare le APIView con serializer\_class / @extend\_schema e tipizzare i parametri path/UUID → OpenAPI completo. 2. Abilitare throttling DRF su /auth e /api/transfer/ + blocco su tentativi multipli di login. 3. Aggiungere test automatici su: bonifico (OK/KO), permessi cross-utente, idempotenza. 4. Cifrare eventuali campi sensibili residui; rivedere CORS in ottica produzione. 5. Migliorare la UI (feedback, filtri, paginazione visibile, stato richieste).   *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 completezza della documentazione OpenAPI, la sicurezza avanzata (rate limiting, audit) e l’esperienza utente del front-end. Questi interventi sono incrementali e non richiedono di cambiare l’architettura, che è già adatta a evoluzioni produttive. 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 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). *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 dei conti, trasferimenti, obiettivi di risparmio e generazione di estratti conto.

Dal punto di vista tecnico, il lavoro ha permesso di applicare e consolidare competenze in architetture RESTful, autenticazione JWT, gestione di transazioni ACID, containerizzazione con Docker e documentazione OpenAPI. L’integrazione tra frontend e backend ha reso evidente l’importanza di un design modulare e scalabile.

Dal punto di vista critico, il progetto ha evidenziato punti di forza (sicurezza, modularità, estensibilità) e alcune limitazioni (UI essenziale, test automatizzati ancora migliorabili, mancanza di funzionalità avanzate di sicurezza bancaria come antifrode e KYC). Questi elementi costituiscono al tempo stesso aree di miglioramento e spunti per sviluppi futuri.

In sintesi, 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. Ciò ha favorito l’acquisizione di competenze trasversali — tecniche, organizzative e critiche — che rappresentano un valore aggiunto per il proseguimento del percorso formativo e professionale.