

Progetto ingegneria del software

Anno accademico: 2022-2023

Gruppo di lavoro:

Matteo Soldini matricola n° 1074601

Paolo Zanotti matricola n° 1074166

Samuel Gherardi matricola n° 1076850

Project Plan

Come modello principale per lo sviluppo del progetto seguiremo i seguenti 14 punti del Project Plan, stilati da tutto il team in fase iniziale di progettazione.

1. Introduzione

Siamo un gruppo di studenti di ingegneria informatica dell'università di Bergamo: Samuel Gherardi, Matteo Soldini e Paolo Zanotti. Abbiamo deciso di realizzare questo progetto in quanto sentiamo l'esigenza di tenere traccia delle spese fatte giornalmente e di pianificare la gestione delle entrate per il futuro (budget e obiettivi) in modo facile e veloce. Le applicazioni attualmente sul mercato non soddisfano completamente le nostre necessità, abbiamo quindi pensato di sviluppare una applicazione web che permetta di gestire le entrate ed uscite personali tramite interfaccia web.

2. Modello di processo

Come modello di processo abbiamo deciso di seguire il metodo Agile per lo sviluppo del software, sfruttando in particolare la funzione 'Projects' fornita da Github. In questo modo avremo a disposizione una Kanban intuitiva che ci permetterà di gestire i task come degli issues normali, dividendoli in TO DO, IN PROGRESS e DONE.

3. Organizzazione del progetto

Dato che il team è composto solo da 3 persone, non risulta necessaria la figura del Project Manager per coordinarci. Questo ci ha permesso di adottare una suddivisione equa dei ruoli decisionali e quindi le scelte relative alla gestione dello sviluppo del progetto verranno prese congiuntamente. Inoltre, abbiamo deciso che il team si dividerà in ruoli tecnici, come spiegato meglio al [punto 7](#) del Project Plan.

4. Standard, linee guida, procedure

Il progetto verrà diviso in 2 componenti principali che comunicheranno tra loro tramite delle API rest. In particolare avremo il:

Frontend

Si tratta dell'interfaccia grafica che verrà esposta all'utente dell'applicazione. Per svilupparla useremo [React](#) (Libreria JS) e [TailwindCSS](#) (Framework CSS).

Backend

Sarà il componente responsabile della gestione dei dati e della loro distribuzione, utile per esporre le API al frontend e per le comunicazioni con il database tramite [SQLite](#). Abbiamo deciso di svilupparlo con Python e il framework [Django](#).

Database

Per la gestione dei dati useremo un database relazionale, progettato tramite schema ER e modello logico. Concretamente lo realizzeremo con l'applicativo SQLite.

Deploy

Per quanto riguarda la distribuzione del software abbiamo deciso che il progetto finale verrà caricato su una nostra VPS. Questo ci permetterà di avere un maggiore controllo sulla gestione del sistema. Per semplificare il processo di deployment abbiamo inoltre deciso di utilizzare i Docker container.

5. Attività di gestione

Per quanto riguarda l'attività di organizzazione, abbiamo deciso di incontrarci con frequenza settimanale. Durante gli incontri documentiamo il lavoro fatto ed organizziamo le attività che ognuno dovrà svolgere durante la settimana successiva. Ad ogni incontro produciamo un documento che riassume gli argomenti trattati (Decision Making). Ci siamo posti l'obiettivo di terminare il progetto entro 4 settimane lavorative e daremo priorità alla documentazione del software rispetto al suo sviluppo.

6. Rischi

Dato che il progetto nasce da una nostra idea e non da una commissione da parte di un cliente, ci aspettiamo diversi rischi una volta messo in produzione. In particolare:

- Finire in ritardo lo sviluppo, per questo motivo abbiamo deciso di cominciare con largo anticipo rispetto alla consegna del Project Plan;
- Il progetto non verrà utilizzato, ad esempio per il prezzo dell'abbonamento. Per questo motivo abbiamo deciso di rilasciare il software in 2 versioni: una gratis con funzionalità limitate ed una a pagamento senza nessun limite.

7. Personale

Il team, specializzato su aree differenti, si dividerà nei seguenti ruoli:

- Database Architect: [Gherardi Samuel](#)

- Backend Developer: [Soldini Matteo](#)
- Frontend Developer: [Zanotti Paolo](#)
- Tester: [Gherardi Samuel](#)

Il Database Architect inizierà a lavorare prima degli altri, per poter rendere disponibile lo schema delle API su cui si baserà il backend. Seguirà poi il lavoro dei developer per poi finire con il testing del software, documentando man mano il tutto.

8. Metodi e tecniche

Potremmo estrarre i requisiti della nostra applicazione da un'utenza base tramite indagini di mercato e verificarli successivamente con dei diagrammi UML. In particolare usando:

- USE-CASE DIAGRAM
- UML-CLASS DIAGRAM
- ER DIAGRAM
- STATE-CHART DIAGRAM
- ACTIVITY DIAGRAM
- SEQUENCE DIAGRAM

Per il versioning del codice invece utilizzeremo Git tramite la piattaforma Github. Verificheremo poi il tutto con il testing del software.

9. Garanzia di qualità

Abbiamo deciso di seguire il modello di McCall per garantire che il software in fase di sviluppo soddisfi i seguenti requisiti di qualità:

1. Correttezza
2. Affidabilità
3. Integrità
4. Manutenibilità
5. Usabilità
6. Riutilizzabilità Garantiremo i precedenti requisiti sfruttando le tecnologie più moderne, ad esempio React permette il riutilizzo del codice tramite l'uso di componenti. Inoltre adotteremo le best practices del web development moderno, come ad esempio la sicurezza dei dati sensibili dell'utente, l'accessibilità del contenuto e una buona ottimizzazione dei contenuti per il posizionamento del sito sui motori di ricerca principali (SEO).

10. Pacchetti di lavoro

Il software sviluppato, in particolare, seguirà il pattern MVC (Model View Controller). Nel nostro caso il backend gestirà la parte di Model e Controller esponendo delle API Restful, le quali potranno essere consumate dal frontend realizzato con React.

11. Risorse

L'intero progetto verrà realizzato sfruttando l'attrezzatura personale del team, in quanto non siamo un'azienda ma 3 lavoratori autonomi che hanno deciso di collaborare. Nello specifico verranno utilizzati i laptop dei professionisti per lo sviluppo dell'applicativo e gli smartphone per il testing (aggiuntivo) della parte di responsive. Per la messa in produzione, come già accennato nel [punto 4](#), useremo una VPS che ci fornirà uno spazio dove caricare la nostra immagine Docker. Come spazio di ritrovo/lavoro useremo principalmente le aule messe a disposizione dall'Università di Bergamo.

12. Budget

Dato che il progetto nasce da una nostra esigenza e lo sviluppo non richiede grandi risorse economiche, non partiamo da un budget prefissato. Tuttavia, possiamo stimare il costo della realizzazione finale calcolando quanto tempo ci richiederà di lavoro.

Partendo da una tariffa oraria di 30€/ora e 120 ore di sviluppo (40 ore * 3 persone) si arriva ad un totale di 3600€. Per rientrare nei costi entro 1 anno, supponendo un costo dell'abbonamento di 5€/mese, è necessaria l'attivazione di 60 piani a pagamento.

13. Cambiamenti

Gli issue riportati dagli utenti (tramite relativa segnalazione) forniranno una base per la costruzione della roadmap per lo sviluppo di nuove funzioni. Questo approccio è reso più facile dal fatto che seguiremo il modello di McCall per la garanzia di qualità ([vedi punto 9](#)), in particolare la parte di riutilizzabilità e manutenibilità del codice. In questo modo non ci saranno problemi nell'aggiungere modifiche future.

14. Consegna

Seguendo il modello git-flow, ad ogni nuova funzionalità verrà creato un branch su Github. Dopo averla implementata, il nuovo branch verrà mergiato in Development (branch di sviluppo), dove verrà verificata la stabilità dell'applicazione. Una volta terminati i test, Development verrà mergiato in Main (branch principale) e a quel punto verrà rilasciata una nuova release disponibile a tutti gli utenti.

Software Life Cycle

Per il processo di sviluppo del software il team ha deciso di utilizzare un approccio di tipo Agile poichè ci è sembrata la modalità più consona alle nostre modalità di sviluppo. Di seguito indichiamo alcuni aspetti particolarmente importanti:

- Alla base del lavoro del team c'è il creare le condizioni migliori per poter sviluppare al meglio: i lavori vengono assegnati in base alle abilità che ognuno ha. Un ulteriore principio importantissimo è la coesione tra di noi: tutti devono sentirsi liberi di chiedere aiuto in caso di difficoltà ed ognuno deve essere sempre disponibile per offrire il proprio aiuto.
- Nel team non c'è la presenza di un Project Manager: in questo modo c'è una suddivisione equa dei ruoli decisionali che ci permette di ottenere più compattezza ed unità d'intenti.
- Seguendo il manifesto dello sviluppo agile del software, ci interessa di più avere un software funzionante rispetto alla documentazione. Se durante lo sviluppo ci accorgiamo di dover aggiungere dei dettagli che in precedenza non erano stati considerati, li andiamo ad aggiungere in un secondo momento all'interno dei documenti.
- Sfruttiamo la tecnica del pair programming: gli sviluppatori del lato front-end e back-end hanno la possibilità di lavorare in coppia, in questo modo la creazione del sistema risulta più omogeneo e unidirezionale.
- Il team è propenso al cambiamento: l'idea di sviluppo è nata da una nostra esigenza, quindi ogni nuovo bisogno verrà preso in considerazione.
- Ci siamo basati sulla Model Driven Architecture (MDA).

Configuration Management

Tutto il lavoro, sia in termini di codice che di documentazione, viene salvato su un repository GitHub in condivisione con tutti i membri del team.

Struttura

Sono presenti 2 cartelle:

- Cartella `docs`: conterrà i vari file riguardanti la documentazione del progetto, partendo dal project plan a tutti i diagrammi UML.
- Cartella `src/frontend`: conterrà il codice front-end della piattaforma, creato attraverso React e NextJS.
- Cartella `src/backend`: conterrà il codice back-end della piattaforma, creato attraverso Python e il framework Django.

Issue

Le diverse attività che vengono assegnate all'interno del team vengono create sotto forma di `issue` su GitHub, i quali possono trovarsi in diversi stati in base al loro avanzamento. Per tenere traccia dello stato utilizziamo una board Kanban, suddivisa in 3 colonne: **Todo**, **In Progress**, **Done**.

- **Todo**: contiene gli issue che sono stati creati ma non ancora sviluppati.
- **In Progress**: contiene gli issue che si stanno sviluppando.
- **Done**: contiene gli issue che sono stati sviluppati e che sono terminati.

Branch

Il branch principale è il `main`. Su questo ci sarà tutto il codice e tutta la documentazione creata. Se un issue che fa riferimento alla creazione di documenti viene creato ed è nello stato di **Todo**, prima viene portato nello stato di **In Progress** e a quel punto viene sviluppato mantendosi sul branch principale `main`; una volta terminato il suo sviluppo viene creata una **Pull Request** che deve essere accettata dai membri del team: una volta revisionata l'issue passa nello stato di **Done** e termina. Discorso a parte per quanto riguarda gli issue che fanno riferimento allo sviluppo di codice, sia backend che frontend: questi issue vengono creati e sviluppati un branch `develop`, a cui verrà fatto un **merge** con il branch `main` una volta completati tutti gli issue relativi al codice. Anche per questa seconda tipologia di issue è presente il discorso della **Pull Request**.

People Management

Il team ha scelto di seguire una struttura organizzativa di tipo **adhocratica**, perciò alla base del lavoro c'è l'adattamento reciproco. Non abbiamo suddiviso i task in modo fisso, ma di volta in volta li abbiamo assegnati seguendo le possibilità di ciascuno. Il più grande vantaggio nell'utilizzo della adhocrazia è quella di avere un'organizzazione molto flessibile e adattabile.

Il gruppo di lavoro è inoltre organizzato secondo il modello di sviluppo **SWAT** (skilled worker with advanced tools): vi sono 3 persone che sviluppano comunicando tramite canali informali. Le riunioni, che vengono fissate per discutere della situazione progettuale, vengono effettuate anch'esse in modo informale.

Dal nostro punto di vista i vantaggi principali dell'organizzazione tramite adhocrazia e SWAT sono:

- Tranquillità e flessibilità nello sviluppo da parte dei partecipanti.
- Suddivisione dei task principalmente effettuata sulla base del piacere, componente per noi fondamentale per la creazione di un buon prodotto.

Queste sono le principali mansioni svolte dai partecipanti del team:

	Progettazione Database	Back-end	Front-end	Testing
Matteo	V	V		
Paolo	V		V	
Samuel	V			V

Software Quality

Il team ha scelto di seguire i parametri di qualità definiti da **MCall**, abbiamo assegnati seguendo le possibilità di ciascuno. Di seguito la descrizione dettagliata.

Parametri riguardanti l'operatività del software

- **Correttezza:** il software rispetta tutte le specifiche che ci siamo preposti di seguire per le nostre esigenze.
- **Affidabilità:** il software è sicuramente affidabile in quanto revisionato più volte dai membri del team e poi sottoposto ad una fase di testing.

- **Efficienza:** essendo un'applicazione web, per poterla utilizzare occorre una connessione ad internet ed un browser attraverso il quale inviare le richieste al server.

- **Integrità:** il nostro software è sicuro:

1) La gestione delle password degli utenti che si registrano nel sistema viene effettuato attraverso SHA-256 e token JWT. Grazie a SHA-256 le password vengono crittografate, quindi non vengono rese visibili a chi ha accesso al database. JWT, acronimo di JSON Web Token, è un sistema di cifratura e di contatto in formato JSON per lo scambio di informazioni tra i vari servizi di un server. Si genera così un token che può essere cifrato e firmato tramite una chiave disponibile solo a colui che lo ha effettivamente generato.

2) Il protocollo HTTPS (HyperText Transfer Protocol over Secure Socket Layer), consiste nella comunicazione tramite il protocollo HTTP (Hypertext Transfer Protocol) all'interno di una connessione criptata, tramite crittografia asimmetrica, dal Transport Layer Security (TLS).

- **Usabilità:** il prodotto è assolutamente semplice da utilizzare, in quanto non servono particolari abilità nel poterlo comprendere.

Parametri riguardanti la transizione verso un nuovo ambiente

- **Portabilità:** la ricerca di errori è facilitata dalla separazione che abbiamo effettuato tra la gestione dei dati, back-end, e la rappresentazione dei dati, il front-end.

- **Riusabilità:** il template dell'applicazione sarà sicuramente riutilizzabile in futuro, in quanto si tratta di una base grafica.

- **Interoperabilità:** sicuramente il nostro prodotto software potrà essere integrato ad altri servizi, in quanto è presente una netta separazione tra il lato front-end e back-end.

Requirement Engineering

Una delle parti più importanti è quella di definire i requisiti funzionali (funzionalità del sistema) e non funzionali (vincoli imposti al sistema come ad esempio la scalabilità).

L'ingegneria dei requisiti è formata da 4 fasi, l'elicitazione dove si definiscono i requisiti del problema, poi la specifica in cui si formalizzano i requisiti raccolti nell'elicitazione, dopodiché è presente la validazione all'interno della quale si verifica la correttezza della specifica dei requisiti creata ed infine ci sarebbe la fase della trattativa, che serve per trovare un punto d'incontro con il cliente, ma nel nostro caso, dato che il cliente non è presente, viene saltata.

Abbiamo deciso di suddividere i diversi requisiti secondo il modello MoSCoW.

Funzionali

Must Have

- Accesso degli utenti tramite e-mail e password.
- L'utente per poter utilizzare l'applicazione deve essere obbligatoriamente autenticato.
- Per ogni utente occorre memorizzare lo username, l'e-mail e la password.
- Un utente può registrare delle transazioni in ingresso o in uscita solamente se ha registrato un conto.

Should Have

- Ogni utente può visualizzare l'andamento delle sue spese/guadagni sulla base di grafici chiari ed esplicativi.
- Ogni utente può decidere di cambiare un budget creato in precedenza sulla base di ciò che ha realmente ottenuto.

Won't

- Recupero della password dimenticata da un utente.
- Doppia autenticazione da parte degli utenti.

Non Funzionali

Must Have

- La password degli utenti deve essere memorizzata tramite cifratura hash a 256 bit (SHA-256);
- Il sistema viene progettato per poter utilizzato sia da personal computer che da smartphone.

Should Have

- Il tempo di risposta del sistema all'autenticazione degli utenti deve essere al massimo di 2 secondi.
- Il tempo di risposta del sistema alla creazione di un nuovo account deve essere al massimo di 2 secondi.

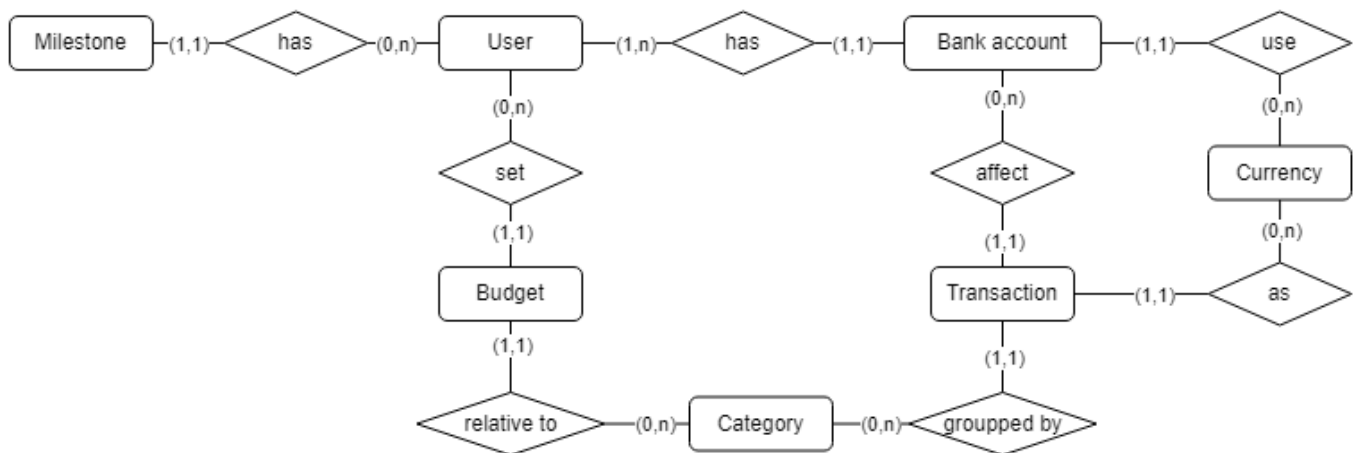
Won't

- Durante il recupero della password il sistema deve inviare la mail di conferma al massimo dopo 3 secondi.

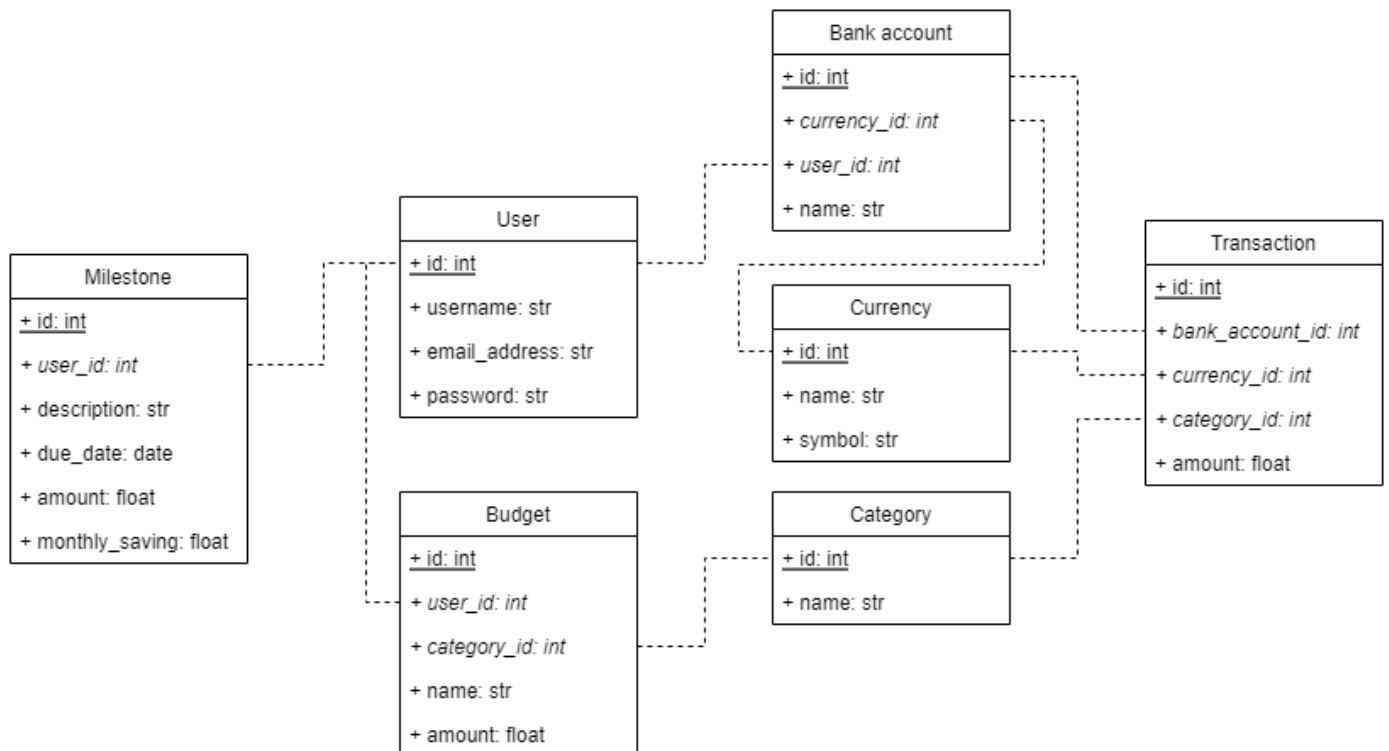
Model

Modello ER

Schema ER



Modello logico



Descrizione

User

Contiene le informazioni relative all'utente. In modo particolare verrà eseguito l'hashing della password prima di essere salvata. L'utente potrà impostare la valuta che desidera visualizzare.

Bank account

Contiene le informazioni relative ai conti dell'utente.

Transaction

Contiene le informazioni relative ai movimenti (entrate ed uscite) di un conto e divisi in categorie.

Category

Contiene le informazioni relative alle categorie che raggruppano transazioni e budget (es. in Cibo, Auto, ecc).

Budget

Contiene le informazioni utili per gestire la creazione di budget di spesa mensile per una specifica categoria.

Currency

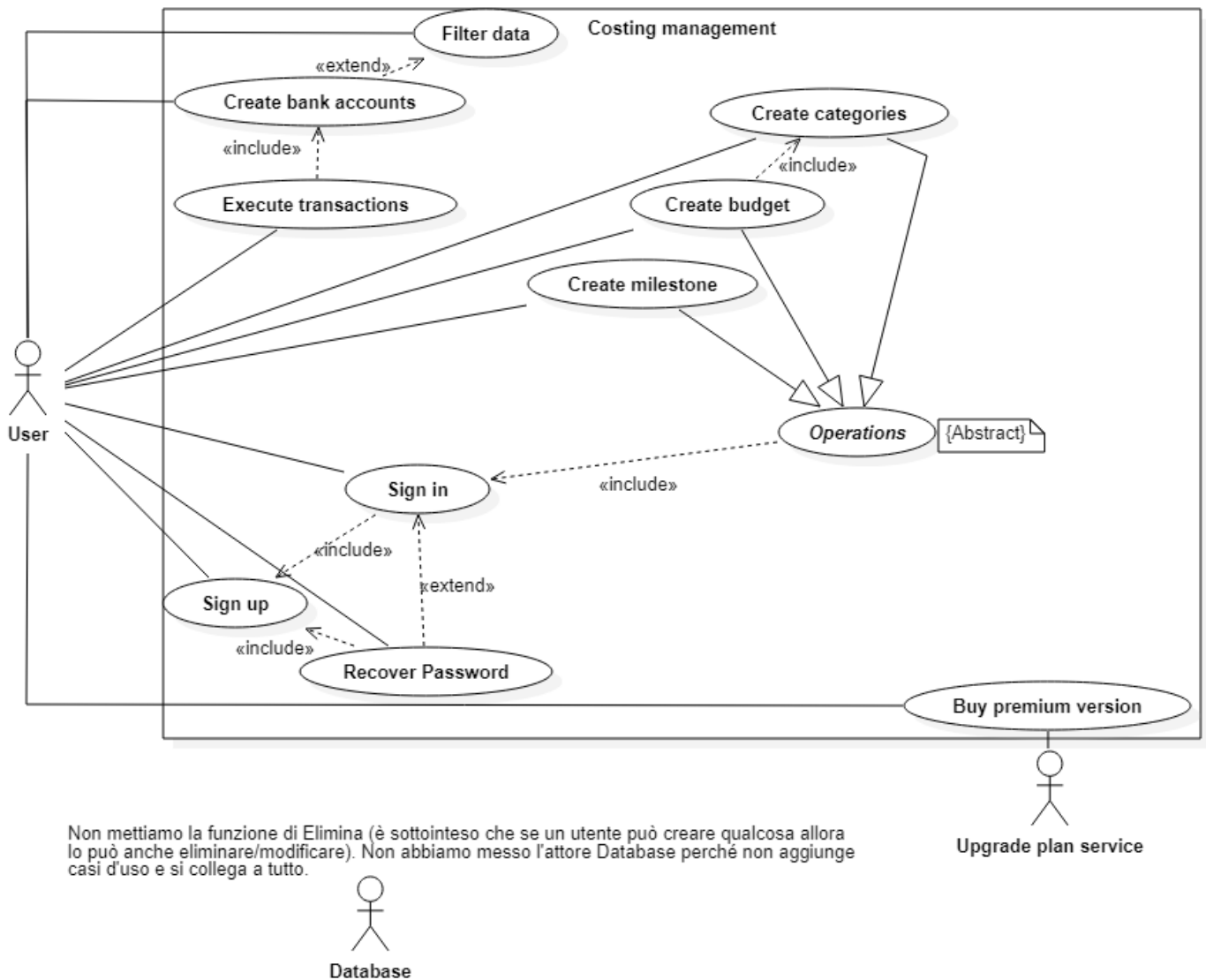
Contiene le informazioni relative alle valute disponibili. Viene usata dalle transazioni e viene scelta dall'utente.

Milestone

Contiene le informazioni relative agli obiettivi che l'utente si pone di raggiungere entro un certo lasso di tempo (es. vacanza Budapest). Questo viene effettuato mettendo da parte una percentuale delle proprie entrate mensili (`monthly_saving`).

Diagramma dei casi d'uso

Diagramma



Descrizione

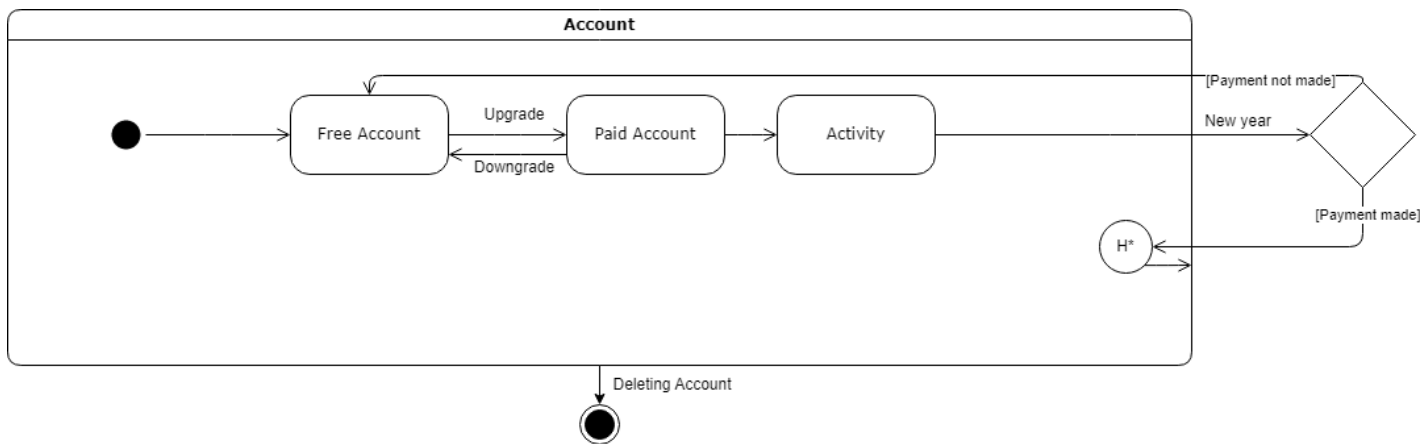
L'attore principale è lo **User**, il quale all'inizio si registra (Sign up), può successivamente accedere al sistema (Sign In) e recuperare la password (Recover Password). Una volta fatto tutto ciò può aggiungere i propri conti (Create bank accounts) e solo dopo può segnare i suoi movimenti (Execute transactions) e filtrare i dati per avere una visione corretta delle sue spese (Filter data). Si possono riassumere queste attività con un caso d'uso astratto (Transactions), che può quindi essere eseguito solo dopo che l'utente ha fatto l'accesso al sistema.

Anche senza aggiungere i propri conti, l'utente ha la possibilità di creare le categorie dei suoi movimenti (`Create categories`) e di crearsi un obiettivo da raggiungere nel futuro (`Create milestone`). Al contrario, per crearsi un budget (`Create budget`) deve prima aver inserito delle categorie. Tutte queste operazioni vengono generalizzate dal caso d'uso astratto (`Operations`), il quale può essere eseguito dopo il login-in dell'utente. Se l'utente vuole cambiare piano di abbonamento e passare da quello gratuito a quello a pagamento può comprarlo (`Buy premium version`) tramite l'attore **Upgrade plan service**.

Nota !

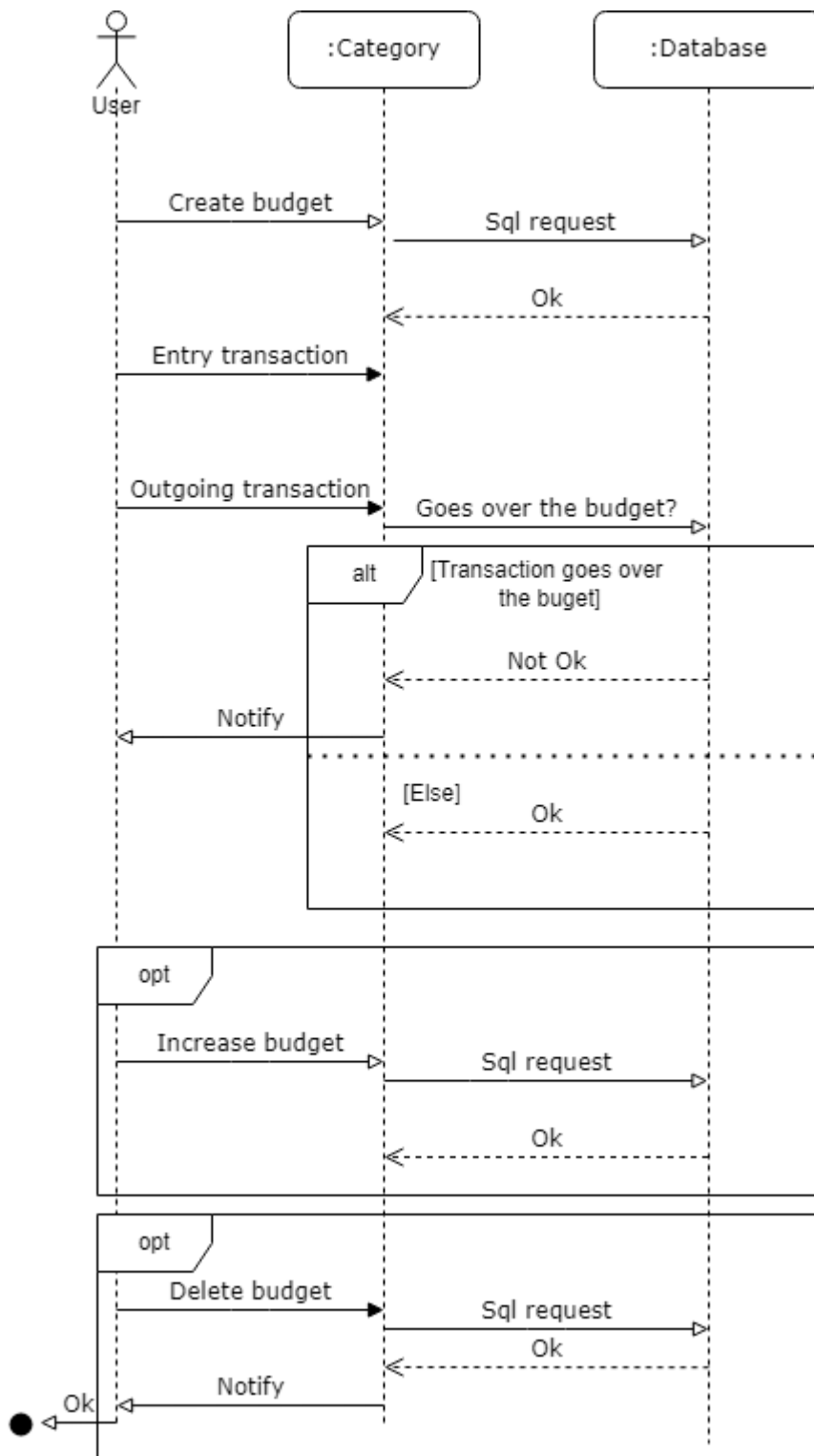
- Abbiamo deciso di non inserire l'attore **Database** poiché non aggiunge nuovi casi d'uso e si collega a tutto, rendendo il diagramma difficile da leggere.
- Non abbiamo messo il caso d'uso `Elimina` perché già sottointeso che se un utente può aggiungere i propri conti e i suoi movimenti, allora può anche modificarli e/o eliminarli.

Diagramma di macchine a stato



Abbiamo deciso di rappresentare più nel dettaglio le azioni di creazione dell'account gratuito e l'eventuale passaggio all'account a pagamento da parte di un utente attraverso un diagramma di macchine a stato. In particolare l'utente una volta creato l'account gratuito (Free Account), può decidere di effettuare un upgrade all'account a pagamento (Paid Account) sbloccando nuove funzionalità. Da questo punto si ha la possibilità di effettuare un downgrade e tornare all'account gratuito, altrimenti ogni anno occorre effettuare un pagamento per mantenere i privilegi: se ciò avviene allora l'utente può continuare ad utilizzare il servizio (grazie al nodo storia), altrimenti ritorna ad avere un profilo base.

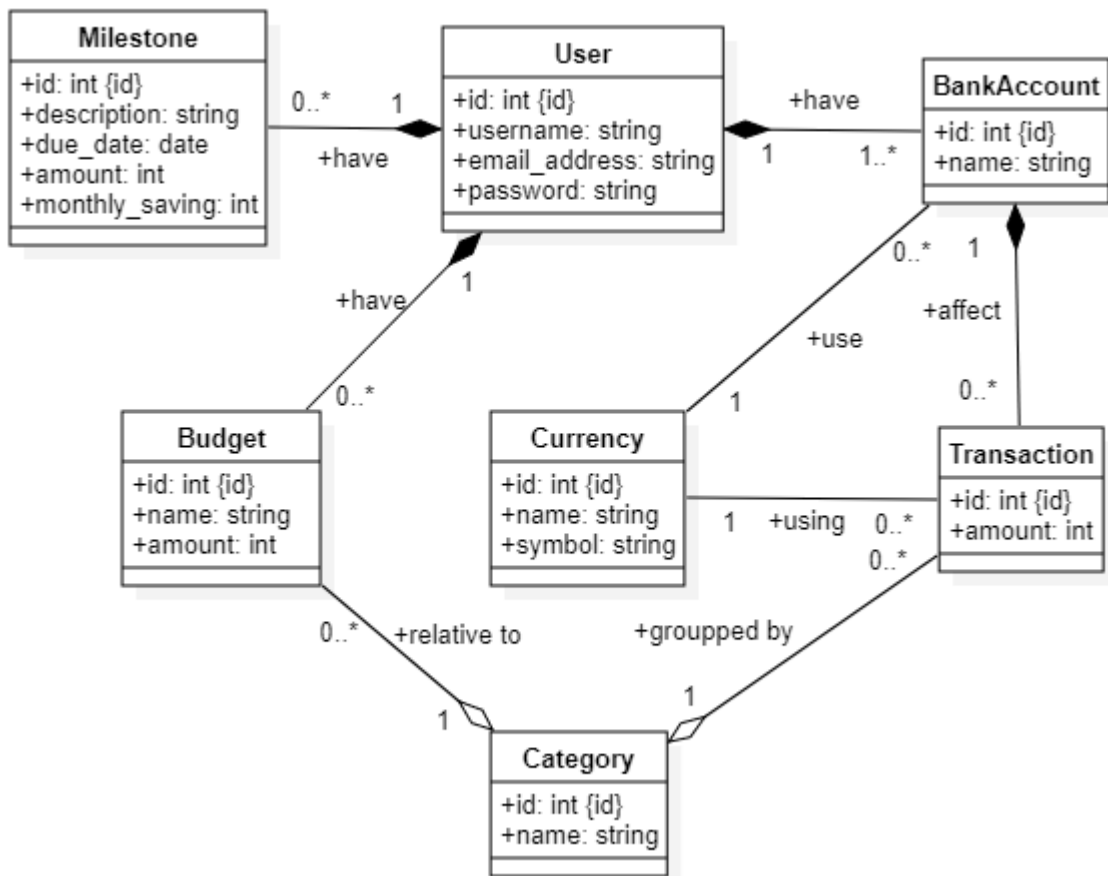
Diagramma di sequenza



Abbiamo deciso di rappresentare più nel dettaglio il meccanismo di creazione del budget da parte di un utente(User) in una determinata categoria, e la relativa possibilità di non rispettarlo, attraverso un diagramma di sequenza. In particolare l'utente, una volta selezionata una categoria (:Category), può creare un budget (Create budget), e a questo punto tale budget viene creato anche a livello di database del sistema (Sql

request,:Database). Successivamente l'utente può effettuare transizioni in entrata (Entry transaction) oppure in uscita (Outgoing transaction): in quest'ultimo caso il sistema richiede al database se tale spesa sfora il budget (Goes over the budget?) e se così fosse allora viene inviata una notifica all'utente (Notify). Questa condizione viene gestita tramite il frame *alt*, il quale gestisce l'invio di messaggi alternativi. Se la notifica di sforamento di budget viene inviata, a quel punto l'utente può decidere se aumentarlo (Increase budget) oppure eliminarlo (Delete budget). Queste azioni opzionali vengono modellate dal tipo di frame *opt*.

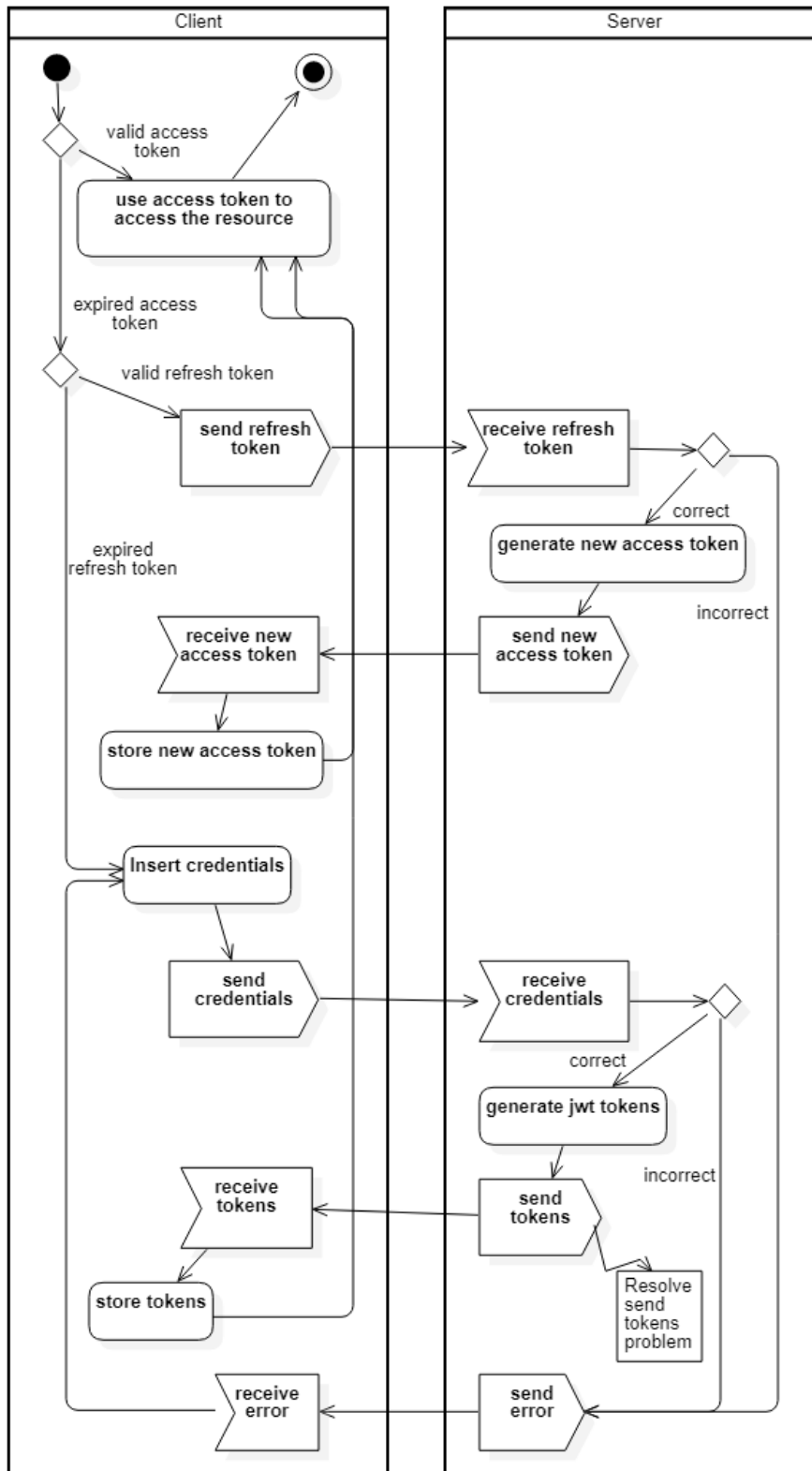
Diagramma delle classi



In questo diagramma andiamo a specificare in modo più approfondito le entità che interagiscono nel nostro sistema. In particolare abbiamo utilizzato le stesse classi definite nel [diagramma er](#) specificandone gli attributi. Inoltre abbiamo arricchito il diagramma con cardinalità ed aggregazioni: ad esempio, se viene eliminato User, di conseguenza BankAccount, Transaction, Milestone e Budget non sono più rilevanti, per questo motivo vengono eliminati; Invece Budget e Transaction hanno un legame debole con Category, ciò significa che se Category viene eliminato ciò non porta all'eliminazione di Budget e Transaction.

Diagramma delle attività

Accesso risorse tramite token JWT



In questo diagramma viene descritto il meccanismo di accesso alle risorse tramite token JWT. Ad ogni richiesta, il client deve includere nel messaggio il token access. Questo per ottenere il permesso da parte del server di accedere ad una determinata risorsa.

Inizialmente l'utente deve fornire le proprie credenziali che vengono successivamente inviate ed autenticate dal server. Se l'utente è presente nel database e la password è corretta allora vengono ritornati al client 2 token: access per autenticare le richieste alle api e refresh per rinnovare il token access. Allo scadere della validità del token access, il client invia una richiesta al server che contiene il token refresh. Quest'ultimo risponde con il nuovo token access. Se anche il token refresh è scaduto allora l'utente deve riautenticarsi.

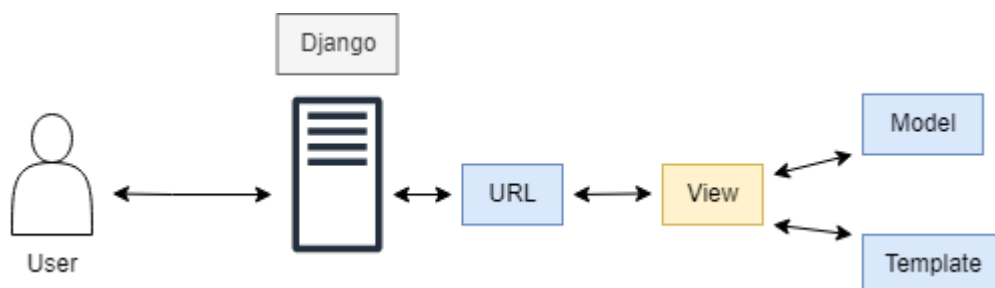
Software Architecture

L'architettura del nostro software è basata sul pattern **MVC** (Model View Controller). Perciò abbiamo il front-end che ha il ruolo di view in quanto mostra i risultati voluti dall'utente; il back-end invece ricopre sia del ruolo di model, infatti fornisce i metodi per accedere ai dati utili dell'applicazione, che quello di controller, in quanto riceve i comandi dell'utente (attraverso la view) e li attua modificando lo stato degli altri 2 componenti.

Di seguito dettagliamo in modo più specifico le caratteristiche dell'architettura che segue il framework che utilizziamo per lo sviluppo back-end Django.

Model

Il Model è responsabile della gestione di tutte le attività relative ai dati. Può essere una tabella in un database, un file JSON o qualsiasi altra cosa. Inoltre il Model prende i dati da dove archiviati e quindi li elabora prima di inviarli alla View. Il Model è un componente dell'architettura di Django che contiene la Business Logic. Il flusso di informazioni è il seguente: quando ti iscrivi a qualsiasi sito web, si effettua un clic su un pulsante di registrazione. Quando si fa clic sul pulsante di registrazione, viene inviata una richiesta al Controller. Quindi il Controller chiama il Model e gli chiede di applicare la logica sul modulo che viene ricevuto come parametro. Il Model quindi applica la sua logica e fornisce una risposta al Controller. Il Controller passa quindi la risposta a te, che sei il client.



View

Il componente View nell'architettura Django viene utilizzato per visualizzare i dati dal componente Model. Può anche essere utilizzato per raccogliere dati dall'utente e inviarli al Model come input del modulo. In questo modo, il componente View contiene la logica dell'interfaccia utente. Ad esempio, se si fa clic su una spesa e poi vai alla pagina dei dettagli, la nuova pagina web che viene generata è la visualizzazione dei dettagli della spesa. Allo stesso modo, se si fa clic su una categoria e poi si passa alla visualizzazione per categoria, la nuova pagina Web generata è la visualizzazione per categoria.

Controller

Poichè il controller decide quale vista visualizzare, ha il potere di manipolare il modello della vista. In questo modo, può applicare qualsiasi logica e regola al modello della vista. Il controller determina anche come visualizzare la vista e come rispondere all'input dell'utente. Possiamo dire che il controller è colui che decide quando e cosa deve essere

visualizzato. I controller sono molto utili per mantenere il nostro codice DRY (Don't Repeat Yourself), gestibile e scalabile.

Benefici dell'architettura di Django

Il framework Django utilizza questa architettura e non richiede codice complesso per far comunicare tutti e tre questi componenti. Ecco perché Django sta diventando popolare.

I seguenti sono alcuni dei vantaggi dell'utilizzo di questa architettura in Django:

- **Sviluppo rapido:** più sviluppatori possono lavorare contemporaneamente su diversi aspetti della stessa applicazione.
- **Debolmente accoppiato:** ogni parte dell'architettura di Django deve essere presente per mantenere un alto livello di sicurezza del sito web.
- **Facilità di modifica:** il vantaggio di utilizzare Django è che ci offre molta più flessibilità nella progettazione del nostro sito Web rispetto ad altri framework.

Software Design 📌

Per quanto riguarda il software design il team si è preposto di scrivere codice basandosi sui principi di progettazione di **modularità** e **complessità**.

In questo modo ciò che si ottiene è un sistema che ha:

- **Alta coesione**, ossia avere una alta correlazione tra le funzionalità che sono definite all'interno di uno stesso modulo.
- **Basso accoppiamento**: ossia avere una forte indipendenza dei moduli tra di loro.

Design Pattern

Durante lo sviluppo del codice il team ha deciso di utilizzare i design pattern Observer e Adapter. (serializer)

Per quanto riguarda l'Observer pattern, in particolare lato front-end, lo abbiamo applicato attraverso lo strumento React degli *Hooks*.

Il pattern Observer permette di definire una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato interno, ciascuno degli oggetti dipendenti da esso viene notificato e aggiornato automaticamente. L'Observer nasce dall'esigenza di mantenere un alto livello di consistenza fra classi correlate, senza produrre situazioni di forte dipendenza e di accoppiamento elevato.

Ma cosa è un Hook? Gli Hooks sono funzioni che ti permettono di "ancorarti" all'interno delle funzioni di React state e lifecycle da componenti funzione. Gli Hooks ti permettono di usare React senza classi.

Invece riguardo l'Adapter pattern, utilizzato lato back-end, lo abbiamo realizzato grazie allo strumento *serializer* di Django.

L'Adapter è un Design Pattern di tipo strutturale che è utilizzato quando si ha la necessità di rendere compatibili due interfacce che di fatto non lo sono, senza modificarne il codice.

I Serializer consentono di convertire dati complessi come set di query e istanze di modelli in tipi di dati Python nativi che possono quindi essere facilmente resi in JSON , XML o altri tipi di contenuto. Inoltre i Serializer forniscono anche la deserializzazione, consentendo la riconversione dei dati analizzati in tipi complessi, dopo aver prima convalidato i dati in ingresso.

Testing

Durante lo sviluppo del backend sono stati implementati dei test di unità tramite il modulo `TestCase` incluso in Django. In modo particolare abbiamo analizzato la classe `Transactions` e abbiamo individuato i seguenti test da effettuare dal punto di vista delle api:

Descrizione del test	Risultato aspettato
Visualizzare tutte le transazioni	Successo
Visualizzare una transazione che non appartiene all'utente	Fallito
Creare una transazione	Successo
Creare una transazione con un campo mancante	Fallito
Creare una transazione con una valuta inesistente	Fallito
Creare una transazione con account bancario che non appartiene all'utente	Fallito
Creare una transazione con una categoria che non appartiene all'utente	Fallito
Aggiornare un campo di una transazione	Successo
Aggiornare un campo inesistente di una transazione	Fallito
Aggiornare un campo di una transazione inesistente	Fallito
Eliminare una transazione	Successo
Eliminare una transazione di un altro utente	Fallito

I test di Django permettono di eseguire i test senza modificare il database principale. Quando si avvia un test, in automatico viene creato un database temporaneo che verrà successivamente eliminato. Prima di poter fare i test sopra citati è necessario creare dei dati fittizi. In modo particolare vengono creati 2 utenti: uno principale ed uno per testare i permessi.

In ogni app (es. transactions) di Django è presente un file `test.py` dove si possono sviluppare i test di unità.

L'esecuzione dei test può essere effettuata manualmente tramite il comando `python manage.py test`. In questo caso abbiamo implementato una Github Action che permette, ad ogni nuova push request nel repository, di eseguire i test in un ambiente controllato ed in caso di un test fallito notificare i contributori.

Software Maintenance

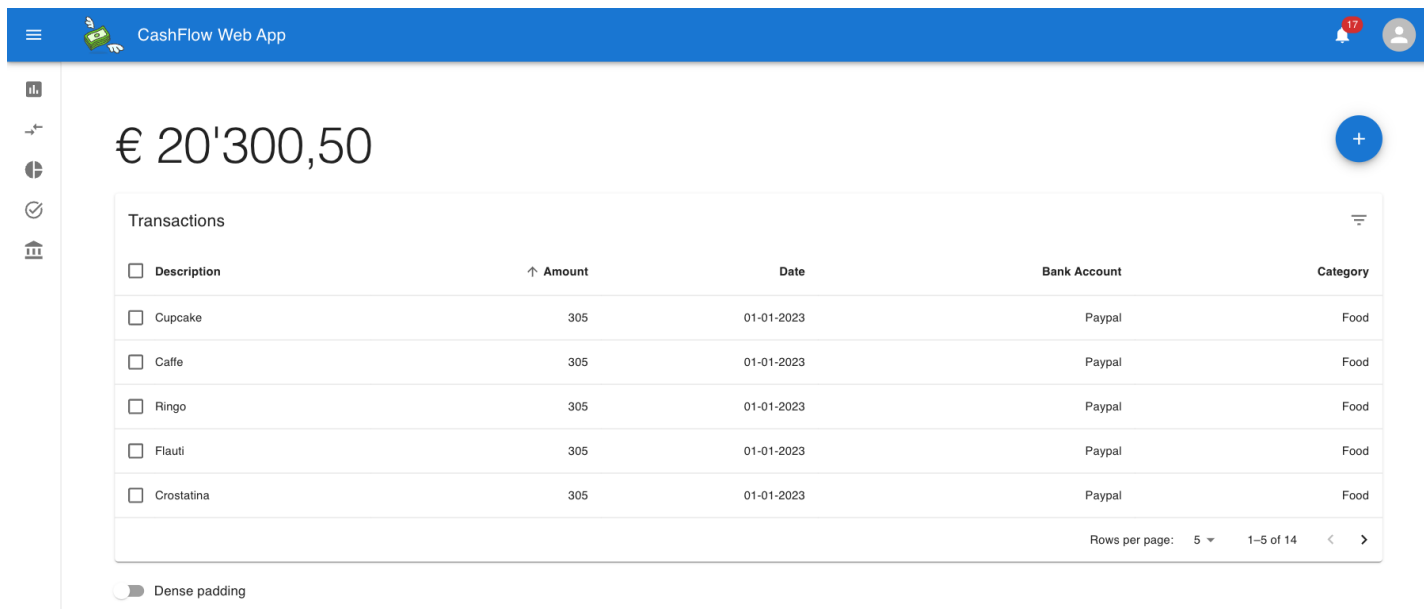
All'interno del progetto abbiamo eseguito due tipi di manutenzione:

- *Perfettiva* e *Refactoring* per migliorare l'interfaccia utente e risolvere quindi alcuni Bad Smells
- *Preventiva* per aggiornare la documentazione e i commenti

Non è stata documentata la *manutenzione correttiva* perché il progetto era ancora in fase di sviluppo e quindi non aveva guasti da sistemare. Non è stata aggiunta nemmeno la manutenzione adattiva in quanto il software si occupa di default delle interazioni con l'hardware e con il sistema operativo tramite delle dipendenze interne, andando ad astrarre direttamente il tutto.

Perfettiva e Refactoring

Come intervento di manutenzione perfettiva e, nel nostro caso, di refactoring del codice è stata migliorata l'interfaccia grafica dopo uno studio condotto sui beta tester. Dal test infatti è emerso che secondo gli utenti le informazioni non erano facilmente raggiungibili all'interno della prima versione della web app. Abbiamo quindi eseguito un refactoring dell'interfaccia grafica in modo tale da avere una dashboard più intuitiva.



CashFlow Web App

€ 20'300,50

Transactions				
<input type="checkbox"/> Description	↑ Amount	Date	Bank Account	Category
<input type="checkbox"/> Cupcake	305	01-01-2023	Paypal	Food
<input type="checkbox"/> Caffè	305	01-01-2023	Paypal	Food
<input type="checkbox"/> Ringo	305	01-01-2023	Paypal	Food
<input type="checkbox"/> Flauti	305	01-01-2023	Paypal	Food
<input type="checkbox"/> Crostatina	305	01-01-2023	Paypal	Food

Rows per page: 5 1-5 of 14

☐ Dense padding

L'interfaccia grafica dopo il refactoring del codice

Preventiva

Per quanto riguarda la manutenzione preventiva abbiamo aggiornato la documentazione ogni volta che veniva introdotta una nuova funzionalità all'interno del sistema. Un esempio possono essere l'aggiunta dei file README.md del frontend e del backend per spiegare come eseguire i progetti e far partire il server.

Inoltre ogni procedura importante aggiunta nel codice è stata commentata per spiegare il suo funzionamento e aumentare quindi la manutenibilità del progetto per possibili interventi futuri.

```
51 # Routers provide an easy way of automatically determining the URL conf.
52 router = routers.DefaultRouter()
53 router.register(r'users', UserViewSet)
54 router.register(r'bank_accounts', BankAccountsViewSet)
55 router.register(r'transactions', TransactionViewSet)
56 router.register(r'categories', CategoriesViewSet)
57 router.register(r'milestones', MilestonesViewSet)
58 router.register(r'budgets', BudgetsViewSet)
59
60 urlpatterns = [
61     # urls
62     path('', include(router.urls)),
63     # admin
64     path('admin/', admin.site.urls),
65     # swagger
66     re_path(r'^swagger(?P<format>\.json|\.yaml)$',
67             schema_view.without_ui(cache_timeout=0), name='schema-json'),
68     re_path(r'^swagger/$', schema_view.with_ui('swagger',
69             cache_timeout=0), name='schema-swagger-ui'),
70     re_path(r'^redoc/$', schema_view.with_ui('redoc',
71             cache_timeout=0), name='schema-redoc'),
72     path('api-auth/', include('rest_framework.urls')),
73     # auth
74     path('register/', RegisterView.as_view(), name='auth_register'),
75     path('token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
76     path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
77 ]
```

Commenti relativi all'associazione dei router con le relative viste