



Money Expense

Documento di sviluppo webapp

Scopo del documento

Il seguente documento è il sunto dello sviluppo di una consistente versione della web app Money Expense, che negli scorsi documenti abbiamo esplorato.

Il seguente documento è diviso in varie sezioni che ora brevemente spiegheremo.

Inizieremo con l'esposizione e l'esplorazione dello user flow, un diagramma che mostra tutte le azioni che un utente può fare all'interno della piattaforma, le relative reazioni che l'applicazione avrà, includendo quindi le pagine che ci farà visualizzare, ed eventuali chiamate a front-end e back-end.

Passeremo poi a descrivere nel dettaglio tutte le API realizzate ed implementate nel nostro codice, mediante il diagramma delle risorse e di estrazione delle risorse (API diagram), i quali sono stati estrapolati dal diagramma delle classi proveniente dallo scorso documento (D3-Gog).

Sempre a proposito di API, mostreremo com'è stato possibile documentare questi vari metodi da noi creati mediante l'utilizzo di Swagger.

Per quanto concerne l'implementazione effettiva dell'applicazione, daremo uno sguardo alla struttura del codice che la nostra applicazione utilizza, ponendo anche un focus sulle varie dipendenze installate. Verranno poi discussi i vari modelli istanziati e creati.

Forniamo poi un chiaro resoconto delle pagine create, per poi passare ad una descrizione del GitHub in cui tutto ciò è stato salvato. Passeremo poi alle istruzioni per effettuare il deployment del codice, per poter far funzionare localmente tale applicazione anche su altri dispositivi.

Infine, ci sarà un resoconto di tutti i test creati per verificare ed assicurare la robustezza e la tenuta dell'applicazione.

User flow

<https://app.flowmapp.com/share/projects/e3ode767-a3bc-4df8-89da-ed731317f870/userflow/517ceb58-65bf-46d3-8eb3-33ea3ed20359>

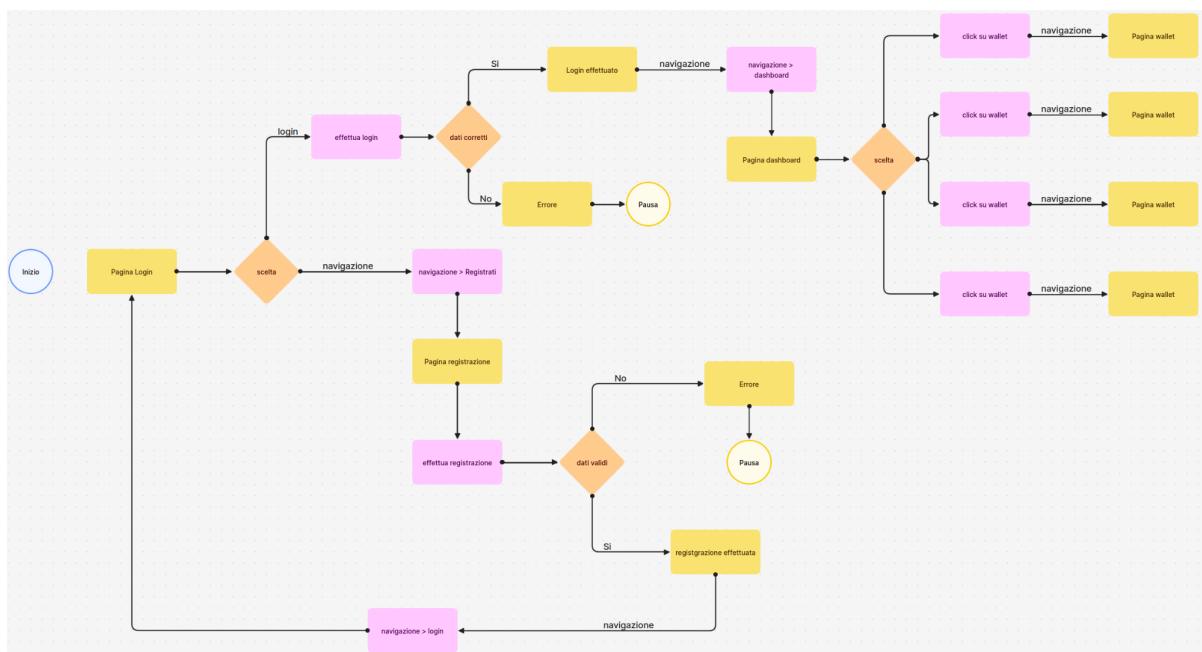
Iniziamo questo documento con uno user flow, nel quale viene mostrato tutte le azioni che un utente puo' svolgere all'interno della versione della piattaforma sviluppata, di cui i dettagli si trovano in questo documento. Questo mostra come funzionano le interazioni applicazione-utente.

Per chiarire ulteriormente lo schema che segue, riportiamo una breve legenda dei componenti che verranno inseriti in questo user flow diagram.

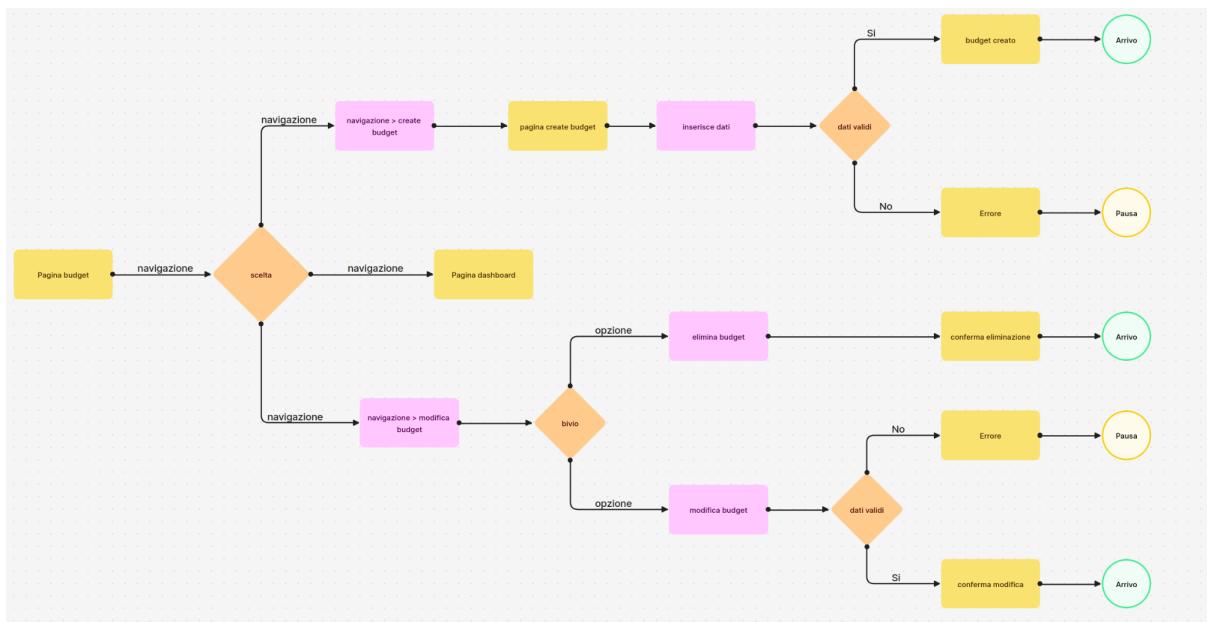


Di seguito i vari pezzi del diagramma.

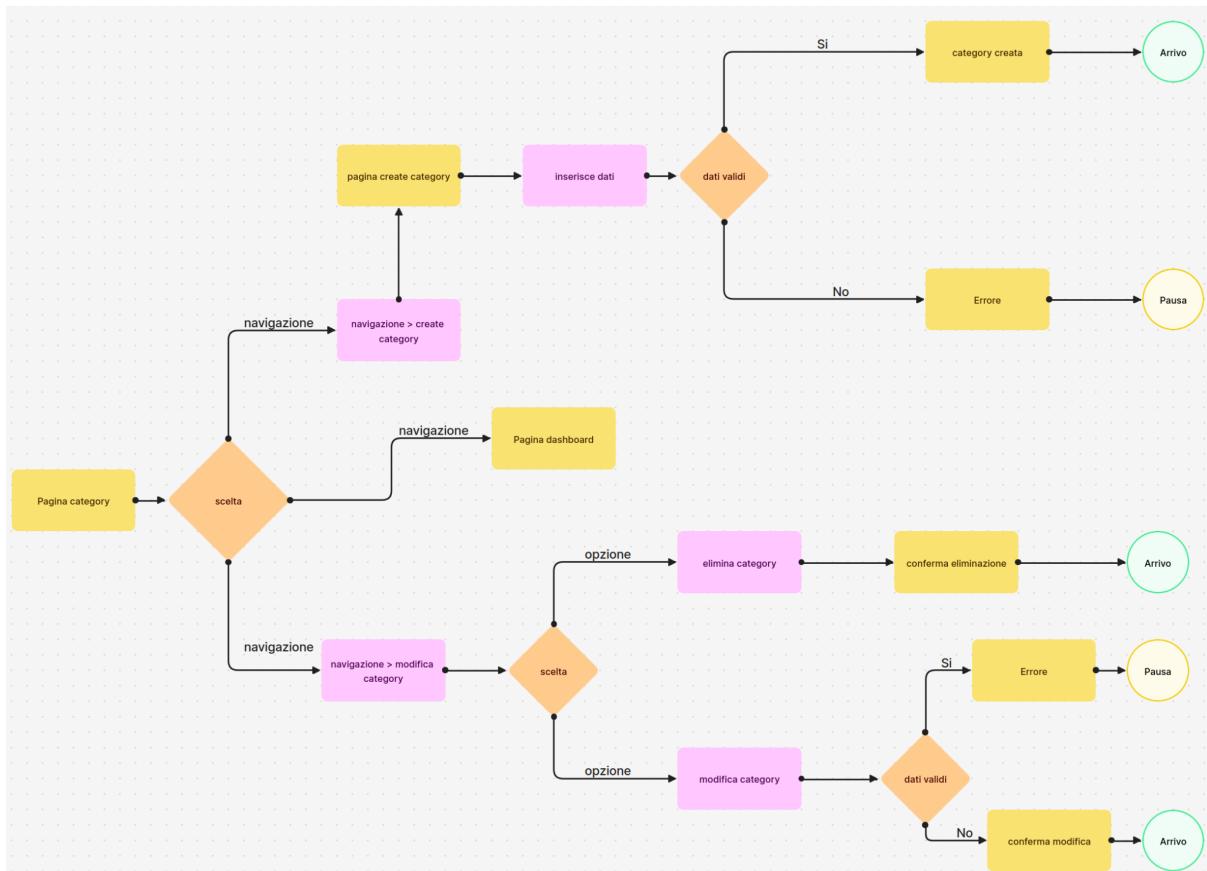
Flow diagram #1: Login e dashboard



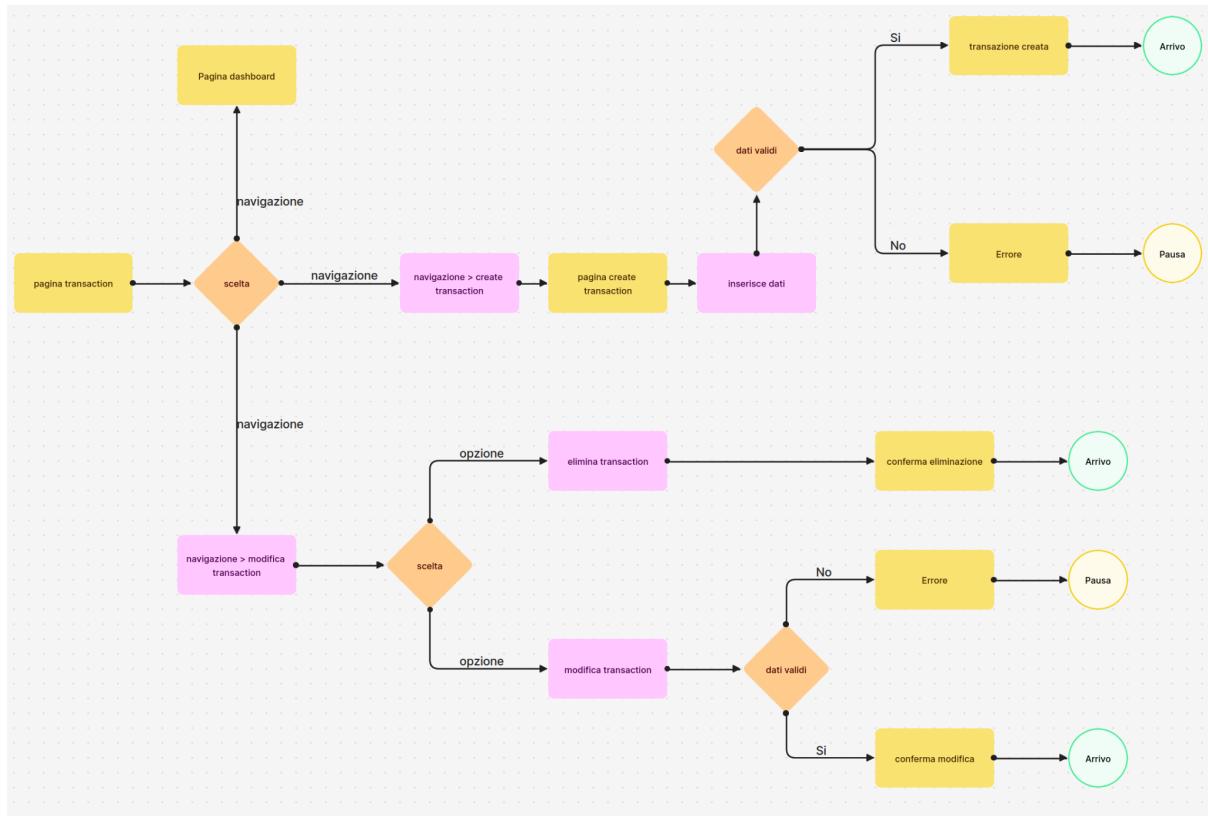
Flow diagram #2: Budget



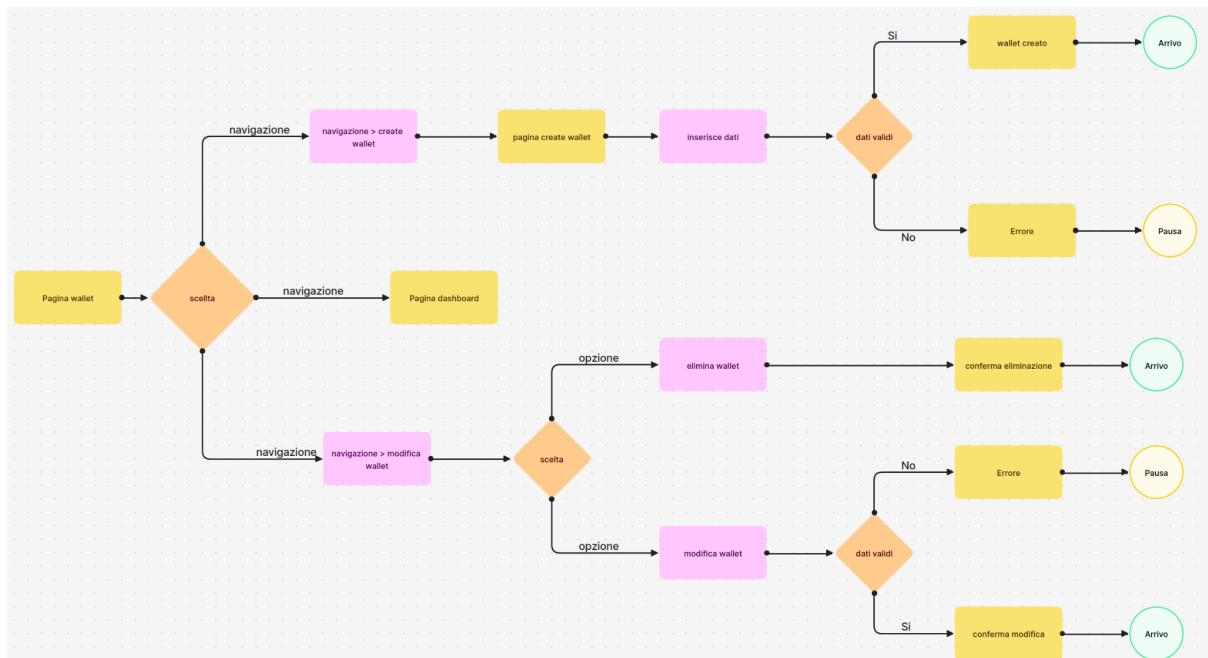
Flow diagram #3: Category



Flow diagram #4: Transaction



Flow diagram #5: Wallet



API dell'applicazione

Questa sezione del documento si occupa di descrivere e mostrare le API che sono state sviluppate a partire da un sottoinsieme di quelle presenti nel documento D3-Gog, dove abbiamo creato il corrispondente diagramma delle classi. In questa sezione troveremo due diagrammi. Uno deputato all'estrazione delle risorse partendo dal class diagram, e uno apposito per rappresentare ciò che abbiamo sviluppato, ovvero le API che verranno utilizzate nel sistema.

Estrazione risorse dal class diagram

Questo diagramma, come anticipato si occupa di mostrare il processo di estazione delle risorse a partire dal class diagram del D3, per portarlo poi nella nostra applicazione attuale.

Il processo è iniziato con l'estrazione delle varie risorse, dalla quale abbiamo estratto i vari modelli che abbiamo realizzato. Una visione più completa viene riportata nella parte relativa ai **Modelli del Database**. Abbiamo individuato successivamente i tipi di dato che applicavano meglio al nostro caso specifico, e abbiamo visto quali erano gli attributi cruciali per lo sviluppo della versione della nostra applicazione. Da qui, li abbiamo dunque portati su MongoDB a partire dal Class diagram.

Per questa specifica implementazione abbiamo deciso di omettere la parte relativa alla chat con l'intelligenza artificiale, in quanto questo avrebbe reso molto più complesso lo sviluppo dell'applicazione, e inoltre chiamare le API portava un prezzo extra in termini monetari. Abbiamo inoltre messo da parte la possibilità di esportare i dati mediante file CSV, in quanto questo richiederebbe query molto dispendiose e uno script di parsing avanzato. Abbiamo deciso quindi di porre il focus sugli altri modelli.

Per ogni risorsa e interazione, viene specificato il metodo HTTP per interagire con essi (siano mediante una GET, POST, PUT o DELETE, in base a ciò che desideriamo ottenere alla fine) e gli eventuali parametri necessari, siano essi presi dal link URL in caso di GET e DELETE, oppure mediante il body, in caso di POST, PUT. Si noti che alcuni metodi utilizzano dati anche nel URL anche se sono richieste di tipo POST e PUT.

Ogni una di queste chiamate ha un effetto, che può essere su backend o sul frontend. In genere, le richieste di tipo PUT, POST, DELETE hanno effetto sul backend, in quanto non ritornano qualcosa che va direttamente a schermo. Viceversa, le GET ritornano dei dati, in alcuni casi in grandi quantità, che vengono poi redirizzati e mostrati dal frontend.

Di seguito portiamo dunque il nostro **diagramma delle risorse**, contenente tutte le informazioni relative al nostro sistema attuale.

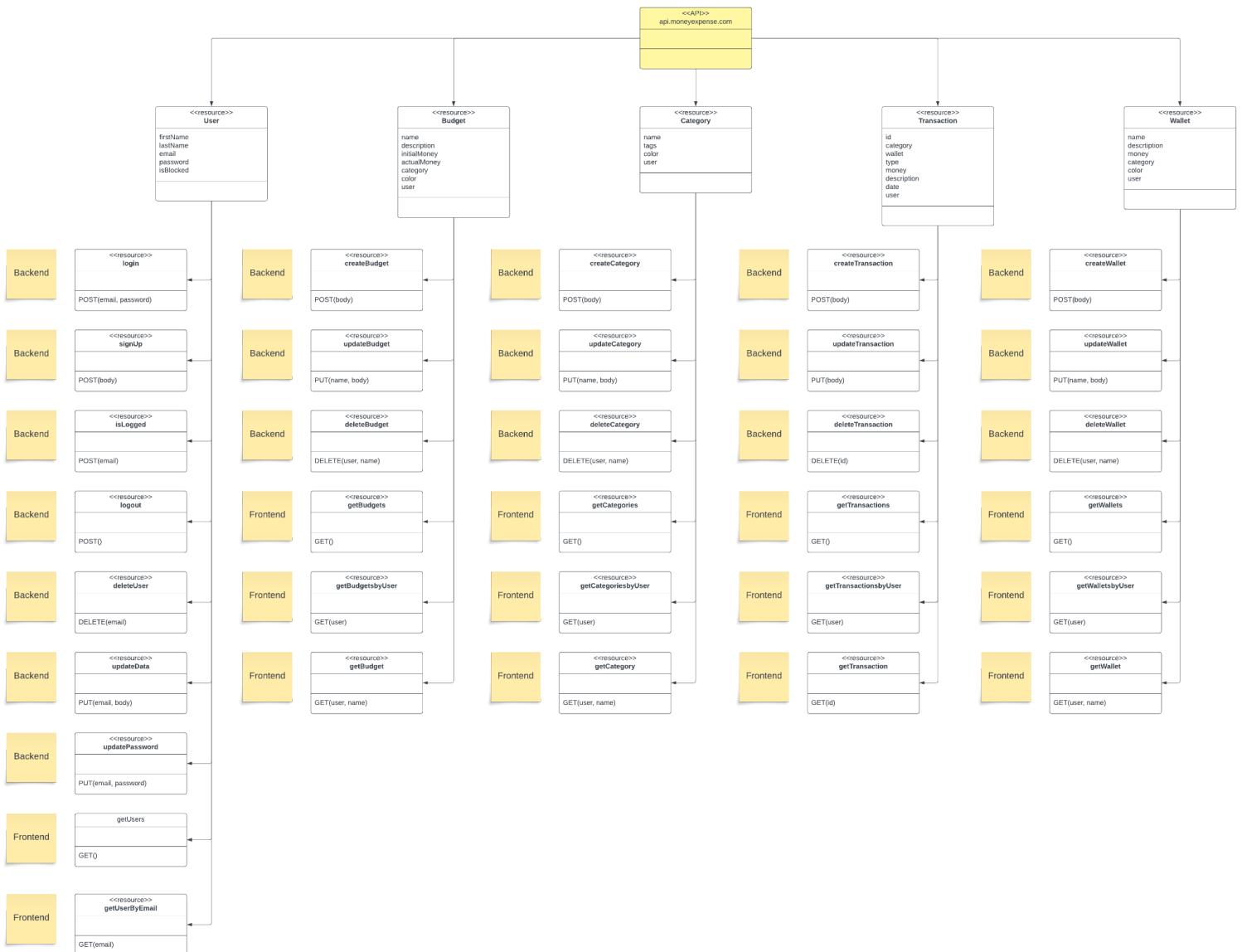


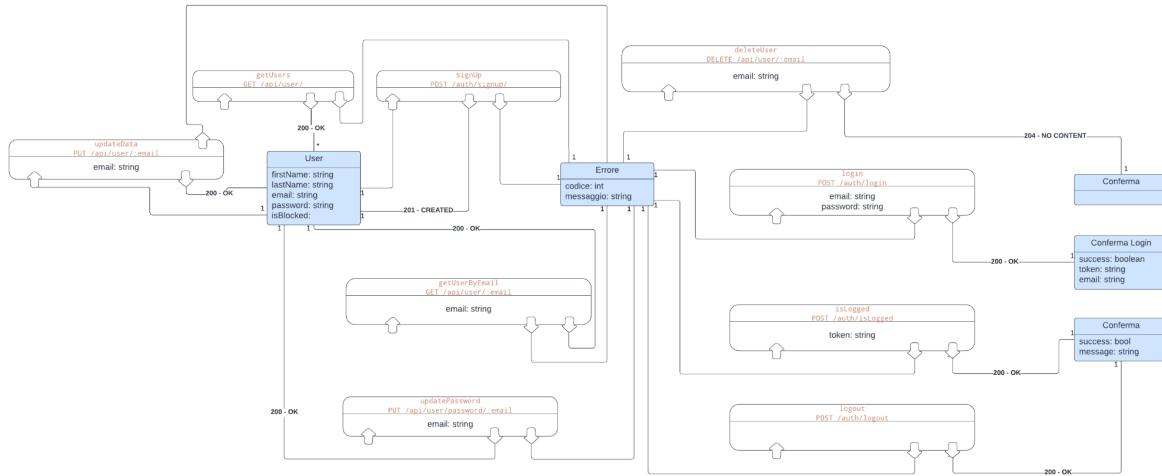
Diagramma delle risorse

Passiamo ora alla rappresentazione effettiva delle varie API presenti in questa versione di Money Expense, e lo facciamo mediante un diagramma di risorse, o diagramma delle API. Essendo la nostra una applicazione con diversi modelli, che sono la colonna portante dell'applicazione, abbiamo deciso di dividere il nostro diagramma in base a questi modelli, per consentire una visione più granulare e separando i vari concetti nel migliore dei modi.

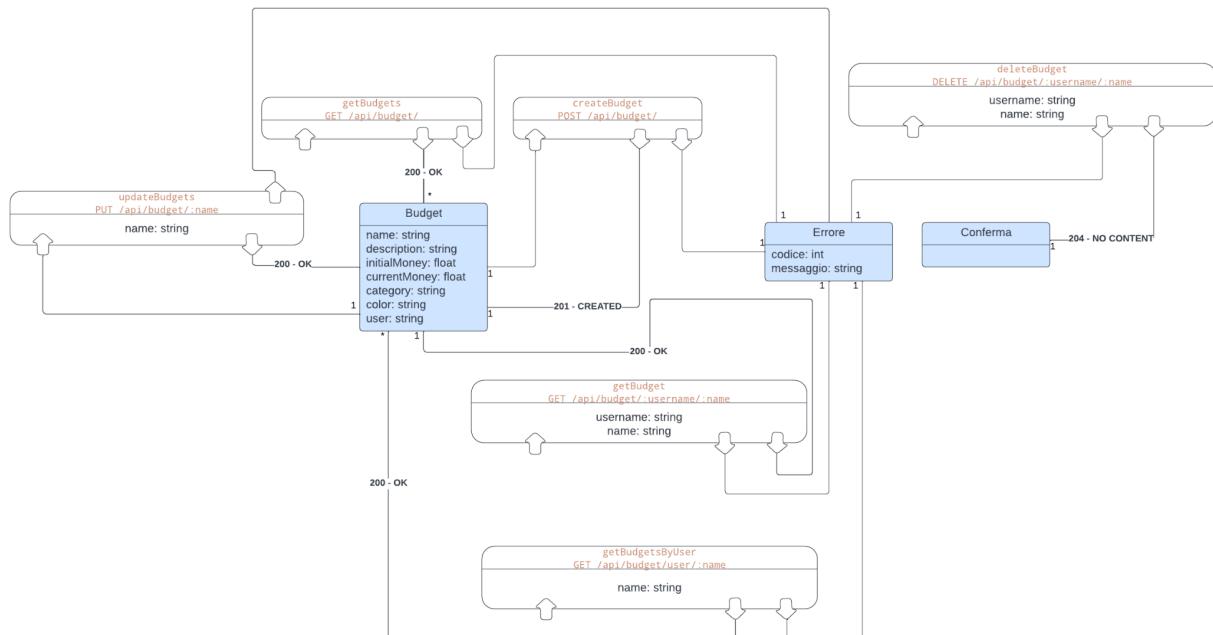
Come è tipico in questi diagrammi, ci sono in ogni caso specificati input ed output di ogni API che il sistema espone. Ci sono dei casi in cui varie API hanno output comuni, come nel caso degli errori. Per evitare di complicare eccessivamente il diagramma, abbiamo quindi deciso di accorpare tutti gli errori in una classe comune, che si occupa di portare lo status code. Nel caso invece di successo, se la API non ritorna nulla, viene creata una classe Conferma, che sta solo a significare che non ci sono stati problemi, e che la operazione richiesta è andata a buon fine.

Di seguito si trovano i vari diagrammi, e infine una vista completa di tutte le risorse.

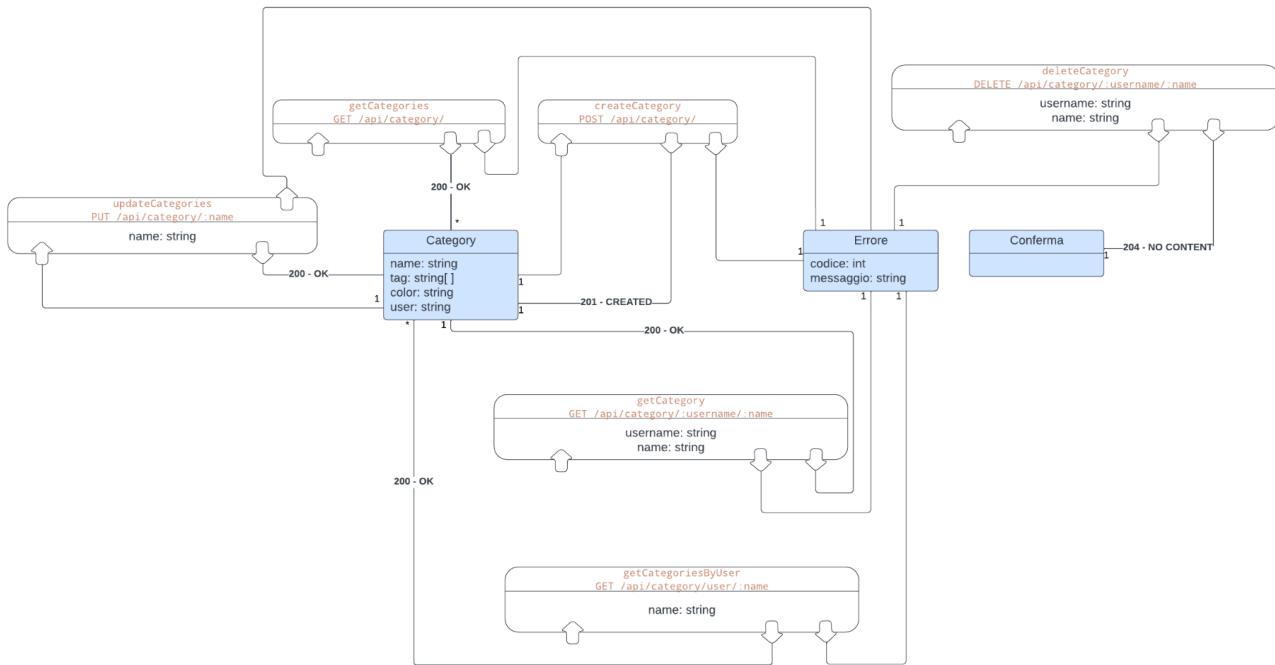
Resource Diagram 1



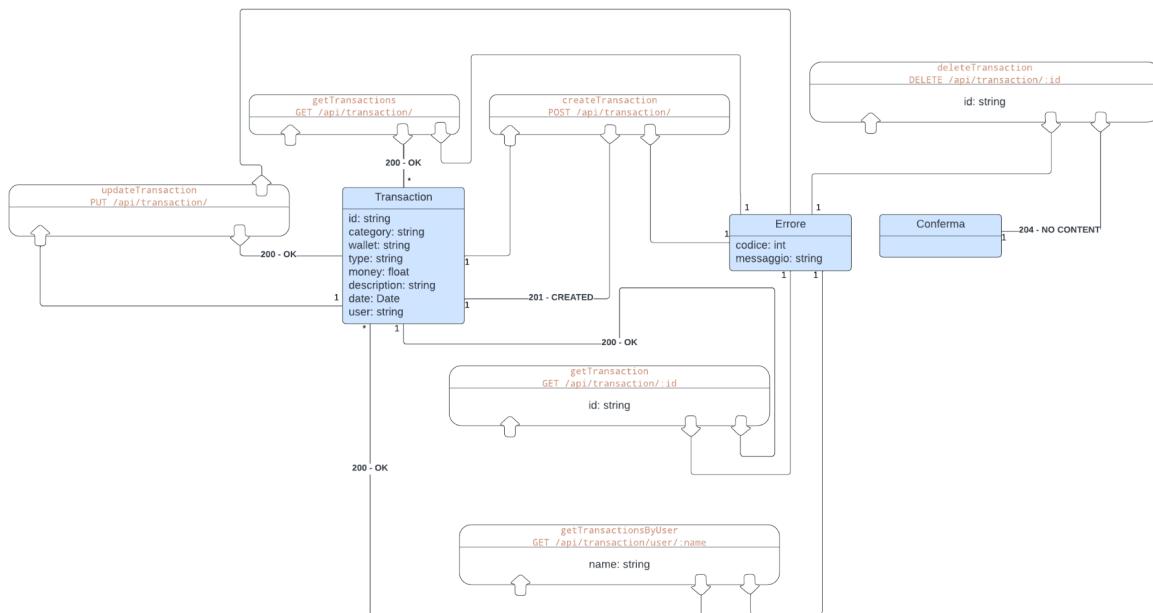
Resource Diagram 2



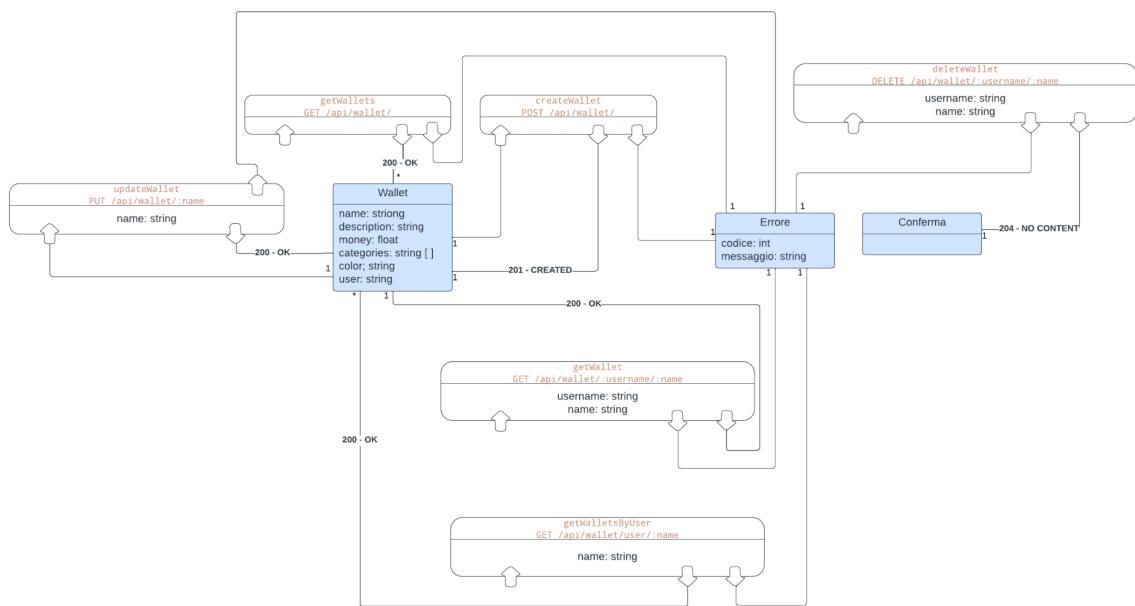
Resource Diagram 3



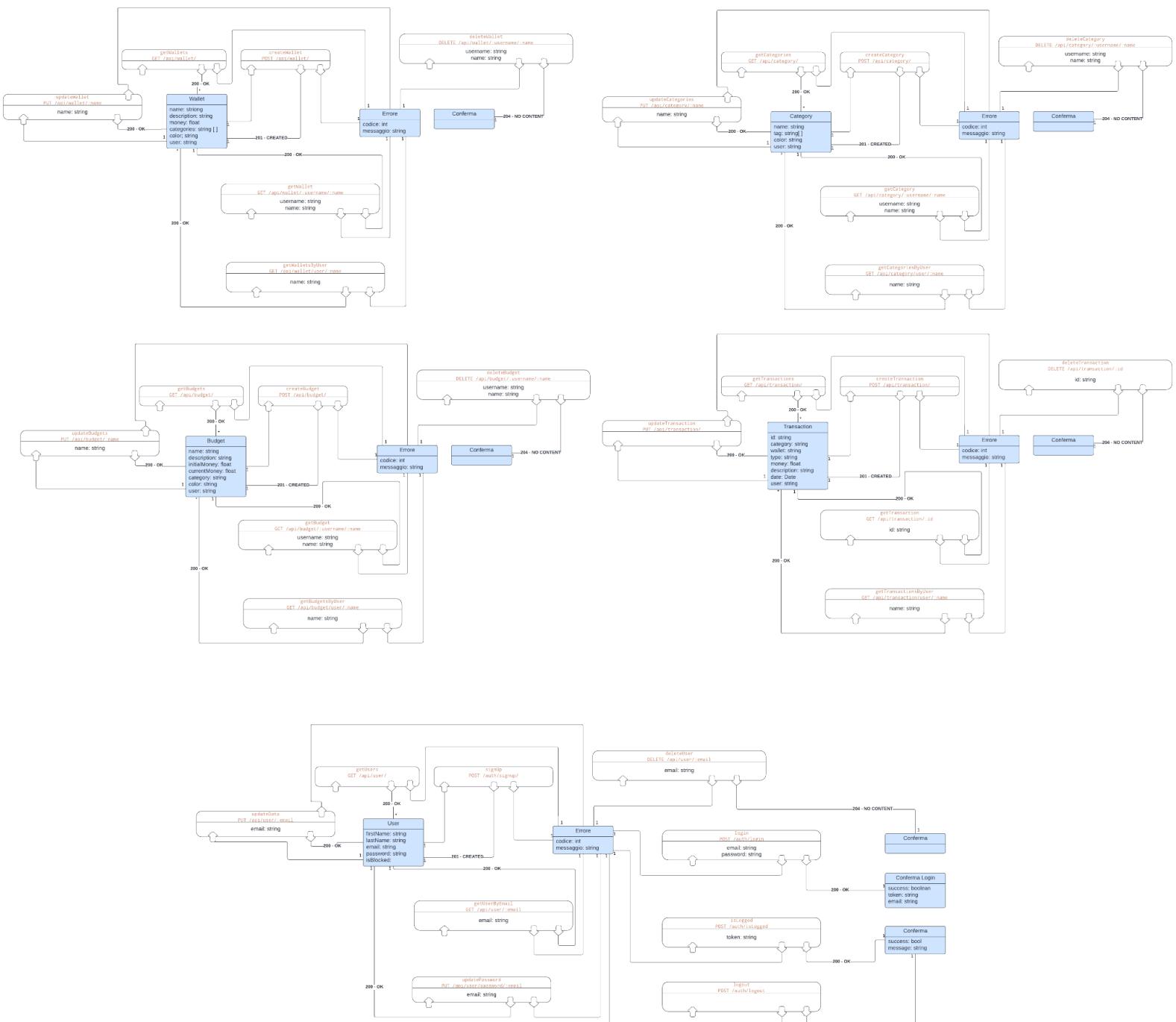
Resource Diagram 4



Resource Diagram 5



Resource Diagram Completo



Implementazione dell'applicazione

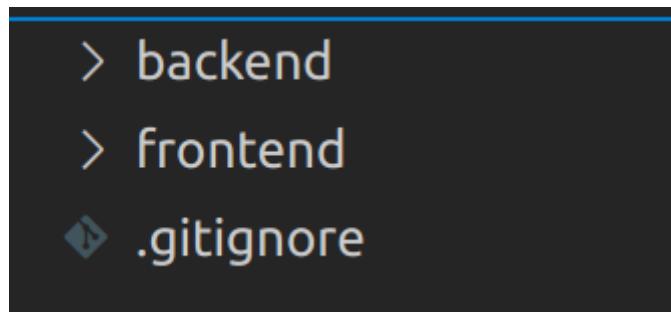
Per la creazione di Money Expense, abbiamo scelto di adottare Typescript come linguaggio principale sia per il front-end che per il back-end. Nello specifico, il front-end è stato sviluppato utilizzando la libreria UI ReactJS.

Per il back-end, abbiamo selezionato NodeJS in combinazione con il framework ExpressJS.

Per quanto riguarda il database utilizzato per memorizzare i dati necessari, abbiamo scelto MongoDB, poiché si integra perfettamente con questo ecosistema.

Struttura del progetto

La struttura del progetto base la possiamo mostare nell'immagine sottostante.



Abbiamo deciso di scomporre il piu' possibile la logica del codice, in modo da formare il piu' possibile moduli indipendenti, ma che fossero in grado di lavorare insieme nel miglio modo possibile, separando i vari concerns, quindi frontend e backend in questo caso.

La directory contiene quindi:

- Cartella backend, dove tutta la logica del backend, le API e i modelli risiedono
- Cartella frontend, dove tutto il frontend dell'applicazione risiede, con tutta la logica e gli asset grafici
- file .gitignore, utilizzato per selezionare i file da non salvare in git, perche' sarebbero superflui o con dati sensibili. Nel nostro caso, abbiamo escluso le directory node_modules e il file .env

Piu' nello specifico, nella cartella backend troviamo tutta la logica del backend, scritto in Express, che ci permette di far funzionare correttamente le API che rendono possibile il funzionamento di questa web app nella maniera corretta. Al suo interno troviamo:

- cartella __test__, dove sono contenuti tutti i fali adibiti al testing della nostra API
- cartella api, che contiene un solo file Typescript, che serve a esporre l'applicazione a Vercel, ovvero il servizio usato per deployare il backend
- cartella controller dove ci sono tutte le logiche delle API. Qui si trova effettivamente come le varie funzioni interagiscono con le richieste HTTP provenienti dall'esterno e modificano o interrogano il database a dovere. Questa e' la logica di business core della nostra applicazione
- cartella coverage, cartella generata da jest, software per testing, che ci permette di visualizzare come la nostra API ha performato e il coverage dei nostri test cases
- cartella models contiene tutte le strutture dati implementate per questa applicazione e che vengono in questa forma salvate poi nel database che utilizziamo, ovvero MongoDB. Questo per dare uniformita' ai dati della piattaforma

- cartella public che contiene solo un file .gitkeep. E' richiesto per poter fare il deploy su Vercel in maniera corretta. Servirebbe a gestire i file statici, che in questo caso la nostra piattaforma non gestisce
- cartella router contiene tutti i router di Express. I router sono componenti logici che si occupano di gestire il traffico in entrata, per indirizzarlo alla corretta funzione in questione, presa adeguatamente tra quelle disponibili nella business logic dei controller
- cartella utils contenente varie funzioni di utilita' che vengono utilizzate piu' volte nella applicazione, ma non appartengono a nessuno degli altri domini fino ad ora esplorati
- file index.ts contiene la logica che permette al server delle API di funzionare. In particolare, si occupa di connettersi al database, di creare tutte le funzioni, di creare un middleware e si mette poi in ascolto di eventuali richieste dal frontend. Questo e' il componente che fa partire tutto il sistema correttamente
- file jest.config.js, file con le configurazioni di jest, ovvero la suite di testing per Javascript/Typescript che abbiamo utilizzato per la nostra API
- file package.json contiene le informazioni necessarie a Node relative a cio' che abbiamo installato e agli script che utilizziamo
- file package-lock.json. File di node con tutte le versioni di dipendenze correntemente installate
- file swagger.json, il file contenente tutta la documentazione integrale della API del sistema di Money Expense, che permette all'esterno di visualizzare correttamente il contenuto di questa API
- file tsconfig.json, file di configurazione per l'utilizzo di Typescript localmente
- file vercel.json e' un file di configurazione che avvisa il servizio di deployment Vercel dove andare a cercare l'applicazione da esporre

Nella seguente immagine appare la directory backend appena descritta:

```

> __test__
> .vercel
> api
> controller
> coverage
> models
> node_modules
> public
> router
> utils
⚙️ .env
TS index.ts
JS jest.config.js
{} package-lock.json
{} package.json
{} swagger.json
TS tsconfig.json
{} vercel.json

```

Per quanto riguarda invece tutta la parte di codice relativa al frontend, quella viene inserita nell'apposita cartella. Questa contiene tutti i file tipici di una applicazione in ReactJS, con qualche libreria grafica esterna utilizzata. Molti file presenti in questa directory sono simili a quelli visti in backend, in quanto entrambi utilizzano Javascript/TypeScript. Questi file comuni sono:

- package.json
- package-lock.json
- tsconfig.json

Dopo di che, in questo caso troviamo i file tailwind.config.js e postcss.config.js, che sono file di configurazione per librerie come Tailwind, che abbiamo utilizzato per rendere più gradevole l'interfaccia web. Nella stessa directory, ci sono le due cartelle tipiche di React:

- public: contenente i file pubblici, quindi i vari asset quali immagini, e il file HTML di base
- src: contenente soprattutto i componenti grafici del frontend, quindi tutte le varie viste, sottopagine e file di stile che permettono la personalizzazione dell'esperienza in piattaforma

Di seguito una immagine di come appare la cartella

TODO: quando finito frontend anche

Dipendenze del progetto

Per poter far funzionare correttamente il progetto, abbiamo utilizzato varie dipendenze esterne, che ci hanno permesso l'astrazione di molta logica complicata, e ci hanno permesso di porre il focus effettivo su ciò' che dovevamo implementare. Come in tutti i

progetti Node, tutte le dipendenze finiscono nei file package.json, come spiegato precedentemente. Qui sono le dipendenze che abbiamo usato e a cosa sono servite nello sviluppo di tale applicazione.

- cors: modulo nativo di Javascript per consentire il Cross-Origin Resource Sharing protocol, senza doverlo implementare
- dotenv: permette di accedere alle variabili di ambiente. In locale, queste si trovano in un file .env, in deployment si trovano nelle configurazioni di sistema
- express: framework che permette la creazione semplice di API, mediante la creazione automatica di router. Questo astrae molta della logica per la gestione delle chiamate HTTP, permettendoci di sviluppare la parte di interazione con il Database
- jsonwebtoken: modulo per creare e gestire un token d'accesso. Si occupa della creazione e verifica di token JWT
- mongoose: libreria che consente la creazione di modelli per MongoDB e una interazione semplificata con questo database
- swagger-ui-express: tool usato per documentare e testare le API esposte da Money Expense
- jest e supertest: librerie utilizzate per fare il testing. Jest è la suite javascript per il testing, mentre supertest permette di chiamare API rest per testarle in maniera adeguata

Abbiamo inoltre, all'interno del nostro file package.json creato i seguenti script, utili per la fase di development.

```
"scripts": {  
  "build": "npx tsc",  
  "start": "node dist/index.js",  
  "dev": "nodemon index.ts",  
  "vercel-build": "echo hello",  
  "test": "jest --coverage --detectOpenHandles --forceExit"  
},
```

Questi ci hanno permesso, in ordine, di compilare Typescript, di lanciare il server compilato, di lanciare il server in modalità development, di fare il deploy su Vercel, e di eseguire i test di jest in locale

Modelli nel database

Per poter modellare adeguatamente i dati che dobbiamo presentare nella nostra applicazione, e' stata necessaria la creazione di modelli di database, partendo dal nostro diagramma delle classi. Ognuno di questi modelli ci permette di strutturare adeguatamente l'utilizzo della nostra applicazione, e di essere consistenti. Abbiamo individuato cinque modelli principali, che ora andiamo a spiegare nel maggiore dettaglio.

Modello User

Per modellare e salvare i dati degli utenti abbiamo utilizzato tale modello User, che contiene tutti i dati necessari a noi in piattaforma.

```
const schema = new Schema({
  _id : {type:String},
  firstName: {type:String, required:true},
  lastName: {type:String, required:true},
  email: {type:String, required:true},
  password: {type:String, required:true},
  isBlocked: {type:Boolean, required:true}
});
```

Abbiamo inserito i parametri firstName e lastName per caratterizzare l'utente, in modo che sia l'interazione con la UI sia maggiormente personalizzata.

Abbiamo poi inserito il campo email e password, che permettono all'utente di eseguire l'operazione di login in piattaforma. L'email deve essere univoca, non possiamo avere piu' di un utente con la stessa email, altrimenti non sarebbe possibile gestire il login di tale user.

Poi, un campo isBlocked, che permette di bloccare le azioni di un utente, nel caso vi fosse la necessita'.

Infine, un attributo _id che MongoDB crea in automatico, per rendere univoco il record appena creato in fase di signUp, e quindi creare un utente univoco anche in caso di modifica credenziali.

Di seguito un esempio di un possibile record del modello User

```
_id: "mariorossi@gmail.com1701929271481"
firstName: "Mario"
lastName: "Rossi"
email: "mariorossi@gmail.com"
password: "passwordSecret!123"
isBlocked: false
__v: 0
```

Modello Budget

Per rappresentare invece i vari budget che l'utente crea in piattaforma, e' stato creato l'apposito modello Budget.

```
const schema = new Schema({
  _id : {type:String},
  name: {type:String, required:true},
  description: {type: String},
  initialMoney: {type: Number, required:true},
  actualMoney: {type: Number, required:true},
  category: {type: String, required:true},
  color: {type:String},
  user: {type:String, required:true},
});
```

Anche in questo caso, ha un attributo `_id` che lo identifica univocamente nel database. Dopo di che viene caratterizzato da un nome, e da una descrizione testuale decise dall'utente. Questo per permettere di capire a che budget ci stiamo riferendo in questo momento.

Possiede due attributi numerici, `initialMoney` e `actualMoney`, per dire con quanto budget siamo partiti, e quanto ne abbiamo ancora a disposizione, per poter mostrare statistiche e visualizzazioni UI piu' gradevoli. Questo insieme all'attributo `color`, scelto dall'utente.

Per categorizzare meglio ogni budget, abbiamo messo un campo `category`, per dire a che categoria questo appartiene, e il campo `user`, che mostra quindi l'utente che possiede tale budget in piattaforma.

Di seguito un esempio di un possibile record del modello Budget

```
_id: "Grocerymariorossi@gmail.com1701929901372"
name: "Grocery"
initialMoney: 100
actualMoney: 100
category: "Home expenses"
color: "#ff0"
user: "mariorossi@gmail.com"
__v: 0
```

Modello Category

Per poter compartimentare correttamente le informazioni derivanti da ogni categoria che un utente puo' creare in piattaforma, abbiamo creato un modello Category deputato a questo compito.

```
const schema = new Schema({
  _id : {type:String},
  name: {type:String, required:true},
  tags: {type: [String]},
  color: {type:String},
  user: {type:String, required:true},
});
```

Questo modello, salva un nome e un colore, che caratterizzano le categorie create dall'utente in piattaforma. Dopo di che, una serie di tags, che possono aiutare a fornire una descrizione della categoria, cosi' che l'utente possa sempre ricordare cosa e' memorizzato sotto questo ombrello.

Infine, l'utente che possiede e ha creato la categoria, con l'_id univoco che viene creato in MongoDB, per garantire unicita' dei record.

Di seguito un esempio di un possibile record del modello Category

```
_id: "Foodmariorossi@gmail.com1701930051621"
name: "Food"
tags: Array (2)
color: "#ff0"
user: "mariorossi@gmail.com"
__v: 0
```

Modello Transaction

Per salvare correttamente e in maniera consistente tutte le transazioni che un utente salva in piattaforma, abbiamo creato l'apposito modello per i record di tipo transazione.

```
const schema = new Schema({
  _id : {type:String},
  category: {type:String, required:true},
  wallet: {type:String, required:true},
  type: {type: String, enum: ["income", "expense"], required:true},
  money: {type: Number, required:true},
  description: {type: String},
  date: {type: Date, default: Date.now},
  user: {type:String, required:true},
});
```

In questo modello, abbiamo inserito, come sempre, _id come identificatore universale delle transazioni registrate.

Dopo di che, per caratterizzare specificamente la tipologia di transazione, abbiamo messo il parametro type, che e' una stringa che puo' essere solo "expese" o "income". Per poi dire quanto e' stato mosso, un apposito campo money.

Per chiedere di che categoria fa parte la spesa, abbiamo messo un apposito campo category, che ci permetterà poi di legarci ai vari budget. Inoltre, abbiamo aggiunto il campo wallet che ci permetta di scalare i soldi dal portafoglio selezionato. Ovviamente, ogni transazione avviene in una data, e quindi abbiamo aggiunto anche questo apposito campo.

Per caratterizzare ulteriormente una transazione, abbiamo messo la possibilita' all'utente di scrivere una descrizione della stessa. Per poi dire a chi appartiene una transazione, abbiamo aggiunto l'apposito campo user.

Di seguito un esempio di un possibile record del modello Transaction

```
_id: "2023-12-07T06:42:29.414Z - mariorossi@gmail.com"
category: "Grocery"
wallet: "Paypal Family"
type: "expense"
money: 20
description: "Bought vegetables"
date: 2023-12-07T06:42:29.414+00:00
user: "mariorossi@gmail.com"
__v: 0
```

Modello Wallet

Per infine caratterizzare tutti i wallet, quindi le fondi di liquidita', registate dall'utente in piattaforma per poter fare migliore contabilita', abbiamo creato il modello Wallet.

```
const schema = new Schema({
  _id : {type:String},
  name: {type:String, required:true},
  description: {type: String},
  money: {type: Number, required:true},
  categories: {type: [String], required:true},
  color: {type:String},
  user: {type:String, required:true},
});
```

Il modello, per essere caratterizzato, presenta i campi name, description e color, che gli permettono di avere un nome, una descrizione esaustiva creata dall'utente, e un colore per rendere la UI maggiormente piacevole e positiva da usare.

Per rappresentare dati di utilita', ci sono i campi money, per dire quanti soldi ora presenti in esso, categories, per dire a quali categories puo' essere adibito, ed infine user, per dire che utente possiede tale wallet in piattaforma.

Anche in questo caso, abbiamo un _id univoco per ogni wallet

Di seguito un esempio di un possibile record del modello Wallet

```
_id: "Paypal Familymariorossi@gmail.com1701931119609"
name: "Paypal Family"
description: "Money for buying foods"
money: 100
categories: Array (2)
color: "#00f"
user: "mariorossi@gmail.com"
__v: 0
```

Sviluppo delle API

In questa parte del documento ci occupiamo di descrivere come funzionano le varie API presenti nel progetto Money Expense, ed eventuali funzioni in esse chiamate. In particolare, saranno presenti chiamate alla API di MongoDB, che ci permettono di gestire adeguatamente il database a seconda delle nostre necessita'.

La descrizione si limitera' ad una spiegazione testuale, in quanto lo sviluppo effettivo di codice sarebbe troppo prolioso. Il codice e' disponibile nella cartella controller, nella sezione backend della nostra repository di GitHub, nel caso si volesse consultare.

API per modello User

In questo progetto, un utente puo' interagire con le risorse che la piattaforma fornisce soltanto se effettivamente loggato, per motivi legati alla privacy. Per questo motivo, ad eccezione degli endpoint dell'API che si occupano del login e della creazione di nuovi utenti, tutte le altre risorse sono protette dall'autenticazione provvista dal middleware, mediante un apposito token creato dal database, e salvato nell'header, nella sezione x-access-token. Non esiste un altro modo di accedere alle risorse della piattaforma.

Tutti gli utenti

L'API in questione si occupa di ritornare una lista di tutti gli utenti memorizzati in piattaforma. Questa API torna utile per la parte di sviluppo. Non esiste una casistica in cui il client richiede mai l'utilizzo di una tale funzionalita'. Serve quindi solo in fase di sviluppo per poter controllare eventuali errori.

Per fare in modo che questa API funzionasse, il metodo find() di MongoDB viene utilizzato, che ritorna semplicemente tutti gli utenti

Trova utente per email

Questa API ci permette di trovare informazioni relative ad uno specifico utente, che viene caratterizzato mediante l'email. Questo e' possibile farlo, in quanto l'email e' un campo unique, e quindi non esistono piu' utenti con la stessa email. Questo metodo ci e' utile in fase di richiesta delle informazioni dell'utente.

Per far funzionare tale API, viene usato il metodo findOne() di MongoDB, che ritorna la sola istanza che rispetta il campo email. Questo parametro testuale era stato in precedenza passato mediante URL dall'utente.

Modifica Dati

Questo medoto dell'API ci permette di modificare le informazioni relative ad uno specifico utente, siano queste nome, cognome, email o password. Permette quindi una piu' granulare modifica dei dati dell'utente in modo trasparente.

Per capire che utente dobbiamo modificare, nei parametri URL viene passata la mail. Poi, dal body prendiamo i nuovi dati che ci vengono forniti, e passiamo quindi alla modifica di tale record del database. MongoDB espone la funzionalita' chiamata findOneAndUpdate che permette di trovare un singolo record e modificarlo con i dati che gli si forniscono.

Modifica Password

Vista la maggior frequenza e necessita' di modificare la propria password piuttosto che l'intero profilo, abbiamo deciso di inserire nell'API un metodo apposito che permettesse di fare specificamente questo.

Questo metodo prende dall'URL la email dell'utente che ha bisogno di modificare la propria password, e nel body della richiesta ha una nuova password. Prima di completare la modifica dello user, verifica che la password rispetti i requisiti di coplessita' e robustezza che la piattaforma si pone (gestiti da una apposita funzione, messa nella repository utils). Verificati tali requisiti, si passera' all'effettiva modifica della password dell'utente, che avra' effetto immediato. Anche qui la funzione findOneAndUpdate di MongoDB viene utilizzata per completare l'operazione con successo.

Elimina User

Se un utente volesse eliminare dalla piattaforma il proprio account, questo endpoint glielo permette. L'API espone tale funzionalita', che torna utile anche in fase di sviluppo, quanto per testing molti utenti finti vengono creati, e devono in un secondo momento essere eliminati dalla piattaforma.

Il metodo prende dall'URL la email dell'utente specifico che si desidera eliminare. Si passa dunque alla ricerca e all'eliminazione dell'utente. MongoDB permette di fare facilmente questa operazione mediante il metodo deleteOne(), che cerca un unico record e lo elimina autonomamente.

Si noti come questo endpoint sia estremamente critico, e quindi sia meglio affidarsi alle chiamate delle funzionalita' di MongoDB, per evitare errori che possano ledere all'utente.

API di Autenticazione

Login utente

L'API di login permette all'utente di venire loggato in piattaforma e poter quindi usufruire dei servizi da essa fornita. L'utente fornisce una email e una password per verificare la sua identita'. A questo punto, si passa con la funzione findOne di MongoDB alla ricerca di tale utente. Se l'user non dovesse esistere, ci sarebbe un feedback dall'API. Se esiste, si passa ora al controllo di correttezza della password inserita. Qualora anche questo passaggio andasse a buon fine, l'API si incarica di creare e validare un token JWT mediante la libreria jsonwebtoken, al quale da validita' di 12 ore. Questo viene poi ritornato nella risposta, cosi' che l'utente possa richiedere dunque le sue risorse in piattaforma.

Logout utente

Questa API vuole fare in modo che la sessione di un utente termini la sua sessione. Per terminare una sessione, e' necessaria che una sessione fosse stata iniziata, quindi questo endpoint e' chiamabile solo da utenti loggati in piattaforma.

L'API prende il token che l'utente usava per loggarsi in piattaforma e accedere alle proprie risorse, e ritorna un messaggio che garantisce che e' stato revocato, chiudendo quindi con successo la sessione di navigazione.

Controllo se utente e' loggato (isLoggedIn)

Questo metodo specifico della API ci permette di capire se un utente e' loggato oppure no in piattaforma. E' semplicissimo. Si occupa di leggere il token che viene provvisto negli header della richiesta HTTP dell'utente che sta usando la piattaforma. Se il token non e' presente oppure il token e' scaduto, allora l'API rispondera' che l'utente non e' loggato. Se invece il token e' presente ed e' ancora valido, si avra' risposta affermativa.

Questo e' il funzionamento che ha anche il middleware che si occupa di arbitrare le richieste degli utenti in piattaforma.

SignUp utente

Questa API si occupa della creazione di un nuovo utente in piattaforma. Questa API viene messa sotto autorizzazione e non sotto utente, in quanto tutte le API utente sono controllate da un middleware che controlla che l'utente sia loggato. In questo caso, un utente non puo' essere loggato se non ha un proprio profilo. Per questo abbiamo deciso di mettere SignUp sotto le API di autenticazione.

In questo metodo, vengono passati nel body tutti i vari attributi necessari alla creazione di un nuovo utente in piattaforma. Al termine della operazione, se ha tutto avuto successo, viene ritornato il nuovo utente creato con i suoi dati.

I metodi utilizzati sono findOne() per assicurarsi che la email sia unica. Dopo di che, si salva l'utente in maniera permanente, mediante un save()

API per modello Budget

Crea budget

Quando un utente desidera creare un nuovo budget in piattaforma, puo' utilizzare il seguente endpoint dell'API. Crea in piattaforma la nuova risorsa apposita.

Per funzionare, prende dal body della richiesta di tipo POST i dati necessari per una nuova istanza di un budget, e poi la salva nel database. Per fare cio', MongoDB espone la funzionalita' save() che salva un oggetto che aderisce ad un modello specifico creato nella sezione model (vedi sezione Modelli nel Database).

Viene fatto pero' prima un controllo, mediante la funzione findOne, che tale utente non abbia gia' creato un budget con lo stesso nome, altrimenti il tutto sarebbe confusionario. Nonostante cio' e' possibile che due utenti diversi abbiano un budget con il medesimo nome. Al termine della creazione, la API ritorna la nuova risorsa creata.

Tutti budget

Questa API ci permette di andare ad estrarre dal database tutti i budget attualmente presenti in piattaforma. Anche questa API e' solo di supporto agli sviluppatori, in quanto non esiste un utente che abbia bisogno di vedere tutti i budget salvati e presenti in piattaforma.

Per fare questo, e' sufficiente utilizzare la funzione find() di MongoDB, che per un dato schema ritorna tutte le istanze salvate.

L'API ritorna quindi tutte le risorse di tale modello.

Trova budget per utente

Quando volgiamo trovare tutti i budget di un determinato utente per poterli listare nella sua pagina, tale API viene utilizzata. Prende dall'URL l'email dell'utente che vuole fare tale richiesta, e in base a quella va alla ricerca delle varie risorse.

Per cercare tutti i budget di un utente, si utilizza la funzione `find()` di MongoDB mettendo come filtro la email ricevuta.

Si ritornano quindi di conseguenza tutte le istanze trovate nel database.

Trova un budget

Lo scopo di questa API e' quello di trovare le informazioni relative ad uno specifico budget in piattaforma.

Per poter capire che budget specifico, si utilizza la email dell'utente e il nome del budget, i quali vengono forniti nella stringa URL, e vengono quindi letti dal sistema di backend. MongoDB utilizza il metodo `findOne()` che permette quindi la ricerca del singolo record nel database, e ci permette di trovare la risorsa specifica di cui avevamo bisogno, e che sara' la risposta alla richiesta che abbiamo ricevuto.

Modifica budget

Questa API si pone come obiettivo la modifica di uno specifico budget. Nell'URL gli passiamo solamente il nome. Questo perche' la proprieta' di un budget non cambia, e quindi l'utente lo possiamo trovare nel body, con tutti i nuovi dati che l'utente vuole eventualmente modificare.

MongoDB ci fa utilizzare il metodo `findOneAndUpdate`, che fa una modifica del record trovato. Alla fine, rispondiamo con la risorsa modificata.

Elimina budget

Se un utente desidera eliminare un budget, questa API gli permette di farlo.

Nell'URL gli forniamo la email dell'utente e il nome del budget che desideriamo rimuovere dal nostro database, e MongoDB, mediante il metodo `deleteOne()` si occupera' di rimuoverlo in maniera automatica.

Ritorniamo un messaggio vuoto, che sta a significare che l'operazione e' andata a buon fine. Non abbiamo infatti alcun dato da mostrare, in quanto ne abbiamo solo eliminati.

API per modello Category

Crea category

Lo scopo di tale metodo e' quello di creare una nuova category in piattaforma, di modo che un utente possa usufruirne.

Per fare cio' viene passato un body contenente tutte le informazioni necessarie. Il sistema si occupa quindi di controllare che l'utente non abbia gia' una categoria con questo nome, per evitare confusione, anche se diversi utenti possono avere categorie con nomi comuni. Per fare questo viene usato il metodo `findOne()`. Dopo questo check locale, si passa alla creazione.

MongoDB aiuta nella creazione mediane il metodo `save()`, che salva la nuova risorsa, che viene anche ritornata come risposta all'utente finale.

Tutte category

API di supporto agli sviluppatori, dedicata alla ricerca e al display di tutte le category presenti in piattaforma. E' solo di supporto in quanto un utente non puo' vedere le categorie create da altri utenti.

Per fare si che questa API funzioni correttamente, viene usato il metodo find() di MongoDB, che trova tutte le istanze che poi sono ritornate nella risposta.

Trova category per utente

Quando si desidera trovare tutte le category di uno specifico user, questo endpoint e' quello che va chiamato.

L'utente viene passato come parametro dell'URL, mediante la sua email. Dopo di che, si passa a controllare il database, con il metodo find() di MongoDB, al quale mettiamo il filtro della email ricetura, e ritorniamo come risposta tutte le istanze trovate nella query.

Trova una category

Lo scopo di tale API e' quello di trovare una singola specifica risorsa di tipo category nel database. Viene fatto questo cercando mediante un email dell'utente e il nome della categoria. Questo perche' la combinazione di questi attributi rendono la categoria unica nel database. I parametri sono passati mediante URL.

MongoDB utilizza il metodo findOne(), a cui poniamo come filtri i parametri prima descritti, e ritorna la singola istanza trovata nel database.

Modifica category

Lo scopo di questo metodo dell'API e' la modifica di una specifica category all'interno del database.

Viene fornito mediante URL il nome della categoria. Ancora, visto che l'utente che la possiede non cambia, quello lo prendiamo mediante il body della funzione. Leggiamo quindi il body e modifichiamo tale record.

Il record lo troviamo e aggiorniamo con la funzione findOneandUpdate() di MongoDB, che astrae tutta la logica per noi. Ritorniamo alla fine, l'istanza modificata.

Elimina category

Lo scopo di questa API e' quello di eliminare una specifica category dal database.

La category in questione viene univocamente identificata dal nome della stessa, e dalla email del suo proprietario, ovvero l'utente che l'aveva creata.

Viene quindi utilizzato su tale risorsa il metodo deleteOne() di MongoDB, che si occupa della sua eliminazione. Il messaggio di risposta e' un codice 204, quindi senza contenuto.

API per modello Wallet

Crea wallet

Quando un utente desidera creare un wallet, puo' utilizzare questa specifica API. Essa crea in piattaforma la nuova risorsa apposita.

Per funzionare, prende dal body della richiesta di tipo POST i dati necessari per una

nuova istanza di un wallet, e poi la salva nel database. Per fare cio', MongoDB espone la funzionalita' save()

Viene fatto pero' prima un controllo, mediante la funzione findOne, che tale utente non abbia gia' creato un wallet con lo stesso nome, altrimenti il tutto sarebbe confusionario. Nonostante cio' e' possibile che due utenti diversi abbiano un wallet con il medesimo nome. Se l'operazione ha avuto successo, la API ritorna la nuova risorsa creata.

Tutti wallet

Questa API ci permette di andare ad estrarre dal database tutti i wallet attualmente presenti in piattaforma. Anche questa API e' solo di supporto agli sviluppatori, e non viene mai direttamente chiamata dal sistema di frontend..

Per fare funzionare correttamente questo endpoint, e' sufficiente utilizzare la funzione find() di MongoDB, che per un dato schema ritorna tutte le istanze salvate.

L'API ritorna quindi tutte le risorse di tale modello.

Trova wallet per utente

Quando volgiamo trovare tutti i wallet di un determinato utente per poterli listare nella sua pagina, tale API viene utilizzata. Prende dall'URL l'email dell'utente che vuole fare tale richiesta, e in base a quella va alla ricerca delle varie risorse.

Per cercare tutti i wallet di un utente, si utilizza la funzione find() di MongoDB mettendo come filtro la email che e' stata trovata nella stringa URL.

Si ritornano quindi di conseguenza tutte le istanze trovate nel database.

Trova un wallet

Lo scopo di questa API e' quello di trovare le informazioni relative ad un wallet a cui l'utente e' attualmente interessato. .

Per poter capire che risorsa specifica, si utilizza la email dell'utente e il nome del wallet, i quali vengono forniti nella stringa URL.

MongoDB utilizza il metodo findOne() che permette quindi la ricerca del singolo record nel database, e ci permette di trovare la risorsa specifica di cui avevamo bisogno, e che sara' la risposta alla richiesta che abbiamo ricevuto.

Modifica wallet

Questa API si pone come obiettivo la modifica di un wallet. Nell'URL gli passiamo il nome di tale risorsa. Basta questo parametro in quanto il proprietario di un wallet non cambia, e quindi l'utente lo possiamo trovare nel body, con tutti i nuovi dati che l'utente vuole eventualmente modificare.

MongoDB ci fa utilizzare il metodo findOneAndUpdate, che fa una modifica del record trovato. Alla fine, rispondiamo con il wallet modificato a dovere.

Elimina wallet

Se serve eliminare un wallet, questa API si occupa di far cio', evitando dati inutili in piattaforma. .

Nell'URL gli forniamo la email dell'utente e il nome del wallet che desideriamo rimuovere dal nostro database, e MongoDB, mediante il metodo deleteOne() lo rimuovera'.

Ritorniamo un messaggio vuoto, che sta a significare che l'operazione e' andata a buon fine. Il codice HTTP e' quindi il 204.

API per modello Transaction

Crea transaction

Lo scopo di tale metodo e' quello di creare una nuova transaction, per la registrazione di nuove informazioni.

Per fare cio' viene passato un body contenente tutte le informazioni necessarie. Il sistema si occupa quindi di controllare che l'utente possieda un wallet e una categoria con nomi corrispondenti a quelli passati nel body.. Dopo questo check, si passa alla creazione.

MongoDB esegue la creazione mediane il metodo save(), che salva la nuova risorsa, che verra' ritornata come risposta all'utente finale.

Tutte transaction

Questa e' una API di supporto agli sviluppatori, dedicata alla ricerca e al display di tutte le transaction presenti in piattaforma.

Per fare si che questa API funzioni correttamente, viene usato il metodo find() di MongoDB, che trova tutte le transaction che poi sono ritornate nella risposta.

Trova transaction per utente

Quando si desidera trovare tutte le transaction di uno specifico user, viene usato questo specifico endpoint.

L'utente viene passato come parametro dell'URL, mediante la sua email. Dopo di che, si passa a controllare il database, con il metodo find() di MongoDB,che filtra per quella mail, e ritorniamo come risposta tutte le istanze trovate nella query.

Trova una transaction

Lo scopo di tale API e' quello di trovare una singola transaction nel database. Viene fatto questo cercando mediante l'_id univoco della transazione specifica, che la rende unica nel database. Questo parametro e' ricevuto mediante URL.

MongoDB utilizza il metodo findOne(), e ritorna la singola istanza trovata nel database.

Modifica transaction

Lo scopo di questo metodo dell'API e' la modifica di una specifica transaction dalle risorse disponibili create da un utente. .

Viene fornito mediante URL l'_id univoco della transaction interessata.Leggiamo quindi il body e modifichiamo tale transaction

Il record lo troviamo e aggiorniamo con la funzione findOneandUpdate() di MongoDB, che esegue l'aggiornamento della risorsa. Ritorniamo alla fine, l'istanza modificata.

Elimina transaction

Lo scopo di questa API e' quello di eliminare una specifica transaction dal database.

La risorsa viene univocamente identificata dal suo identificativo universale, salvato nel suo `_id`, e passato nella stringa URL dall'utente. .

Viene quindi utilizzato su tale risorsa il metodo `deleteOne()` di MongoDB, che si occupa della sua eliminazione. Il messaggio di risposta e' un codice 204, quindi senza contenuto.

Documentazione delle API

Tutte le API discusse fino ad ora ed implementate in questa applicazione, e che permettono quindi a Money Expense di funzionare correttamente sono state documentate facendo utilizzo dell'interfaccia Swagger, che e' lo standard de facto per ogni applicazione web attualmente. In questo modo, la documentazione e' visibile a tutti quanti sia lato codice che lato deployment.

Se si volesse consultare la documentazione nel momento il cui le API sono fatte funzionare localmente, l'endpoint da chiamare sarebbe il seguente:

<http://localhost:3000/api-docs>

Ovviamente il numero di porta varia a seconda di quelllo che viene inserito nel file `.env`, quindi il link potrebbe variare.

The screenshot shows the Money Expense Rest API documentation. At the top, there's a header with the Swagger logo and "Supported by SMARTBEAR". Below the header, the title "Money Expense Rest API" is displayed with a version of "1.0.0" and an "OAS 2.0" badge. A note indicates the "Base URL: localhost:3000/" and the "API documentatio for the project Money Expenses". A "MIT" license link is also present. A dropdown menu for "Schemes" is set to "HTTP". The main content area lists several categories: "User", "Authentication", "Category", "Budget", "Transaction", and "Wallet", each with a brief description and a collapse/expand arrow.

La vista iniziale della documentazione delle API e' suddivisa in compartimenti, che permettono di dividere un po' i vari utilizzi e scopi di ogni funzione. Quindi, la pagina di seguito e' come si presenta inizialmente la nostra documentazione.

Volendo entrare nello specifico di una data API di cui si e' piu' interessati, si possono aprire i vari tag facendo click su quello di proprio interesse. A questo punto, il menu a tendina si aprira' e verranno mostrati tutti gli endpoint sviluppati e pubblicati dall'applicazione in questione. Ogni endpoint e' caratterizzato sia testualmente che mediante il colore con il metodo a cui appartiene e fa riferimento, sia questo GET, POST, PUT o DELETE.

Riportiamo di seguito un esempio della documentazione per il modello Transaction, per mostrare come appaiono tutte le altre documentazioni registrate per l'API.

Transaction API per il modello 'Transaction'. Gestisce le transazioni in piattaforma ^

POST /api/transaction/ Creates a transaction ▼

GET /api/transaction/ Get all transactions ▼

PUT /api/transaction/ Modifies a given transaction ▼

GET /api/transaction/user/{name} Get all transactions that are owned by a given user ▼

GET /api/transaction/{id} Gets a given transaction ▼

DELETE /api/transaction/{id} Deletes a transaction ▼

Ogni endpoint si trova in una situazione simile, descritte e documentate interamente da Swagger. In questo modo la funzionalita' e' facile da capire ed utilizzare.

Ogni API ha quindi una breve descrizione testuale, e se si apre il menu a tendina specifica, Swagger mostra i parametri necessari eventuali nell'URL, nel header (come nel caso dei token) e nel body. Tutto allegato di esempio. Al termine della pagina si trova anche una lista con tutti i possibili codici di ritorno che la API puo' restituire, in base al risultato. Di seguito riportiamo l'esempio dell'endpoint incaricato della creazione di una nuova Transaction.

POST /api/transaction/ Creates a transaction ^

Parameters Try it out

Name	Description
body object (body)	Example Value Model <pre>{ "user": "mario.rossi@gmail.com", "category": "Grocery", "wallet": "Paypal Account", "type": "expenses", "money": 12, "description": "Bough the vegetables for the week" }</pre>

Parameter content type application/json

Responses Response content type application/json

Code	Description
201	Created
401	No token provided
404	Wallet not Found
500	Internal Server Error

Implementazione frontend

Nella seguente sezione illustreremo le numerose schermate front-end che abbiamo sviluppato.

Ecco in breve un elenco delle schermate realizzate per questo progetto:

- Signup
- Login
- Dashboard
- Wallets
- Transactions
- Settings

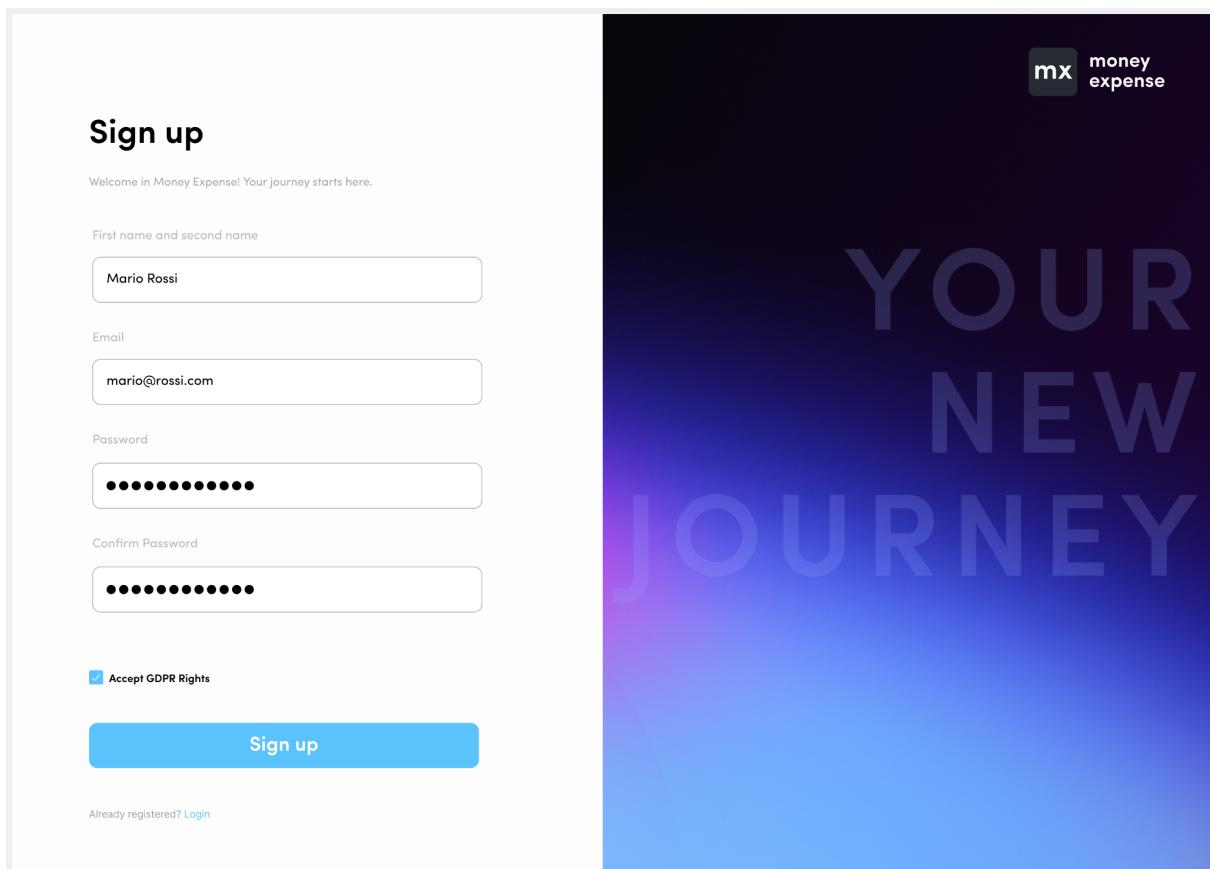
Signup

La schermata Signup dà il benvenuto all'utente e presenta una UI interface che permette di registrare un nuovo account al sistema inserendo i relativi dati.

Si noti che tutti i requisiti devono essere rispettati.

All'utente viene inoltre richiesto di accettare i diritti GDPR.

Infine, dopo aver inserito tutti i dati in maniera corretta l'utente è invitato a cliccare sul bottone [Sign up](#) per procedere con la creazione dell'account.



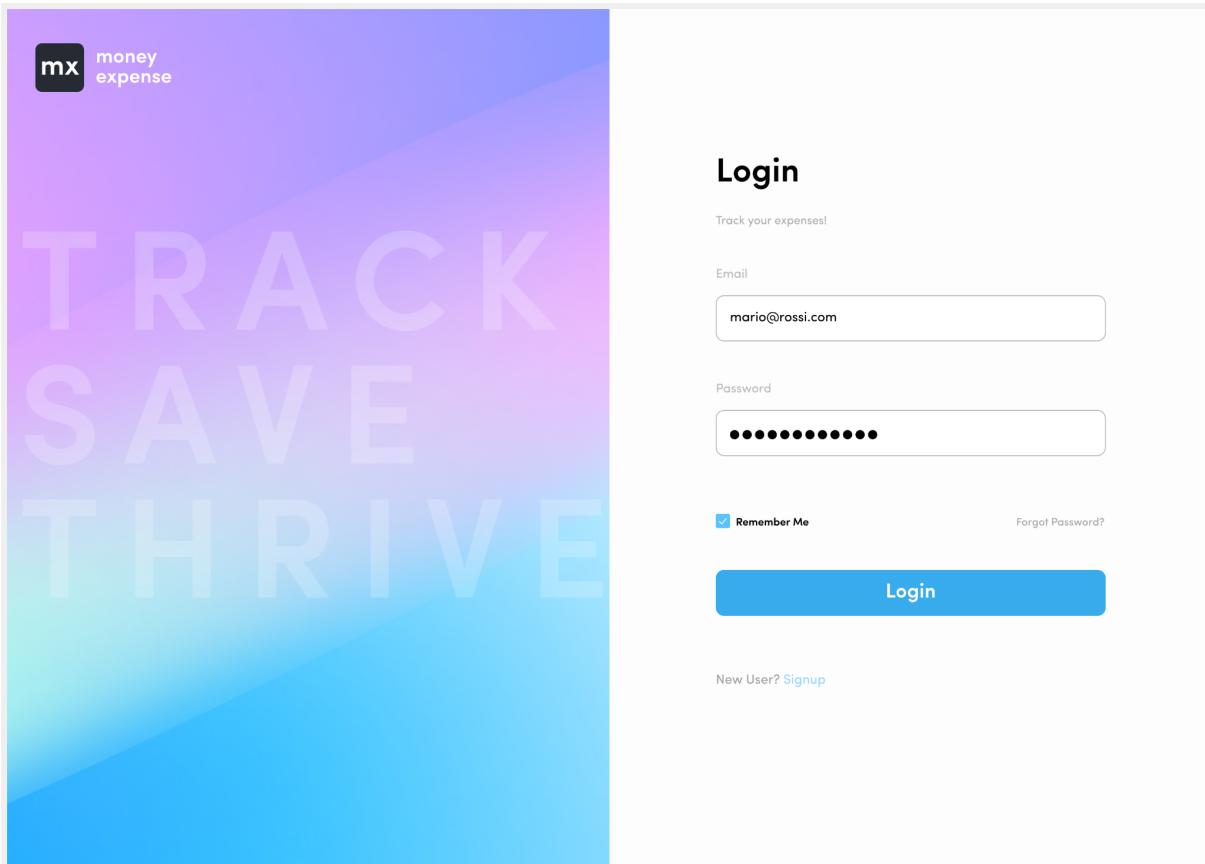
Login

La schermata Login viene utilizzata dagli utenti già registrati alla piattaforma.

Si richiede di inserire email e password.

In caso l'utente voglia memorizzare la sessione per futuri accessi, ha la possibilità di selezionare l'opzione [Remember me](#).

Infine, per procedere con la creazione di una nuova sessione il sistema richiede di cliccare il bottone [Login](#).



Dashboard

Una volta passata la fase di login, l'utente viene trasferito nella [Dashboard](#), la pagina principale dell'applicazione.

Osserviamo che la pagina è suddivisa in 2 sezioni. Quella di sinistra è dedicata alla navigazione nell'app ed è rappresentata dalla Sidebar. Quella di destra invece presenta il contenuto e l'insieme di funzionalità disponibili.

La dashboard permette di:

- Creare una nuova transazione tramite il bottone [Add transaction](#).

- Filtrare ogni dato per il periodo.
- Visualizzare il bilancio sulla wallet selezionata nella sezione [Balance](#).
- Visualizzare l'arrivo e l'uscita di somme di denaro nella sezione [Income/expenses](#).
- Tenere traccia delle somme spese per categoria in [Expense for the last <periodo>](#).
- Elencare l'insieme delle ultime transazioni avvenute nella sezione [Last transactions](#).

The screenshot shows the 'Dashboard' page of the 'money expense' application. On the left, a sidebar lists navigation options: Dashboard (highlighted in blue), Wallets, Transactions, Categories, Budgets, Settings, and Sign out. The main content area includes a 'Welcome back, Mario Rossi' message and a 'American Express Platinum' dropdown. The top right features a 'Dashboard' title, a date updated '12 hours ago', and time filters (1D, 1M, 3M, 6M, 1Y) with '3M' selected. A large central box displays 'Your current balance' as € 1234567.89. To the right, there are two boxes: 'IN' (€ 1920.56) and 'OUT' (€ 420.123). Below these are sections for 'Expenses for the last 3 months' (home accessories, entertainment, food) and 'Last transactions' (multiple entries for a transaction with Ivano).

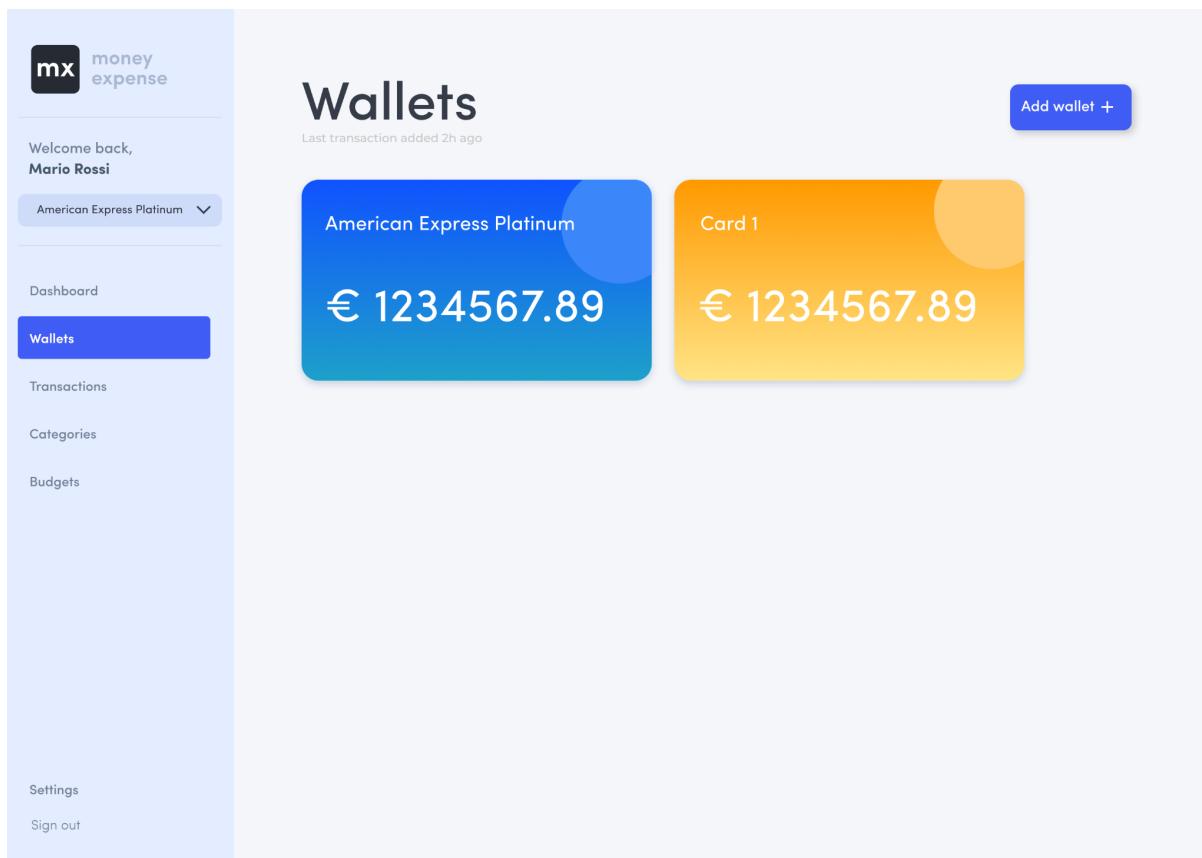
Wallets

L'utente ha la possibilità di accedere alla pagina con l'elenco di tutte le wallets registrate alla piattaforma.

Nella schermata [Wallets](#) sono presenti le seguenti componenti:

- Bottone per l'aggiunta di una wallet in cui sono richieste le informazioni relative alla wallet da inserire.

- Un elenco a griglia in cui troviamo la rappresentazione di ogni wallet mediante delle card. Su ogni card vengono visualizzati il nome e il bilancio attuale della wallet.



Transactions

La seguente schermata permette di visualizzare, aggiungere, modificare, eliminare e svolgere delle operazioni di filtraggio e ordinamento sulle transazioni registrate finora. Contiene i seguenti componenti:

- In alto a destra troviamo il bottone [Add transaction](#), cliccato il quale si apre un modal popup che richiede di inserire i dati relativi alla transazione. Al completamento si può cliccare il tasto [Add](#) ed effettuare l'aggiunta del record della transazione al database.
 - In centro viene mostrata una toolbar. Questa permette di:
 - Ricercare una transazione per il parametro [Description..](#)
 - Ordinare le transazioni per parametri come [Date](#) e [Amount](#).
 - Filtrare le transazioni per parametri come [Type](#) (expense/income), [Date](#) e [Amount](#).
 - Buona parte della UI interface viene occupata da una tabella di transazioni in cui compaiono 6 colonne: 4 descrittive (Type, Description, Date, Amount) e 2 funzionali (Edit, Delete).
- Le colonne descrittive hanno il compito di visualizzare l'informazione della singola transazione:

- **Type**, indicato da 2 cerchi concentrici di 2 possibili colori, rosso e verde, rispettivamente per expense e income.
- **Description**, descrizione del motivo dell'avvenuta transazione.
- **Date**, la data in cui tale transazione è stata eseguita.
- **Amount**, la somma relativa alla transazione.

Le colonne funzionali invece permettono di manipolare la singola transazione facendo richiesta all'API dell'applicazione:

- **Edit**, cliccando l'icona del bottone edit si apre un popup con un form per la modifica dei campi descrittivi della transazione.
- **Delete**, cliccando l'icona del bottone delete si apre anche in questo caso un form per l'eliminazione della transazione inizialmente dall'elenco lato frontend e successivamente dal database.

In fondo alla tabella è stato posizionato un componente di paginazione per facilitare la visualizzazione degli elementi dell'elenco.

Description	Date	Amount
Spesa al supermercato Pane, formaggio, prosciutto...	26/09/23	-15.00€
Stipendio Arrivata paga mensile	26/09/23	+1500.00€
Spesa al supermercato Pane, formaggio, prosciutto...	26/09/23	-15.00€
Pagamento mensile palestra Palestra	26/09/23	-70.00€

Categories

La pagina delle categorie offre la possibilità di visualizzare tutte le categorie create. In questa schermata sono presenti le seguenti componenti:

- Bottone per aggiungere una nuova categoria con una breve descrizione

- Un elenco a griglia che mostra le categorie esistenti con la somma spesa in quella specifica categoria, premendo su una della card è possibile modificarne i dati o eliminarli.

The screenshot shows the 'Categories' section of the mx money expense app. On the left, there's a sidebar with navigation links: Dashboard, Wallets, Transactions, Categories (which is highlighted in blue), Budgets, Settings, and Sign out. The main area has a title 'Categories' and a sub-header 'You have a total of 3 categories'. There are three cards, each representing a category: 'Home' (€129.62 spent on home accessories), 'Entertainment' (€129.62 spent on entertainment), and 'Food' (€129.62 spent on food). Each card has a small icon and a link to 'Spent on [category]'. At the top right, there are time filters (1D, 1M, 3M, 6M, 1Y) and a button 'Add Category +'. The '3M' filter is selected.

Budget

La pagina per i budget permette all'utente di visualizzare, modificare e creare nuovi budget di spesa, in questa schermata sono presenti le seguenti componenti:

- Bottone per aggiungere un nuovo budget
- Un elenco a griglia che mostra i budget precedentemente creati e l'avanzamento attuale del piano, cliccando sulla card è possibile modificare i dati o eliminare il budget.

The screenshot shows the 'Budgets' screen of the mx money expense app. On the left is a sidebar with the 'mx money expense' logo at the top. Below it are links: 'Welcome back, Mario Rossi', a dropdown menu set to 'American Express Platinum', and a list of categories: 'Dashboard', 'Wallets', 'Transactions', 'Categories', 'Budgets' (which is highlighted in blue), 'Settings', and 'Sign out'. The main content area has a title 'Budgets' and a sub-header 'You have a total of 1 monthly budget'. It features a large blue button with the text 'Home budget' and '€129.62 / 300.00'. Below this, it says 'Budget for Home expenses'. At the top right of the main area are buttons for time periods: '1D', '1M', '3M' (which is selected and highlighted in blue), '6M', and '1Y'. A blue button labeled 'Add Budget +' is located at the top right of the main content area.

Settings

The screenshot shows the Money Expense mobile application interface. On the left is a sidebar with the 'money expense' logo at the top. Below it, a welcome message 'Welcome back, Mario Rossi' and a dropdown menu for 'American Express Platinum'. The sidebar also contains links for 'Dashboard', 'Wallets', 'Transactions' (which is highlighted in blue), 'Categories', 'Budgets', 'Settings', and 'Sign out'. The main content area is titled 'Transactions' and shows a list of recent transactions. At the top right of this area are buttons for 'Add transaction' and 'Filter by'. Below the title, a message says 'Last transaction added 2h ago'. A search bar with placeholder text 'Search for transactions...' is followed by 'Sort by' and 'Filter by' dropdowns. The transaction list has three columns: 'Description', 'Date', and 'Amount'. Each transaction row includes a small circular icon (red for expenses, green for income), the transaction description, the date (26/09/23), the amount (e.g., -15.00€ or +1500.00€), and edit (pencil) and delete (trash bin) icons. The transactions listed are: 'Spesa al supermercato' (-15.00€), 'Stipendio' (+1500.00€), 'Spesa al supermercato' (-15.00€), and 'Pagamento mensile palestra' (-70.00€). At the bottom of the transaction list, there is a page navigation bar with numbers 1, 2, 3 (highlighted in blue), 4, and 5.

Description	Date	Amount
Spesa al supermercato Pane, formaggio, prosciutto...	26/09/23	-15.00€
Stipendio Arrivata paga mensile	26/09/23	+1500.00€
Spesa al supermercato Pane, formaggio, prosciutto...	26/09/23	-15.00€
Pagamento mensile palestra Palestra	26/09/23	-70.00€

La seguente schermata invece ha il compito di visualizzare e eventualmente modificare i dati personali di un utente registrato.

Osserviamo un'unica componente presente nell'applicazione. Qui troviamo 4 campi di input:

- 2, First name e Second name modificabili,
- 1, Email non modificabile.
- 1, Password può essere modificata cliccando il bottone Change password. A questo punto si apre un modal popup che richiede di inserire e confermare una nuova password. Al click sul bottone **Save** i dati relativi alla password vengono salvati con successo se la nuova password rispetta i requisiti.

Per salvare tutti i dati è necessario cliccare sul bottone **Save**.

Testing

Vediamo ora come e' stato effettuato il testing del progetto.

Per poter testare le nostre funzionalita', sono state usate le librerie standard de facto di Javascript/Typescript, ovvero Jest, con l'aggiunta della libreria Supertest, usata per le REST API in particolare.

Il codice e' stato messo nella cartella `__test__`, come propongono le linee guida ufficiali presenti nella documentazione del package. Di seguito una immagine della repository con i vari file.

```
▽ __test__
  TS auth.test.ts
  TS budget.test.ts
  TS category.test.ts
  TS transaction.test.ts
  TS user.test.ts
  TS utils.test.ts
  TS wallet.test.ts
```

Abbiamo quindi creato questi file, il cui compito e' quello di effettuare il testing. Ognuno di questi file contiene tutte le funzioni deputate al testing di un modulo particolare della nostra piattaforma, che sia esso una API oppure le funzioni utility che abbiamo creato per scrivere codice migliore. Abbiamo cosi' diviso per evitare di generare file di test enormi, e separare un po' i vari domini della piattaforma.

Così' appare un file di testing:

```

const request = require('supertest');
import app from '../index';
import {server} from '../index';
import { default as mongoose } from 'mongoose';
import {describe, expect, test} from '@jest/globals';
const jwt = require('jsonwebtoken');
require('dotenv').config();
import { generateToken } from '../utils/token';

let token = "";

> beforeAll(async () => { ...
  })

> afterAll(async () => { ...
  })

> describe("Test API GET /api/category", () => { ...
  });

> describe("Test API GET /api/category/user/:name", () => { ...
  });

> describe("Test API GET /api/category/:user/:name", () => { ...
  });

> describe("Test API POST /api/category", () => { ...
  });

> describe("Test API PUT /api/category/:name", () => { ...
  });

```

La struttura dei vari file e' molto simile, in quanto seguono tutti il framework proposto da jest per performare i test. Iniziamo importanto tutti i package necessari al testare le varie API o funzionalita'. Nello specifico, importiamo sempre mongoose per poter utilizzare il database, supertest per poter testare la piattaforma, e jsonwebtoken per generare token utili a simulare un utente reale in piattaforma, e procedere ai test. Importiamo poi le funzionalita' di jest per il testing. Importiamo anche il server dell'applicazione, cosi' da poterlo utilizzare e chiudere una volta finiti i vari tests. In alcuni casi, ci sono anche gli import delle funzioni specifiche che volgiamo testare, come nei test delle funzioni utils.

Nel metodo beforeAll() mettiamo un semplice timeout, per vedere se l'applicazione performa sufficientemente veloce rispetto a quanto ci aspettiamo. Nel metodo afterAll() chiudiamo la connessione al database e il server della piattaforma, cosi' che la routine di testing possa terminare con successo, e restituire i risultati. In alcuni casi, e' necessario pulire il database con la funzione afterAll(), se sono stati creati dei dati "dummy", utili solo ai fini del test.

Ogni API viene testato nel proprio dominio, all'interno del metodo describe di jest. Dentro a questo macro dominio, ci sono tutti i vari metodi test, che vengono chiamati per verificare la corretta funzionalita' della API in questione. Per ogni API verifichiamo tutti i vari codici di ritorno possibili, anche multiple volte se sono possibili piu' modi per raggiungerli, garantendo quindi massima robustezza e sicurezza.

Questo e' un esempio di un dominio di describe():

```

describe("Test API GET /api/category", () => {
  // Put the token in the headers
  test("Chiamata all'API in maniera corretta", async () => {
    const response = await request(app).get("/api/category").set("x-access-token", token).set("Content-Type", "application/json").send();
    expect(response.statusCode).toBe(200);
    expect(response.body).not.toBeNull();
    expect(response.body).not.toBeUndefined();
    expect(response.body.length).toBeGreaterThanOrEqual(1);
  });

  test("Chiamata all'API senza token", async () => {
    const response = await request(app).get("/api/category").set("Content-Type", "application/json").send();
    expect(response.statusCode).toBe(401);
    expect(response.body).not.toBeNull();
    expect(response.body).not.toBeUndefined();
    expect(response.body.message).toBe("Nessun token fornito");
  });
});

```

Per ogni API (in questo caso per la category), vengono controllati tutti i metodi implementati mediante i test() di jest, passando dati corretti e dati incorretti, per vedere come si comporta questo nostro servizio rispetto a quanto ci aspettiamo.

Abbiamo implementato i vari controlli per tutti i metodi HTTP che abbiamo utilizzato (GET, POST, PUT, DELETE), e controlliamo tutti i campi necessari per essere certi che sia andato tutto a buon fine.

Risultati testing

Per poter testare adeguatamente il nostro sistema, abbiamo creato uno script apposito, messo poi nel file package.json, sezione "scripts":

"test": "jest --coverage --detectOpenHandles"

In questo modo, con **npm test** node lancia in automatico i test, contenuti nei vari file .test.ts, e genera i risultati.

Il flag coverage serve per la generazione della reportistica relativa al coverage del codice nel nostro testing, mentre il flag detectOpenHandles controlla che non ci siano processi nascosti che impediscono al nostro test di terminare, come per esempio il database aperto se non lo chiudessimo adeguatamente.

Questo è come i nostri test hanno formato:

```

Test Suites: 7 passed, 7 total
Tests: 111 passed, 111 total
Snapshots: 0 total
Time: 48.745 s

```

Le sette suite sono passate tutte, così come i nostri 111 tests.

Apriamo quindi anche il report generato in automatico dalla libreria di testing, all'interno di coverage/lcov-report/. Apriamo l'apposito file index.html, e visualizziamo i risultati.

Filter:

File	Statements	Branches	Functions	Lines
backend	97.43%	38/39	50%	75%
backend/controller	87%	261/300	91.13%	75.49%
backend/models	100%	20/20	100%	100%
backend/router	100%	58/58	100%	100%
backend/utils	100%	14/14	100%	100%

Il grosso dei casi di codice non coperti, ci siamo resi conto essere tutti i controlli messi nelle varie funzioni di controllo errori di MongoDB, quindi errori fuori dalla nostra portata, dove chiediamo di ritornare un errore 500. Forniamo di seguito un esempio di tale casistica (riga 50)

```
43  export const getCategories = (req: express.Request, res: express.Response) => {
44    Category.find()
45    .then((data: CategoryType[]) => {
46      res.status(200).send(data);
47    })
48    .catch((err: Error) => {
49      res.status(500).send('Internal Server Error');
50    });
51  }
```

Al di là di queste casistiche che non possiamo coprire, tutta la API è testata e controllata dalla nostra suite di 111 test cases, divisi in 7 sub suite.

GitHub, Deployment e codice locale

Per trovare la repository di GitHub del progetto Money Expense si può accedere al seguente link: <https://github.com/Gog-Ingegneria-del-software/ProgettoGog>. Il nome specifico della repository è ProgettoGog, e fa parte dell'organizzazione Gog-Ingegneria-del-software da noi creata. Per maggiori informazioni circa come è strutturato il codice, si può leggere la sezione **Struttura del Progetto**.

Eseguire il server in locale

Nel caso un cui si volesse eseguire localmente il progetto sulla propria macchina, bisogna seguire le seguenti istruzioni.

- 1) Clonare la repository in locale mediante il seguente link:
<https://github.com/Gog-Ingegneria-del-software/ProgettoGog.git>
- 2) Entrare nella cartella del backend, ed eseguire il comando **npm install** per installare tutte le dipendenze richieste
- 3) Creare nella repository backend un file chiamato **.env**, in cui si andranno ad inserire le seguenti variabili di ambiente, necessarie al funzionamento della piattaforma

PORT=3001

```
JWT_SECRET="secret"  
MONGODB_USERNAME=***  
MONGODB_PASSWORD=***  
MONGODB_URI=***  
-----
```

- 4) Se tutto e' stato creato correttamente, quando viene attivato il comando **npm run dev**, il server dovrebbe partire. Attendere finche' non appaiono le scritte "Server started at <http://localhost:3001>" e "Connected to MongoDB". A questo punto la API e' funzionante
- 5) Aprire un nuovo terminale, e entrare nella cartella di frontend. eseguire il comando **npm install**
- 6) Dopo di che, usare il comando **npm start**, e attendere fino al caricamento.
- 7) Collegandosi ora a <http://localhost:3000/> si potra' accedere alla piattaforma

Note per la documentazione locale

Nel caso si volesse consultare la documentazione del backend in locale, si puo'

chiamare il seguente endpoint: <http://localhost:3000/api-docs/>

Si noti che tutto in locale funziona mediante HTTP, e non HTTPS.

Note per la documentazione online

Nel caso di volesse visualizzare la documentazione anche hostata online, si puo' visitare il seguente link. <https://gog-72hkodcm-matteopossamai.vercel.app/api-docs/>

Si utilizzi HTTPS per visitare tale link