



Money Expense

Documento di sviluppo web app

Scopo del documento

Il seguente documento è il riassunto dello sviluppo di una versione della web app di **Money Expense**, che abbiamo analizzato nei precedenti documenti.

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

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

Passeremo successivamente a descrivere nel dettaglio tutte le API implementate nel nostro codice mediante il diagramma delle risorse e di estrazione delle risorse, *API diagram*, i quali a loro volta 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 insieme alle diverse dipendenze installate.

Forniamo successivamente un resoconto delle pagine create, per poi passare ad una descrizione della repository su GitHub in cui è stato salvato il codice del progetto.

Quindi passeremo alle istruzioni per effettuare il deployment del codice, per poter far funzionare localmente tale applicazione anche su altri dispositivi.

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

User flow

Iniziamo questo documento con lo *user flow*, nel quale vengono mostrate tutte le azioni che un utente può svolgere all'interno della versione della piattaforma sviluppata.

Questo illustra il funzionamento delle interazioni applicazione-utente.

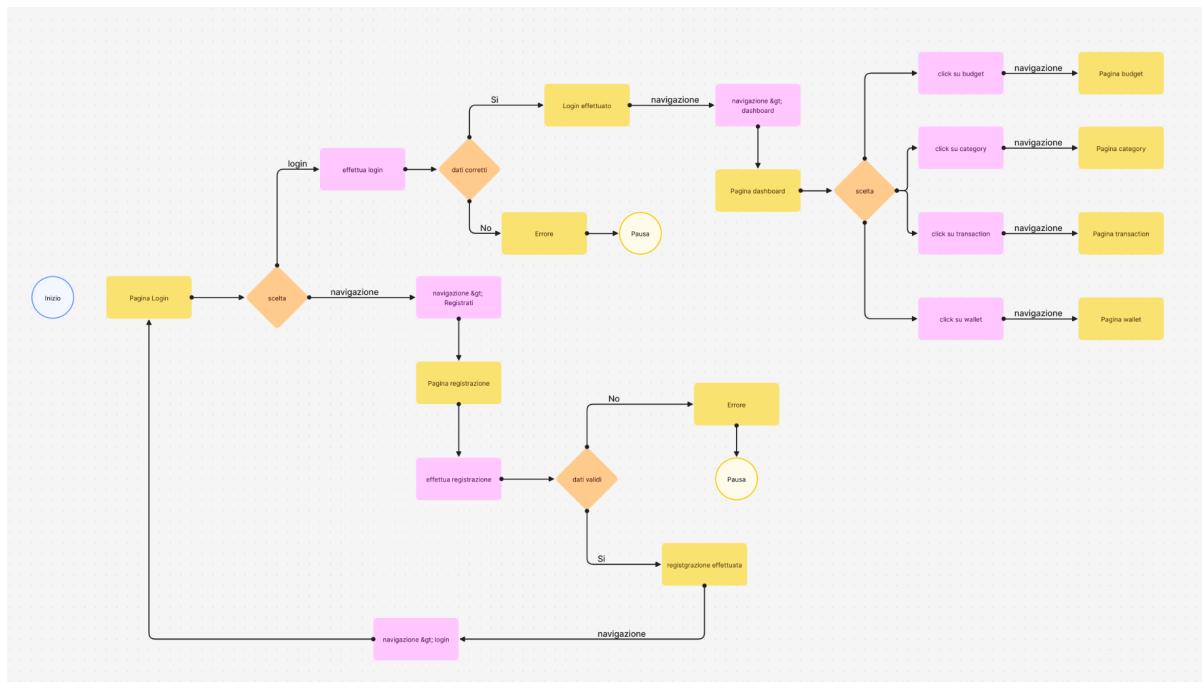


Per chiarire ulteriormente lo schema che segue, riportiamo una breve legenda dei componenti che verranno inseriti in questo user flow diagram.

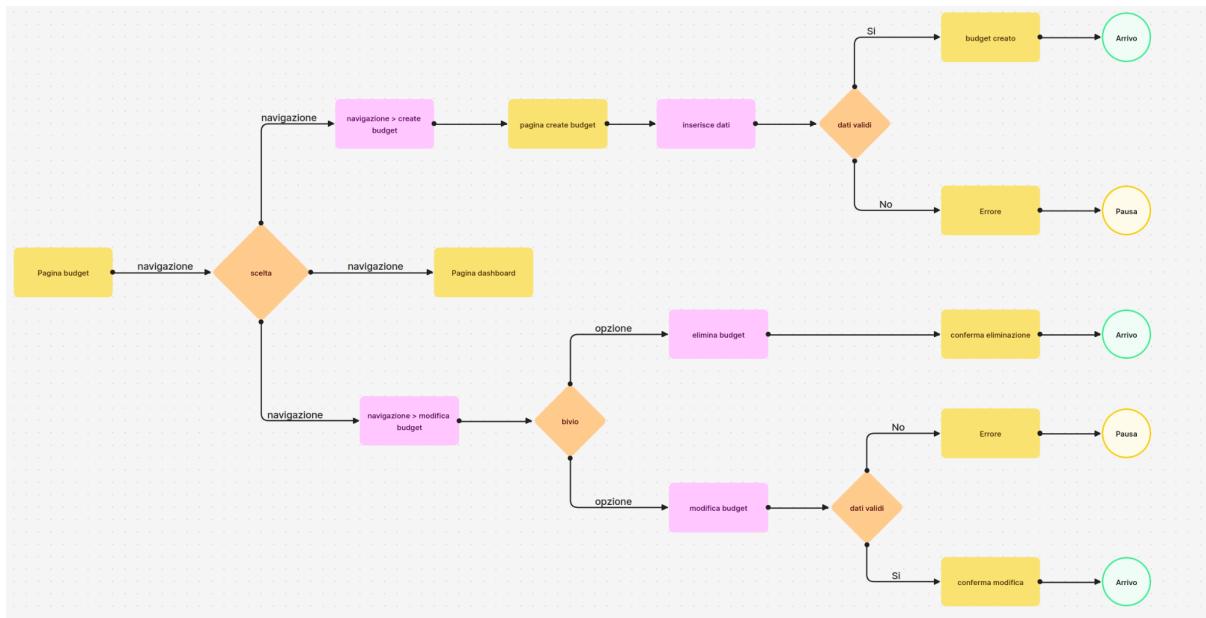
Visti i vari flussi, abbiamo deciso di partizionare il nostro user flow completo in molteplici moduli. Questi procedono in parallelo e sono tutti collegati. Hanno infatti ognuna delle sezioni e dei blocchi comuni, che permettono di collegare tutta la logica implementata. Con questo approccio volevamo permettere una maggiore separazione di interessi, e permettere una maggiore leggibilità del diagramma in sé.

Di seguito i vari componenti del diagramma.

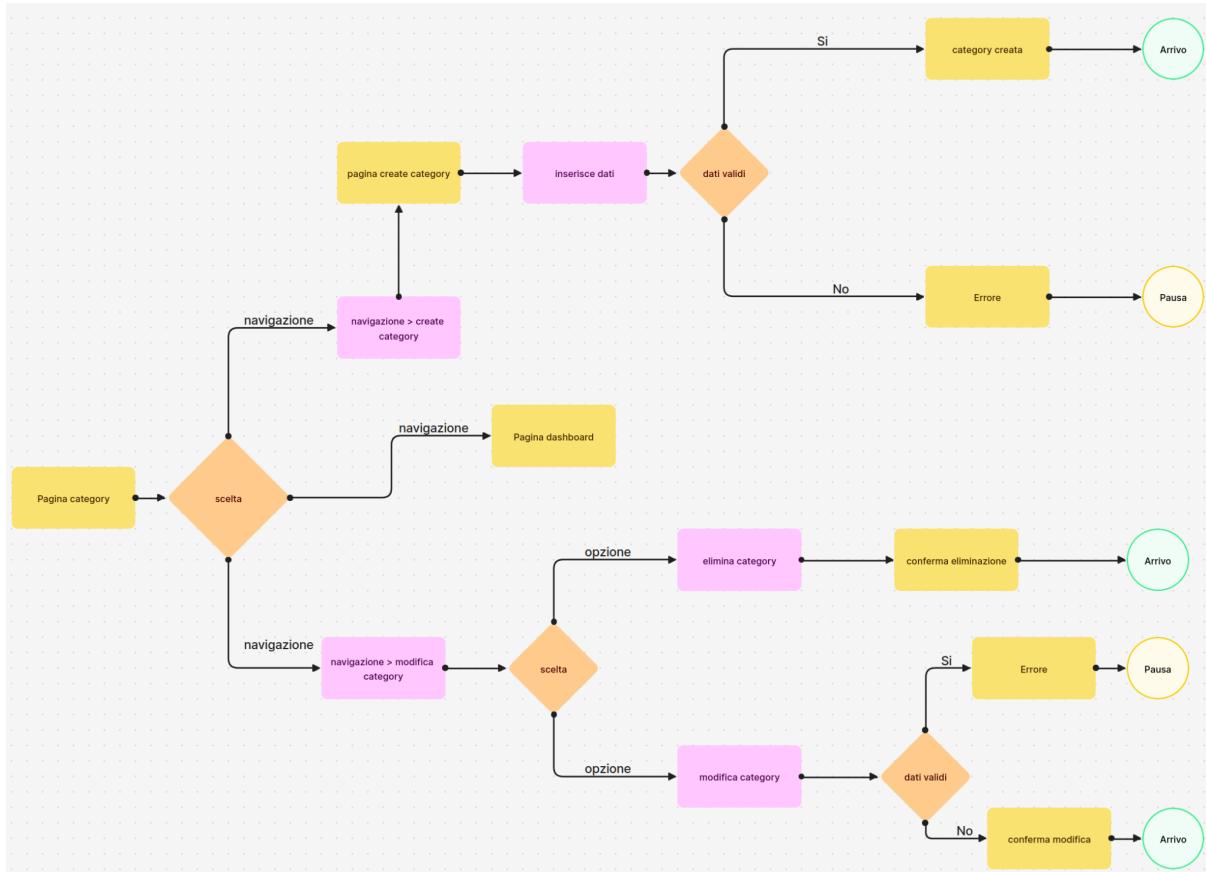
Flow diagram #1: Login e dashboard



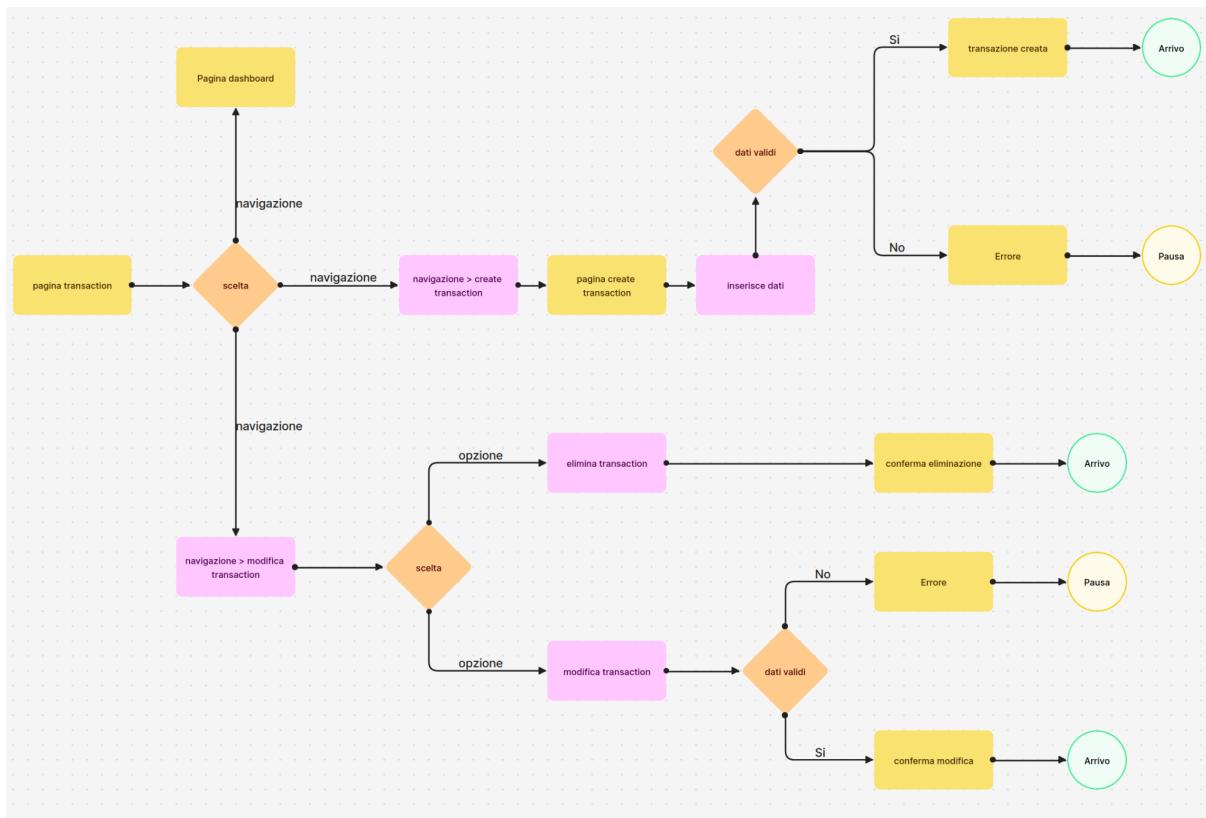
Flow diagram #2: Budget



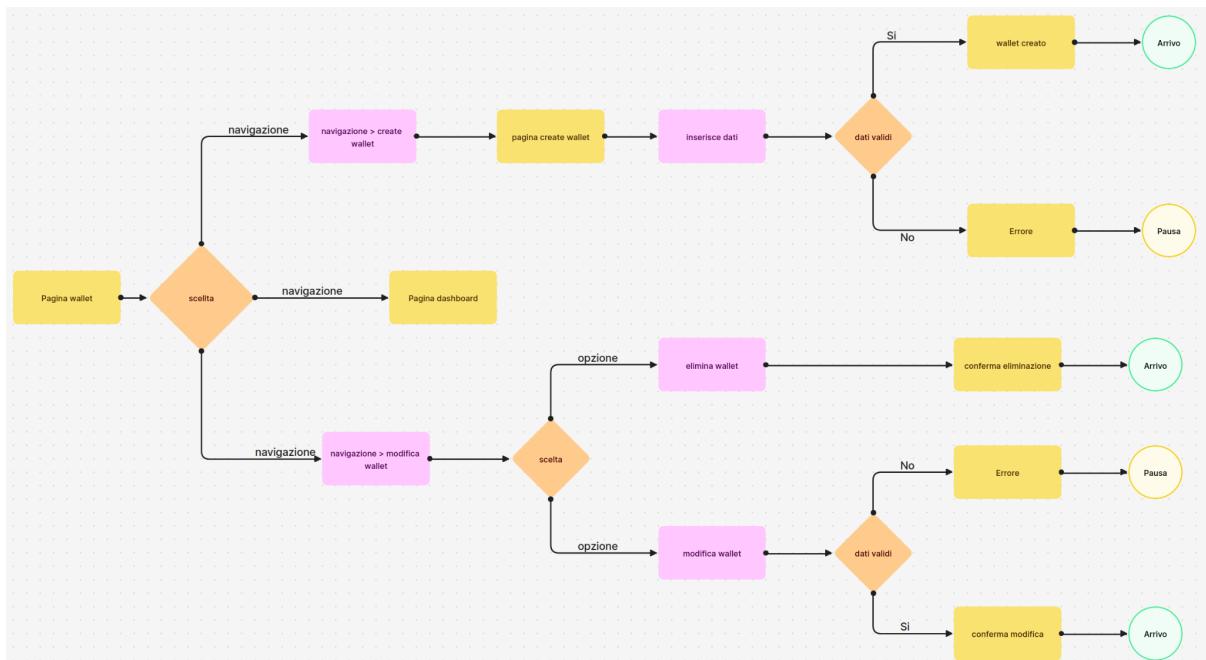
Flow diagram #3: Category



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 è stato presentato il corrispondente diagramma delle classi.

In questa sezione troveremo due diagrammi:

1. Uno deputato allestrazione delle risorse partendo dal *class diagram*.
2. Un altro apposito per rappresentare ciò che abbiamo sviluppato, ovvero le API che verranno effettivamente utilizzate nel sistema.

Estrazione risorse dal class diagram

Questo diagramma, come anticipato, si occupa di mostrare il processo di estrazione delle risorse a partire dal *class diagram* esposto nel [D3-Gog](#), per portarlo successivamente nella nostra applicazione attuale.

Il processo è iniziato con l'estrazione delle varie risorse. Una visione più completa dei vari modelli che abbiamo estratto viene riportata nella parte relativa ai **Modelli del Database** (vedi nelle sezioni successive del documento). Abbiamo individuato quindi i tipi di dati che applicavano meglio al nostro caso specifico e abbiamo selezionato gli attributi cruciali per lo sviluppo della versione della nostra applicazione.

Da qui, li abbiamo dunque trasposti su [MongoDB](#), il database che abbiamo deciso di utilizzare a partire dal *class diagram*.

Per questa specifica implementazione abbiamo deciso di omettere l'integrazione del chat bot con intelligenza artificiale in quanto questo avrebbe non solo reso molto più complesso lo sviluppo dell'applicazione, ma avrebbe implicato investimenti finanziari per effettuare le chiamate all'API di [ChatGPT](#).

Abbiamo inoltre messo da parte la possibilità di esportare i dati mediante file [CSV](#). Questo perché avrebbe richiesto di implementare query molto dispendiose e uno script di parsing piuttosto avanzato.

Abbiamo dunque posto il focus sugli altri modelli, e il loro adeguato sviluppo.

Per ogni risorsa e interazione, viene specificato il metodo HTTP per interagire con essi (GET, POST, PUT o DELETE, scelte in base al risultato finale che si vuole ottenere) e gli eventuali parametri necessari, siano essi presi dal link URL (GET e DELETE), oppure attraverso il body (POST e PUT).

Si noti che alcuni metodi utilizzano dati anche nella URL anche se sono richieste di tipo POST e PUT.

Ognuna di queste chiamate ha un effetto, che può manifestarsi sia sul backend che sul frontend.

In particolare, le richieste di tipo POST, PUT e DELETE hanno effetti sul backend, in quanto non ritornano informazioni che vengono trasmesse direttamente sullo schermo. Mentre le GET ritornano dei dati, in alcuni casi in grandi quantità, che vengono poi renderizzati e mostrati sul lato del frontend.

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

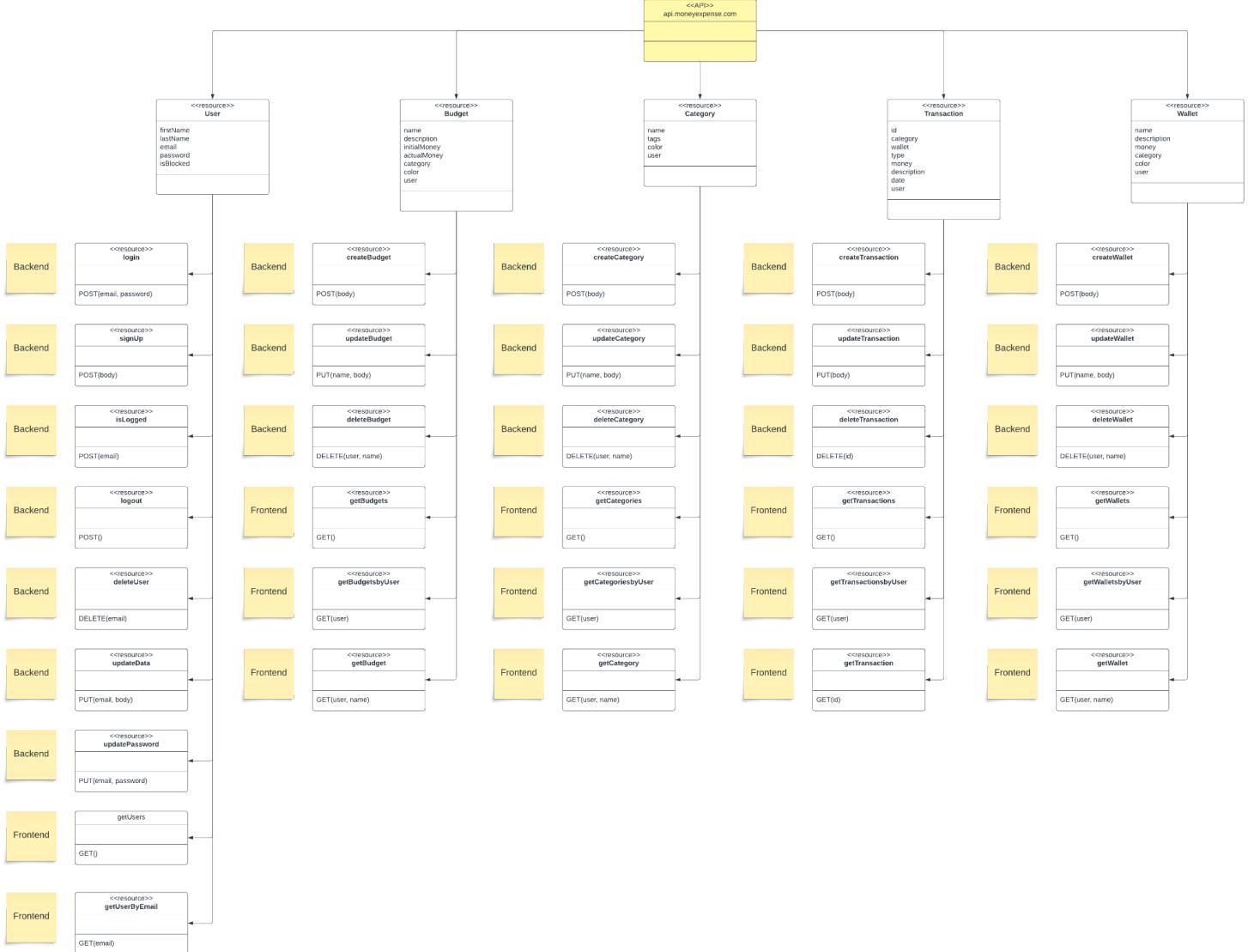


Diagramma delle risorse

Passiamo ora alla rappresentazione effettiva delle varie API presenti in questa versione di **Money Expense**. Lo vogliamo effettuare mediante un diagramma delle risorse, o diagramma delle API.

Poiché il nostro sistema è dotato di diversi modelli, i quali costituiscono il nucleo dell'applicazione, abbiamo scelto di strutturare il nostro diagramma in base a tali modelli. Questo approccio ci consente di ottenere una visione più dettagliata, suddividendo i vari concetti in modo ottimale.

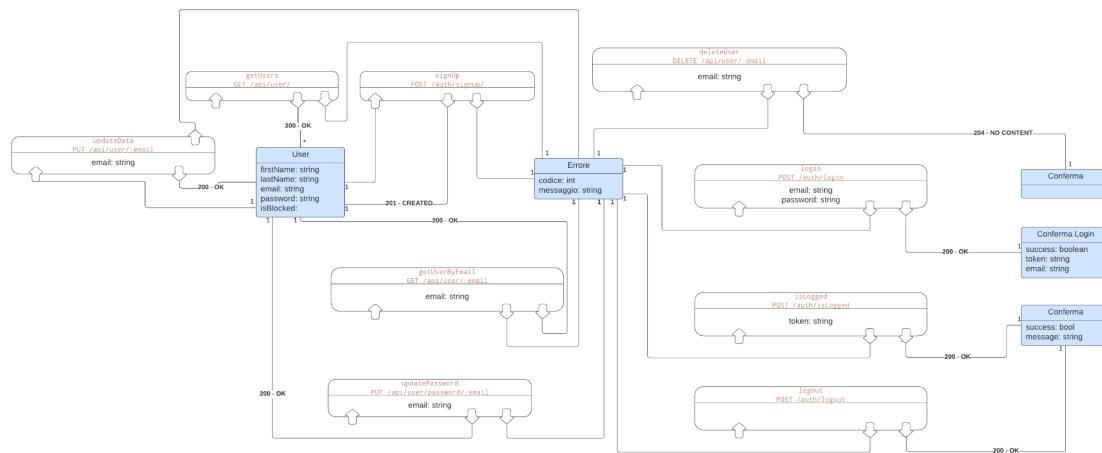
Come è tipico in questi diagrammi, ci sono in ogni caso specificati input ed output di ogni API che il sistema espone.

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*.

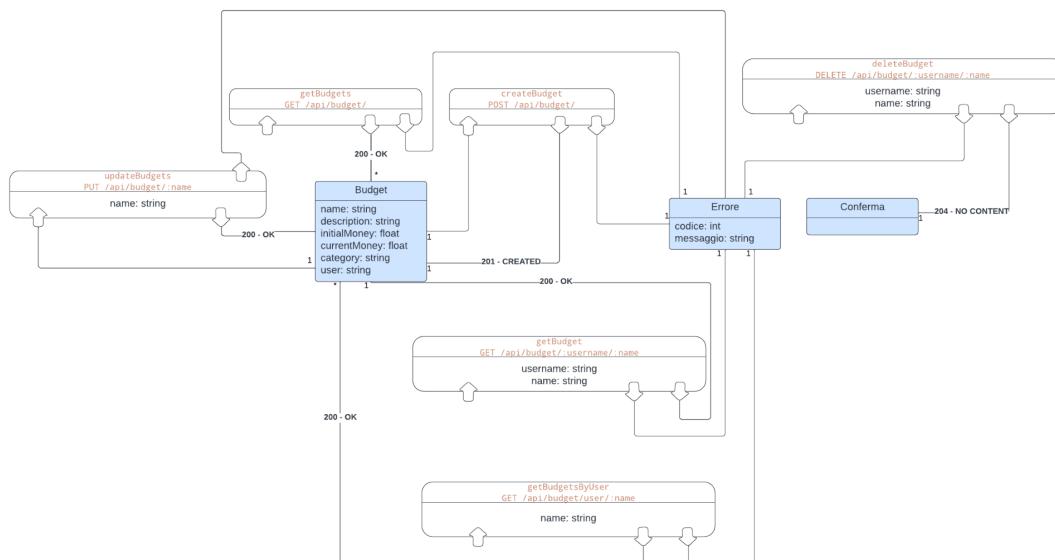
Nel caso invece di successo, se l'API non ritorna nulla, viene creata una classe **Conferma**, che sta solo a significare che non ci sono stati problemi, e che l'operazione richiesta è andata a buon fine.

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

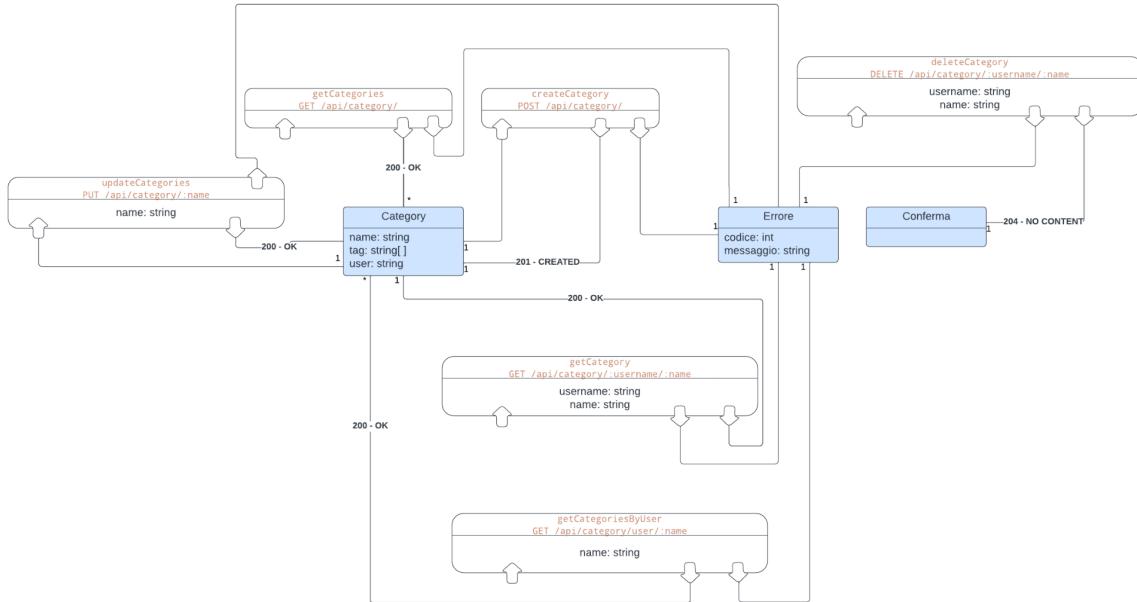
Resource Diagram 1



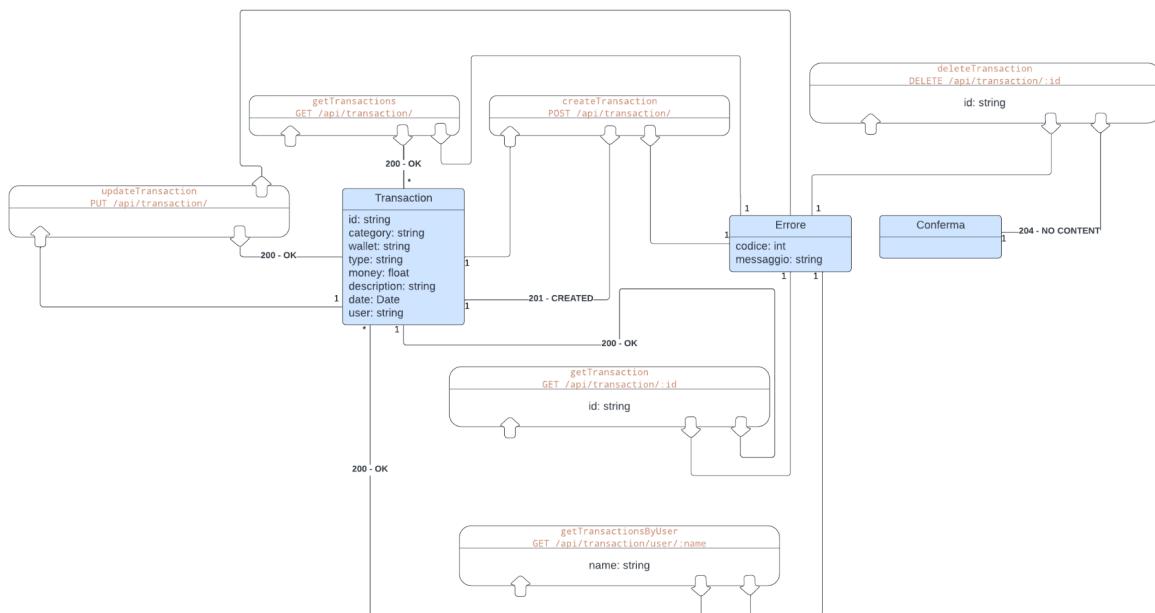
Resource Diagram 2



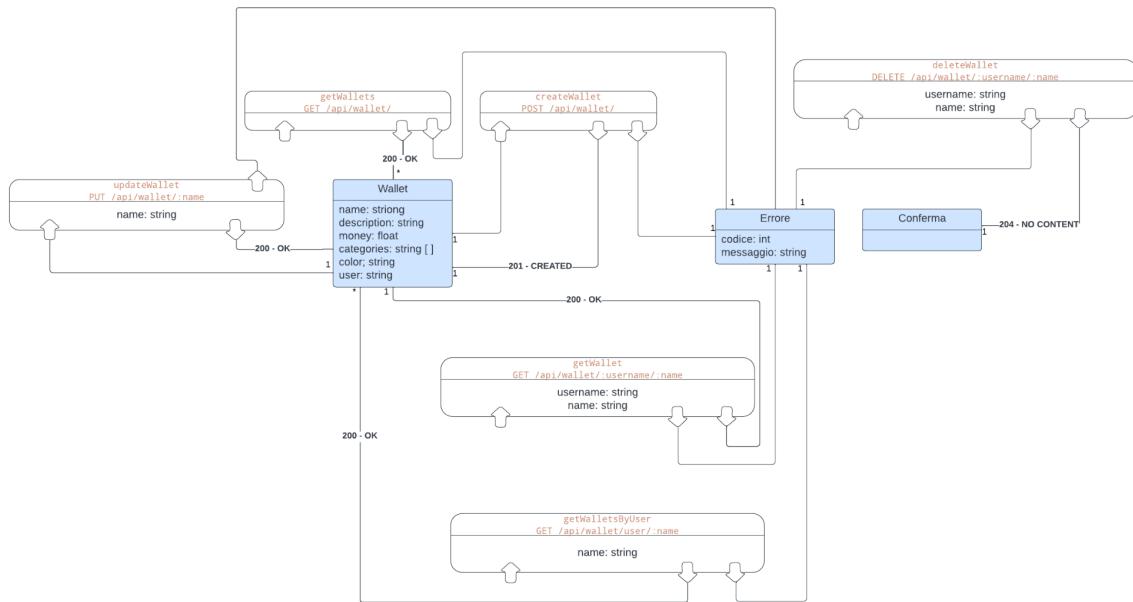
Resource Diagram 3



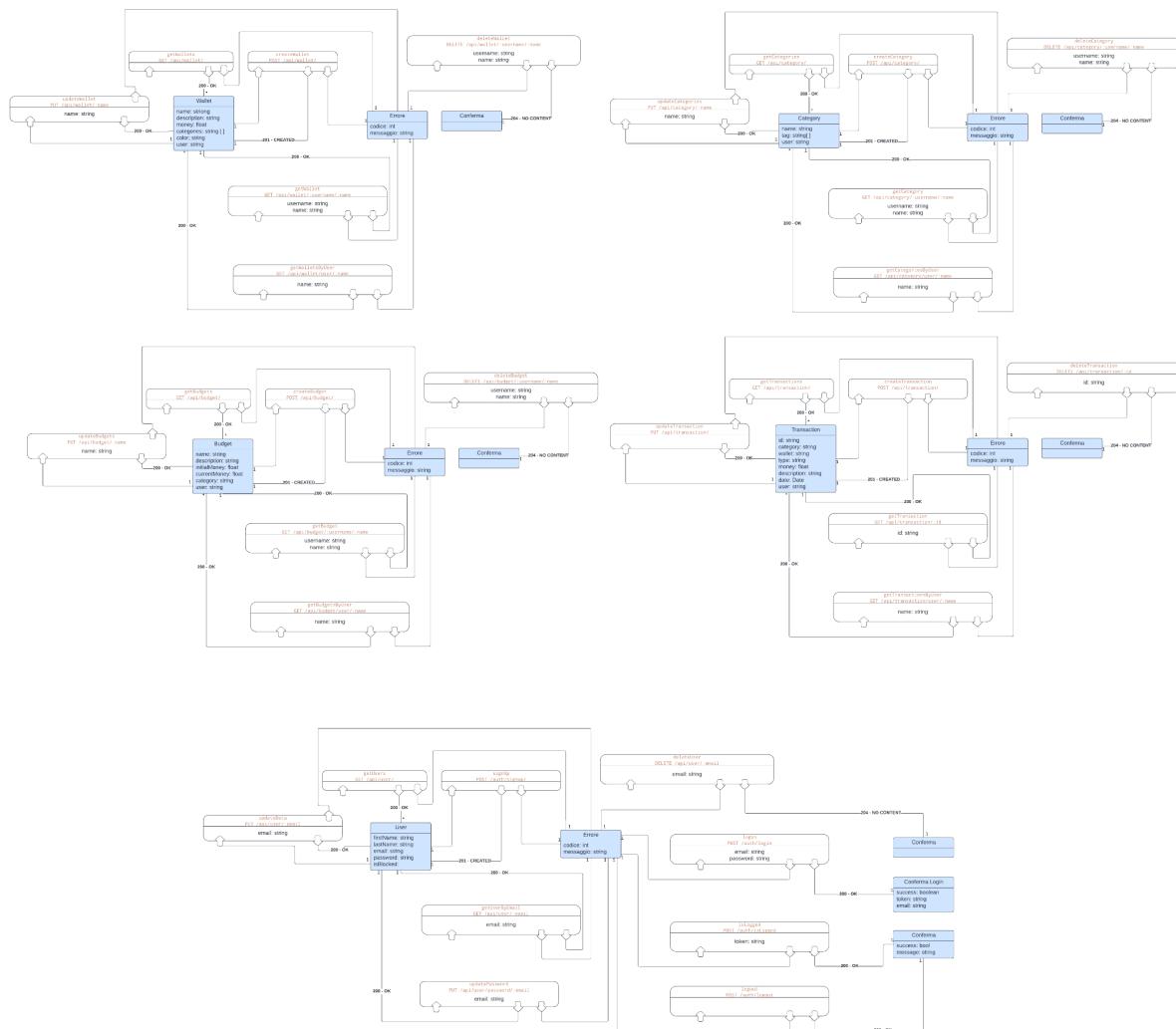
Resource Diagram 4



Resource Diagram 5



Resource Diagram Completo



Implementazione dell'applicazione

Per la creazione di **Money Expense**, abbiamo scelto di adottare *Javascript/TypeScript* come linguaggio principale sia per il frontend che per il backend.

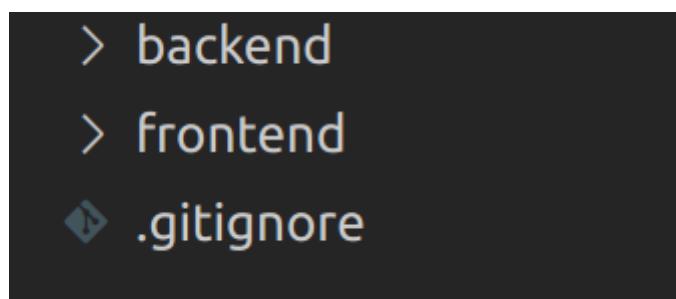
Nello specifico:

- Il frontend è stato sviluppato utilizzando la libreria UI ReactJS.
- Per il backend abbiamo selezionato NodeJS in combinazione con il framework ExpressJS.

Per quanto riguarda il database impiegato per archiviare i dati necessari, abbiamo optato per **MongoDB** in quanto si integra in modo ottimale con l'ecosistema dell'applicazione, beneficiando della completa suite di strumenti nativi sviluppati attorno ad esso.

Struttura del progetto

La struttura del progetto base è descritta dall'immagine sottostante.



Abbiamo optato per un approccio modulare cercando di scomporre il più possibile la logica del codice, formando quindi diversi moduli indipendenti, ma che fossero anche in grado di lavorare insieme nel modo più ottimale possibile.

Abbiamo quindi diviso lo sviluppo tra frontend e backend.

La directory base del progetto contiene quindi:

- Cartella **backend**, dove risiede tutta la logica del backend, le API e i modelli.
- Cartella **frontend**, dove risiede tutto il frontend dell'applicazione, le varie pagine, i react components, insieme a tutta la logica e gli asset grafici.
- file **.gitignore**, contiene i dati da non salvare in *git*, in quanto sarebbero superflui o con dati sensibili. Nel nostro caso, abbiamo escluso le directory *node_modules* e il file *.env*.

Più nello specifico, nella cartella backend troviamo tutta la logica del **backend**, scritto in Express, che permette il corretto funzionamento delle API. Al suo interno troviamo:

- Cartella **__test__**, dove sono contenuti tutti i file adibiti al testing della nostra API.
- Cartella **api**, che contiene un solo file Typescript, che serve a esporre l'applicazione al servizio di deployment.
- Cartella **controller** dove ci sono tutte le logiche delle API. Qui vengono descritte le modalità con cui le varie funzioni interagiscono con le richieste HTTP provenienti dall'esterno e modificano/interrogano il database a dovere. Questa è 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**, include tutte le strutture dati implementate per questa applicazione, le quali vengono successivamente memorizzate nel database da noi utilizzato, ossia **MongoDB**. Questo per dare uniformità ai dati della piattaforma.
- Cartella **public**, contiene solo un file *.gitkeep*. È consigliato per poter fare il deploy su Cyclic in maniera corretta. Serve per 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 utilità che vengono utilizzate più volte nell'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 di mettersi successivamente in ascolto per eventuali richieste dal frontend. Questo è il componente che avvia l'intero 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 NodeJS relative a ciò che abbiamo installato e agli script che utilizziamo.
- File **package-lock.json**, file di NodeJS con tutte le versioni di dipendenze correntemente installate.
- File **swagger.json**, file contenente tutta la documentazione integrale dell'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.

```

    < backend
      > __test__
      > api
      > controller
      > coverage
      > models
      > public
      > router
      > utils
      & .env
      TS index.ts
      JS jest.config.js
      {} package-lock.json
      {} package.json
      {} swagger.json
      TS tsconfig.json
  
```

Per quanto riguarda invece il codice del **frontend**, anche questo viene inserito in un'apposita cartella. Questa contiene tutti i file tipici di un'applicazione in ReactJS, con qualche libreria esterna utilizzata. Molti file presenti in questa directory sono simili a quelli visti nel backend in quanto entrambi utilizzano *Javascript/TypeScript*. Questi file comuni sono:

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

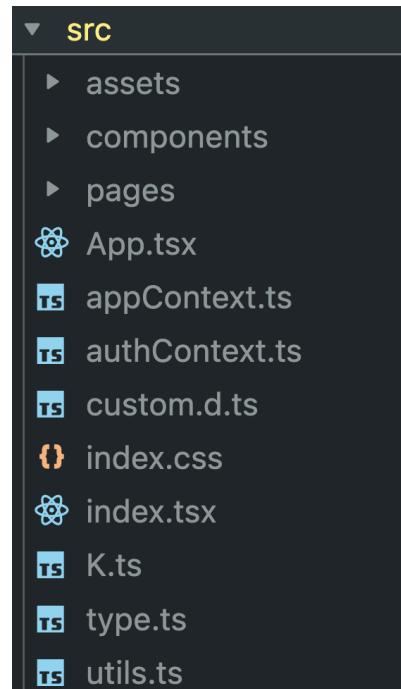
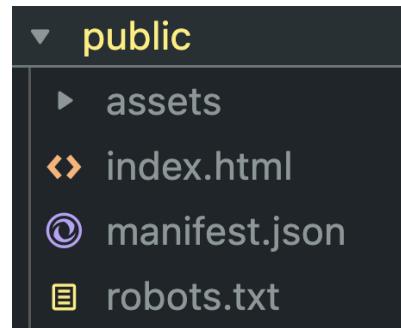
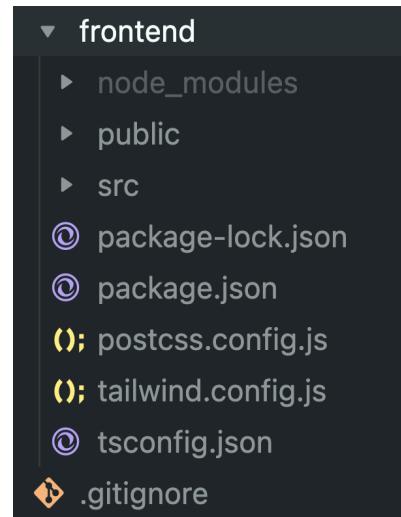
Dopo di che, in questo caso troviamo i file *tailwind.config.js* e *postcss.config.js*, che fungono da file di configurazione per la libreria TailwindCSS, che abbiamo integrato nel progetto per rendere più gradevole e sistematico lo sviluppo dell'interfaccia web.

Nella stessa directory, ci sono le due cartelle tipiche di React: **public** e **src**.

La directory **public** contiene i file pubblici, tra cui i vari asset come il logo e le immagini, il file HTML di base, generato durante *npm run build*, e alcuni file di servizio come *manifest.json* e *robots.txt*, generati automaticamente da ReactJS alla creazione del progetto.

La cartella **src** contiene soprattutto i componenti UI del frontend. Ci sono alcuni file importanti da menzionare:

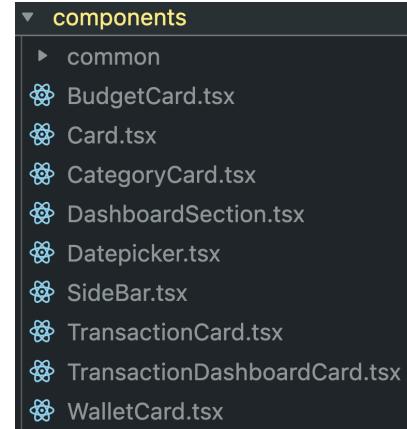
- ***index.tsx***, contiene il codice di startup dell'applicazione.
- ***App.tsx***, è dedicata al routing nell'applicazione e alla definizione degli endpoint della web app.
- ***authContext.ts***, riguarda la dichiarazione e definizione di un ***globalContext*** di React che permette di accedere facilmente ai dati dell'utente autenticato ed eventualmente aggiornarli.
- ***appContext.ts***, include la dichiarazione e definizione di un ***globalContext*** di React per facilitare il recupero e laggiornamento di dati (wallets, transactions, categories, budgets...) frequentemente utilizzati dall'applicazione.



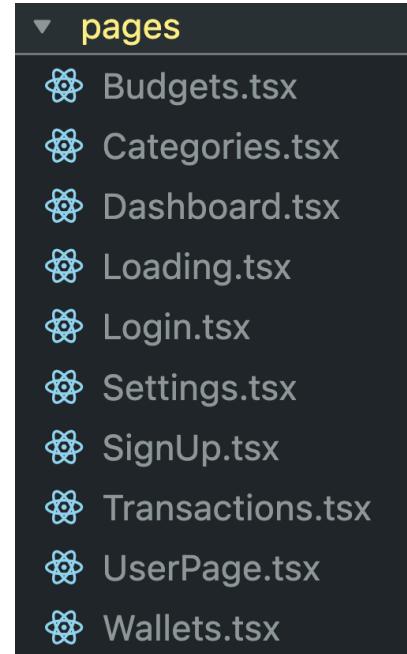
- **K.ts**, contiene un oggetto Typescript con alcune definizioni statiche.
- **type.ts**, include i tipi di dati utilizzati dal programma, sincronizzati con i modelli descritti nel backend.
- **utils.ts**, qui troviamo alcune utility functions frequentemente utilizzate nel progetto.

Inoltre, è possibile osservare 2 cartelle importanti:

- **components**, contiene tutte le componenti di ReactJS con estensione .tsx. All'interno è presente la directory **common** che elenca tutte le componenti frequentemente utilizzate nell'arco del programma. Le rimanenti componenti non presenti in common sono menzionate una o al massimo due volte nel codice del frontend.



- **pages**, qui troviamo tutte le pagine, o viste, descritte in maggiore dettaglio nelle sezioni successive. Una cosa importante da notare è il file **UserPage.tsx** che funge da wrapper per tutte le rimanenti pagine eccetto *Loading.tsx*, *SignUp.tsx* e *Login.tsx*.



Dipendenze del progetto

In modo da garantire il corretto funzionamento del progetto, abbiamo utilizzato varie dipendenze e librerie esterne, che ci hanno permesso l'astrazione di molto codice e di porre il focus effettivo su ciò che dovevamo implementare.

Come in tutti i progetti NodeJS, tutte le dipendenze finiscono nei file `package.json`, come spiegato precedentemente. Di seguito elenchiamo le dipendenze di cui abbiamo fatto uso e il loro scopo.

- **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 una facilitata creazione di API, mediante la stesura 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 i token di 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 effettuare il testing. *jest* è la suite javascript per il testing, mentre *supertest* permette di compiere chiamate di test per le API REST, dando quindi la possibilità di testare, emulando un ambiente frontend.

Inoltre, all'interno del nostro file `package.json` includiamo i seguenti script, utili per la fase di development.

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

Questi ci hanno permesso rispettivamente di:

- **build**, compilare Typescript.
- **start**, lanciare il server compilato.
- **dev**, lanciare il server in modalità development.
- **test**, effettuare i test di jest in locale.

Modelli nel database

Per poter modellare adeguatamente i dati necessari alla nostra applicazione, si è proceduto con la creazione di modelli nel database, partendo sempre dal nostro diagramma delle classi.

Ognuno di questi modelli ci permette di essere consistenti e di strutturare adeguatamente l'utilizzo della nostra applicazione.

Abbiamo individuato cinque modelli principali, che ora andiamo a spiegare in 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 e per favorire una maggiore personalizzazione della UI.

Sono stati inclusi i campi *email* e *password*, che permettono all'utente di eseguire l'operazione di login in piattaforma. L'*email* deve essere univoca, non sono ammessi utenti con indirizzi mail identici.

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

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: ObjectId('658ca64473fabad3c9047bba')  
firstName: "Mario"  
lastName: "Rossi"  
email: "mario.rossi@gmail.com"  
password: "abcABC1!"  
isBlocked: false  
__v: 0
```

Modello Budget

Per rappresentare invece i vari budget che l'utente crea in piattaforma, è 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},
  user: {type:String, required:true},
});
```

Il budget è caratterizzato da un attributo `_id` che lo identifica univocamente nel database, `name`, titolo del budget, e da una `description` testuale.

Troviamo i due attributi numerici, `initialMoney` e `actualMoney`, indicanti il budget di partenza e quello attuale.

Per classificare meglio ogni budget, abbiamo messo un campo `category`, per comunicare a che categoria questo appartiene, e infine il campo `user`, che mostra l'utente a cui tale budget appartiene.

Di seguito un esempio di un possibile record del modello *Budget*:

```
_id: ObjectId('659149abeb52cf6d6bcaa2b9')
name: "Spesa"
description: "Budget per tracciare la spesa ai supermercati"
initialMoney: 100
actualMoney: 90
category: "Spesa"
user: "mario.rossi@gmail.com"
__v: 0
```

Modello Category

Per poter compartimentare correttamente le informazioni derivanti da ogni categoria che un utente può 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]},
  user: {type:String, required:true},
});
```

Questo modello salva un *name* o nome che caratterizza la categoria creata dall'utente sulla piattaforma.

Dopo di che, una serie di *tags* o reference che aiutano a fornire una descrizione della categoria.

Infine troviamo *user*, ossia l'utente della categoria in questione, insieme all'*_id* univoco che viene creato in MongoDB, per garantire unicità dei record.

Di seguito un esempio di un possibile record del modello *Category*:

```
_id: ObjectId('658ca68427108f1f0b290b84')
name: "Spesa"
  ▼ tags: Array (2)
    0: "spesa"
    1: "supermercato"
user: "mario.rossi@gmail.com"
__v: 0
```

Modello Transaction

Per salvare correttamente e in maniera consistente tutte le transazioni che un utente registra 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 identificativo universale delle transazioni registrate.

Dopo di che, per caratterizzare specificamente la tipologia di transazione, abbiamo inserito il parametro `type`, che è una stringa che può assumere valore “expense” o “income”.

Il campo `money` mostra l'ammontare della transazione effettuata.

Per chiedere di che categoria fa parte la transazione, abbiamo messo un apposito campo `category`, che ci permetterà poi di legarci ai vari budget.

Inoltre, abbiamo aggiunto il campo `wallet` che ci permette di specificare la wallet da cui verrà detratta la somma.

Ovviamente, ogni transazione avviene in una data, e quindi abbiamo provveduto ad aggiungere anche il campo `date`.

Per caratterizzare ulteriormente una transazione, abbiamo messo la possibilità all'utente di scrivere una `description` 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: ObjectId('658ca6c027108f1f0b290b8e')
category: "Salario"
wallet: "Conto_corrente_IT"
type: "income"
money: 100
description: "Ripetizioni"
date: 2023-12-27T22:35:44.772+00:00
user: "mario.rossi@gmail.com"
__v: 0
```

Modello Wallet

Per caratterizzare tutti i wallet, quindi i fondi di liquidità, 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},
  color: {type:String},
  user: {type:String, required:true},
});
```

Come sempre troviamo il campo `_id` che funge da identificativo per ciascun record nel database.

Il modello presenta i campi `name`, `description` e `color`, che rappresentano rispettivamente il nome del fondo, una descrizione esaustiva creata dall'utente e un colore per rendere la UI piacevole da utilizzare.

Per rappresentare dati di utilità, ci sono i campi `money`, indicante il bilancio presente sul fondo, ed infine `user`, per comunicare l'utente proprietario della wallet.

Di seguito un esempio di un possibile record del modello Wallet:

```
_id: ObjectId('658ca67a27108f1f0b290b81')
name: "Conto_corrente_IT"
description: "Conto corrente italiano utilizzato per spese universitarie"
money: 10010
user: "mario.rossi@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 necessità.

La descrizione si limiterà ad una spiegazione testuale, in quanto lo sviluppo effettivo di codice sarebbe troppo prolioso. Il codice è disponibile nella cartella

/backend/controller della nostra repository di GitHub, nel caso si volesse consultare.

API per modello User

In questo progetto, un utente può 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.

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 richieda mai l'utilizzo di una tale funzionalità. 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 è possibile farlo, in quanto l'email è un campo *unique*, e quindi non esistono più utenti con la stessa email. Questo metodo ci è utile in fase di richiesta delle informazioni dell'utente.

Per garantire il funzionamento di tale API, viene sfruttato 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 metodo dell'API ci permette di modificare le informazioni relative ad uno specifico utente, siano queste nome, cognome, email o password. Permette quindi una più 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 funzionalità 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 necessità 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.

Il metodo in questione prende dalla URL l'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 complessità e robustezza che la piattaforma si pone (gestiti da una apposita funzione, messa nella repository *utils*). Verificati tali requisiti, si passerà all'effettiva modifica della password dell'utente, che avrà 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 funzionalità, 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 dalla URL l'email dell'utente specifico che si desidera eliminare. Si passa dunque alla ricerca e all'eliminazione dell'utente. MongoDB permette di effettuare 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 funzionalità 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 un'email e una password per verificare la sua identità. 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 dà validità di 12 ore. Questo viene poi ritornato nella risposta, così che l'utente possa richiedere dunque le sue risorse in piattaforma.

Logout utente

Questa API vuole fare in modo che la sessione dell'utente termini.

Per terminare una sessione, è necessario che una sessione venga precedentemente inizializzata, quindi questo endpoint è richiamabile 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 è stato revocato, chiudendo quindi con successo la sessione di navigazione.

Controllo se utente è loggato (isLogged)

Questo metodo specifico della API ci permette di capire se un utente è loggato oppure no in piattaforma. È semplicissimo. Si occupa di leggere il token che viene allegato negli header della richiesta HTTP dell'utente che sta usando la piattaforma.

Se il token non è presente oppure il token è scaduto, allora l'API risponderà che l'utente non è loggato.

Se invece il token è presente ed è ancora valido, si avrà risposta affermativa.

Questo è 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.

Il signUp è posto sotto autenticazione e non sotto utente, in quanto tutte le API utente sono controllate da un middleware che verifica se l'utente è loggato o meno.

In questo caso, un utente non può essere loggato se non ha un proprio profilo. Perciò 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 dell'operazione, in caso di successo, viene ritornato il nuovo utente creato con i suoi dati.

I metodi utilizzati sono **findOne()** per assicurarsi che l'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, può 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 ciò, MongoDB espone la funzionalità **save()** che salva un oggetto che aderisce ad un modello specifico creato nella sezione model (vedi sezione Modelli nel Database).

Viene verificato quindi, mediante la funzione **findOne()**, che tale utente non abbia già creato un budget con lo stesso nome. Altrimenti si creerebbe confusione.

Nonostante ciò, è 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 è 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, è 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 desideriamo determinare tutti i budget di un dato utente utilizziamo questa API. Prende dalla URL l'email dell'utente che vuole svolgere tale richiesta, e in base a quest'ultima si spinge alla ricerca delle varie risorse.

Per cercare tutti i budget di un dato utente, si utilizza la funzione [find\(\)](#) di MongoDB mettendo come filtro l'email ricevuta.

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

Trova un budget

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

Per poter capire che budget specifico, si utilizza l'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.

Quest'ultima sarà anche la risposta alla richiesta.

Modifica budget

Questa API si pone come obiettivo la modifica di uno specifico budget. Nella URL gli passiamo solamente il nome. Questo in quanto la proprietà di un budget non cambia, e quindi possiamo determinare l'utente dal body, con tutti i nuovi dati che l'utente vuole eventualmente modificare.

MongoDB ci permette di utilizzare il metodo [findOneAndUpdate\(\)](#), che effettua una modifica del record così ottenuto.

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 occuperà di rimuoverlo in maniera automatica.

Ritorniamo un messaggio vuoto, che sta a significare che l'operazione è 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 è quello di creare una nuova category in piattaforma, di modo che un utente possa usufruirne.

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

[MongoDB](#) aiuta nella creazione mediante 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 categorie presenti in piattaforma. È solo di supporto in quanto un utente non può vedere le categorie create da altri utenti.

Per fare in modo 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 categorie di uno specifico user, questo endpoint è quello che viene invocato.

L'utente viene passato come parametro della 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 ricevuta, e ritorniamo come risposta tutte le istanze trovate nella query.

Trova una category

Lo scopo di tale API è quello di trovare una singola specifica risorsa di tipo category nel database. Viene fatto questo cercando mediante l'email dell'utente e il nome della categoria. Questo perché 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 è 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, leggiamo quindi il body per estrarre le informazioni necessarie e modifichiamo tale record.

Determiniamo il record e lo 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 è quello di eliminare una specifica category dal database.

La category in questione viene univocamente identificata dal nome della stessa, e dall'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 è un codice 204, quindi senza contenuto.

API per modello *Wallet*

Crea wallet

Quando un utente desidera creare un wallet, può 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 ciò, [MongoDB](#) espone la funzionalità [`save\(\)`](#).

Viene fatto però prima un controllo, mediante la funzione [`findOne\(\)`](#), che tale utente non abbia già creato un wallet con lo stesso nome, altrimenti il tutto sarebbe confusionario. Nonostante ciò è possibile che due utenti diversi abbiano un wallet con il medesimo nome. Se l'operazione ha avuto successo, l'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 è solo di supporto agli sviluppatori, e non viene mai direttamente chiamata dal sistema di frontend..

Per fare funzionare correttamente questo endpoint, è 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

Questa API viene utilizzata quando desideriamo determinare tutti i wallet di un dato utente.

Prende dall'URL l'email dell'utente che vuole effettuare 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 l'email che è stata trovata nella stringa URL.

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

Trova un wallet

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

Per poter capire che risorsa specifica, si utilizza l'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 sarà la risposta alla richiesta che abbiamo ricevuto.

Modifica wallet

Questa API si pone come obiettivo la modifica di un wallet. Nella 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 ciò, evitando dati inutili in piattaforma.

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

API per modello *Transaction*

Crea transaction

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

Per fare ciò 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 mediante il metodo [**save\(\)**](#), che salva la nuova risorsa, che verrà poi ritornata come risposta all'utente finale.

Tutte transaction

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

Per fare sì 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 determinare tutte le transaction di un dato utente, viene usato il seguente 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 è quello di trovare una singola transaction nel database. Viene effettuato cercando mediante l'`_id` univoco della specifica transazione. Questo parametro è 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 è 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.
Determiniamo il record e lo aggiorniamo sfruttando la funzione [`findOneAndUpdate\(\)`](#) di MongoDB, che esegue l'aggiornamento della risorsa. Ritorniamo alla fine, l'istanza modificata.

Elimina transaction

Lo scopo di questa API è 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 è 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 è attualmente lo standard de facto per ogni applicazione web.

In questo modo, la documentazione è visibile a tutti quanti sia lato codice che lato deployment.

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

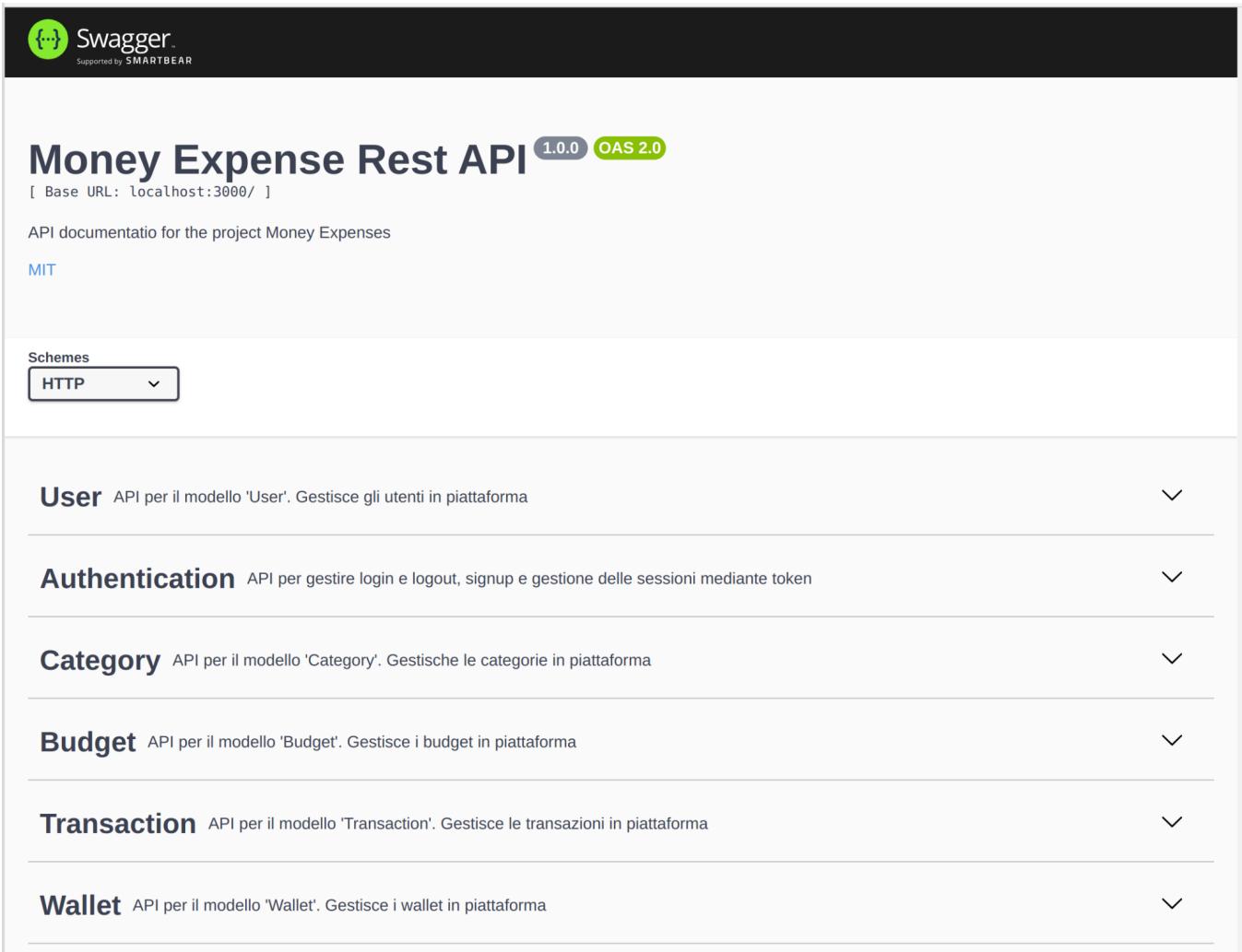
<http://localhost:3000/api-docs>

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

Se invece si volesse vedere le API correntemente deployate, si può visitare il corrente link, mediante chiamata HTTPS:

<https://kind-pear-starfish-belt.cyclic.app/api-docs/>

Riportiamo di seguito un'immagine di come appare l'immagine della documentazione:



The screenshot shows the Swagger UI interface for the Money Expense Rest API. At the top, it displays the title "Money Expense Rest API" with version 1.0.0 and OAS 2.0 compliance. It also shows the base URL as "localhost:3000/". Below the title, there's a brief description: "API documentatio for the project Money Expenses" and a license link to "MIT". A dropdown menu for "Schemes" is open, showing "HTTP" selected. The main content area lists several API endpoints categorized under "User", "Authentication", "Category", "Budget", "Transaction", and "Wallet". Each category has a descriptive text and a collapse/expand arrow icon. The "User" section says "API per il modello 'User'. Gestisce gli utenti in piattaforma". The "Authentication" section says "API per gestire login e logout, signup e gestione delle sessioni mediante token". The "Category" section says "API per il modello 'Category'. Gestisce le categorie in piattaforma". The "Budget" section says "API per il modello 'Budget'. Gestisce i budget in piattaforma". The "Transaction" section says "API per il modello 'Transaction'. Gestisce le transazioni in piattaforma". The "Wallet" section says "API per il modello 'Wallet'. Gestisce i wallet in piattaforma".

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

Volendo entrare nello specifico di una data API di cui si è più interessati, si possono aprire i vari tag facendo click su quello di proprio interesse. A questo punto, il menù a tendina si aprirà e verranno mostrati tutti gli endpoint sviluppati e pubblicati dall'applicazione in questione. Ogni endpoint è 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 funzionalità è 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, nell'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 l'API può 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 frontend che abbiamo sviluppato.

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

- Signup
- Login
- Dashboard
- Wallets
- Transactions
- Categories
- Budgets
- Settings

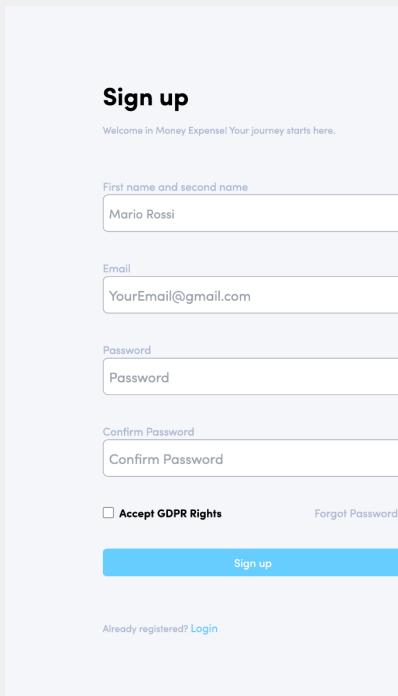
Signup

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.



Sign up

Welcome in Money Expense! Your journey starts here.

First name and second name
Mario Rossi

Email
YourEmail@gmail.com

Password
Password

Confirm Password
Confirm Password

Accept GDPR Rights [Forgot Password?](#)

[Sign up](#)

Already registered? [Login](#)



YOUR NEW JOURNEY

mx money expense

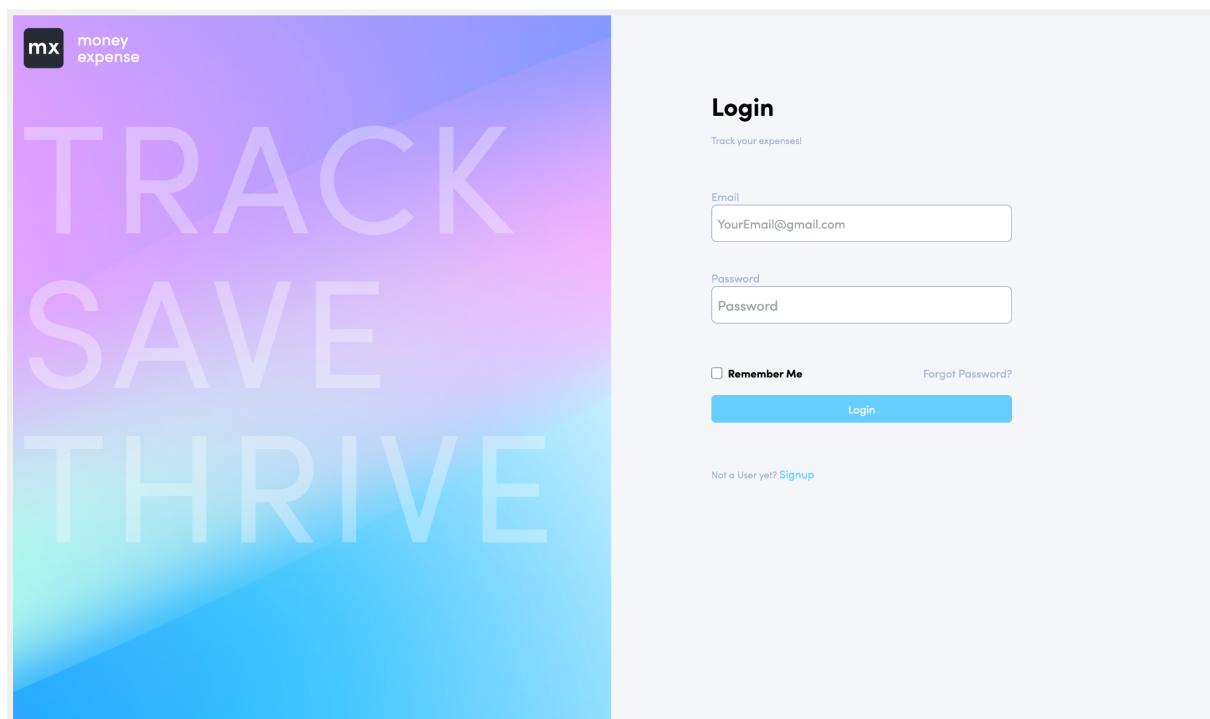
Login

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

Si richiede di inserire email e password.

In caso l'utente volesse 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](#).



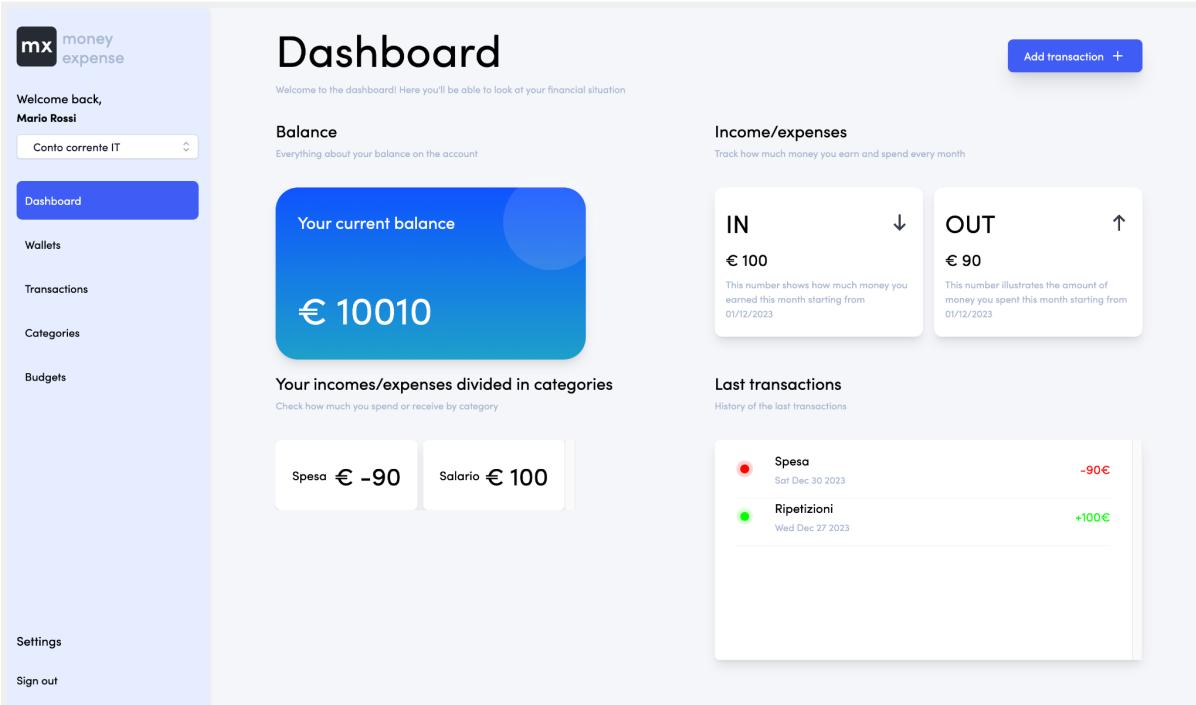
Dashboard

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](#).
- 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>](#).
- Elenicare l'insieme delle ultime transazioni avvenute nella sezione [Last transactions](#).



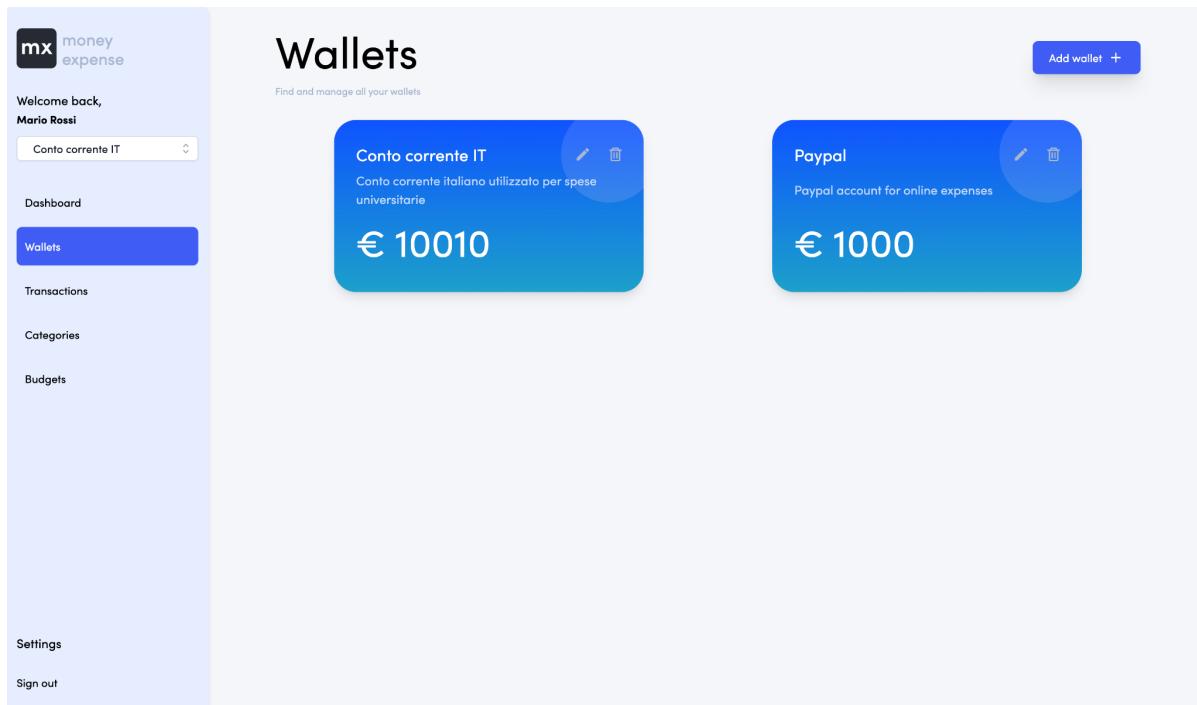
The screenshot shows the 'Dashboard' page of the 'money expense' application. On the left, the sidebar includes 'Welcome back, Mario Rossi', a dropdown for 'Conto corrente IT', and links for 'Dashboard' (which is highlighted in blue), 'Wallets', 'Transactions', 'Categories', 'Budgets', 'Settings', and 'Sign out'. The main content area has a header 'Dashboard' with a sub-instruction 'Welcome to the dashboard! Here you'll be able to look at your financial situation'. It features several sections: 'Balance' (current balance €10010), 'Income/expenses' (IN €100 and OUT €90), 'Last transactions' (Spesa -90€ on Sat Dec 30 2023 and Ripizioni +100€ on Wed Dec 27 2023), and 'Your incomes/expenses divided in categories' (Spesa € -90 and Salario € 100).

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 **Add wallet** 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 da aggiungere. 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 7 colonne: 5 descrittive (Type, Description, Category, Date, Amount) e 2 funzionali (Edit, Delete).

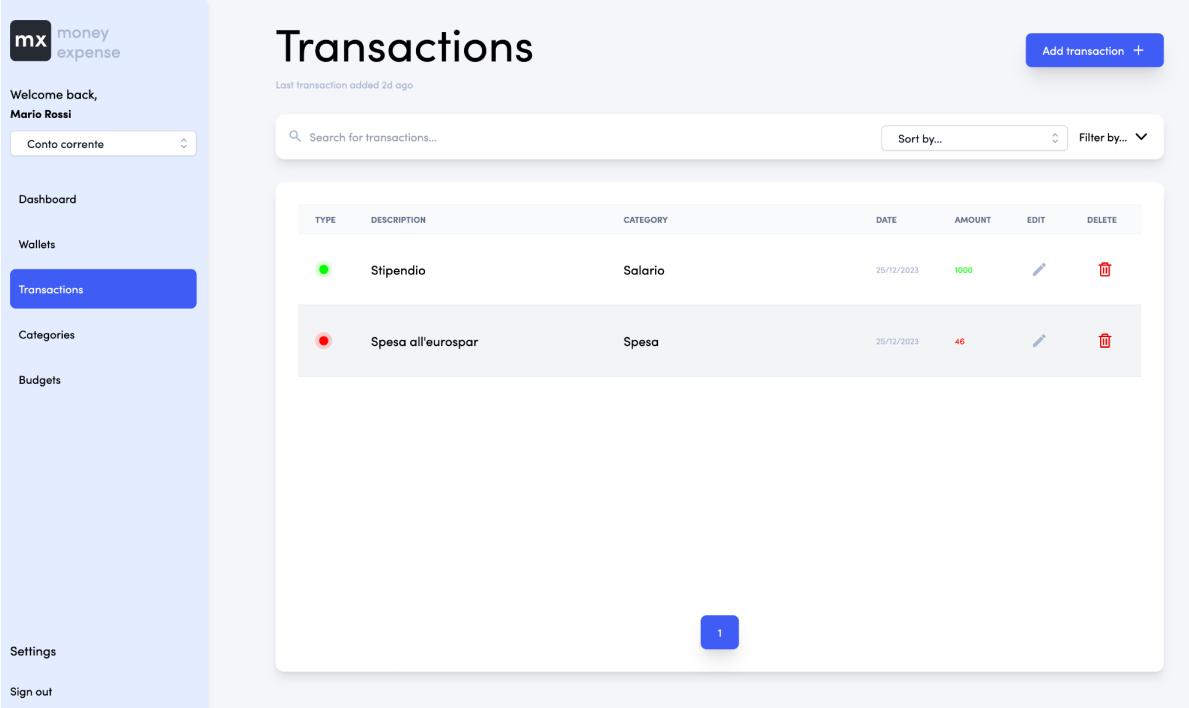
Le colonne descrittive hanno il compito di visualizzare l'informazione della singola transazione:

- [Type](#), indicato da 2 cerchi concentrici di 2 possibili colori, rosso e verde, rispettivamente per expense e income.
- [Description](#), causale o descrizione del motivo dell'avvenuta transazione.
- [Category](#), indica la categoria della transazione avvenuta.
- [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. Cliccando il bottone [Save](#) si salvano le modifiche effettuate.
- [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. Cliccando il bottone [Save](#) si procede con l'eliminazione della transazione.

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



The screenshot shows the mx money expense application interface. On the left, a sidebar menu includes: Dashboard, Wallets, **Transactions** (selected), Categories, Budgets, Settings, and Sign out. The main content area is titled "Transactions" and displays two entries:

TYPE	DESCRIPTION	CATEGORY	DATE	AMOUNT	EDIT	DELETE
●	Stipendio	Salario	25/12/2023	1000		
●	Spesa all'eurospar	Spesa	25/12/2023	-46		

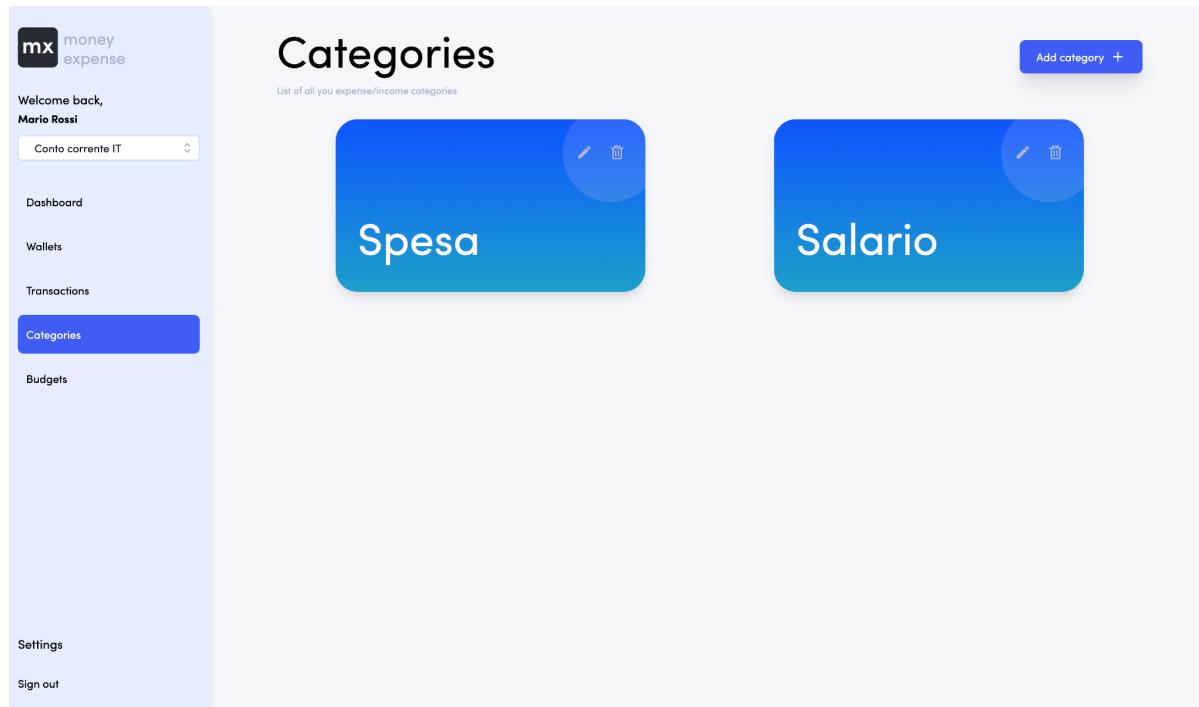
A search bar at the top right says "Search for transactions...". Below it are "Sort by..." and "Filter by..." dropdowns. A blue button at the bottom right of the table area says "1".

Categories

La pagina delle categorie offre la possibilità di visualizzare tutte le categorie create.

In questa schermata sono presenti le seguenti componenti:

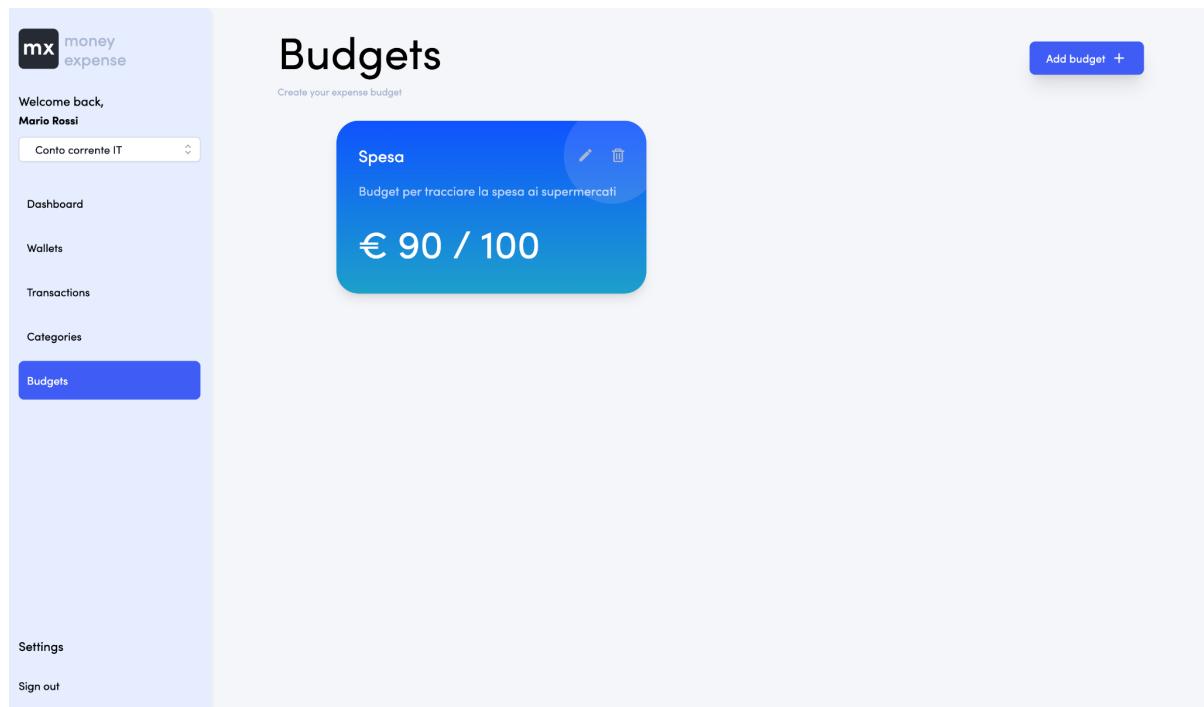
- Il bottone [Add Category](#), cliccato il quale si apre un modal popup che richiede di inserire i dati relativi alla categoria da aggiungere.
- Un elenco a griglia che mostra le categorie esistenti con la somma spesa in quella specifica categoria, premendo su una delle card è possibile modificarne i dati o eliminarli.



Budgets

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:

- Il bottone [Add Budget](#), cliccato il quale si apre un modal popup che richiede di inserire i dati relativi al budget da aggiungere.
- 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.



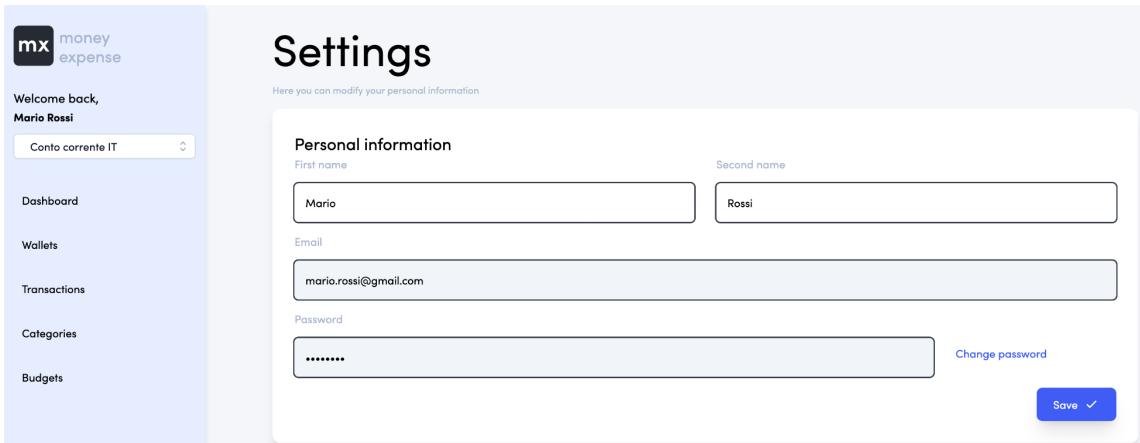
Settings

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

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

- **First name** e **Second name** modificabili.
- **Email** non modificabile.
- **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**.



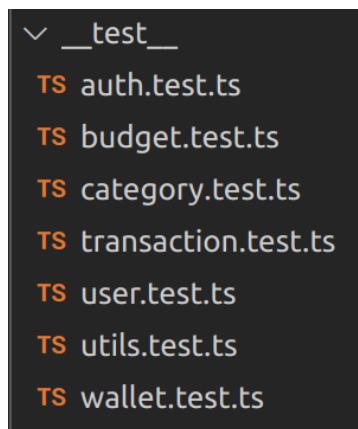
The screenshot shows the 'Personal information' settings page. On the left, there's a sidebar with the 'mx money expense' logo, a 'Welcome back, Mario Rossi' message, and navigation links: 'Conto corrente IT', 'Dashboard', 'Wallets', 'Transactions', 'Categories', 'Budgets', 'Settings' (which is highlighted), and 'Sign out'. The main content area has a heading 'Settings' and a sub-instruction 'Here you can modify your personal information'. It contains a 'Personal information' section with four fields: 'First name' (input: 'Mario'), 'Second name' (input: 'Rossi'), 'Email' (input: 'mario.rossi@gmail.com'), and 'Password' (input: '*****'). To the right of the password field is a 'Change password' link. At the bottom right is a blue 'Save' button with a checkmark icon.

Testing

Vediamo ora come è stato effettuato il testing del progetto.

Per poter testare le nostre funzionalità, 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 è stato messo nella cartella `__test__`, come propongono le linee guida ufficiali presenti nella documentazione del package. Di seguito un'immagine della repository con i vari file.



Abbiamo sviluppato specifici file dedicati all'esecuzione dei test. Ciascun file racchiude tutte le funzioni destinate a testare un modulo specifico della nostra piattaforma, che possa trattarsi di un'API o delle funzioni utility create per migliorare la scrittura del codice. Abbiamo adottato questa suddivisione al fine di evitare la creazione di file di test eccessivamente estesi, consentendo una chiara separazione tra i vari ambiti 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 = "";

> beforeEach(async () => {
| })
> afterEach(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 è 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 funzionalità. Nello specifico, importiamo sempre *mongoose* per poter utilizzare il database, *supertest* per poter testare la piattaforma e *jsonwebtoken* per generare token in grado di simulare un utente reale in piattaforma e procedere ai test.

Importiamo poi le funzionalità di *jest* per il testing.

Importiamo anche il server dell'applicazione, così da poterlo utilizzare e chiudere una volta finiti i vari tests. In alcuni casi, ci sono anche gli import delle funzioni specifiche che vogliamo 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, così che la routine di testing possa terminare con successo, e restituire i risultati. In alcuni casi è 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, sono presenti i vari metodi test, i quali vengono chiamati per verificare la corretta funzionalità dell'API in questione.

Per ogni API verifichiamo tutti i vari codici di ritorno possibili, anche molteplici volte se sono possibili più modi per raggiungerli, garantendo quindi massima robustezza e sicurezza.

Questo è 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");
    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

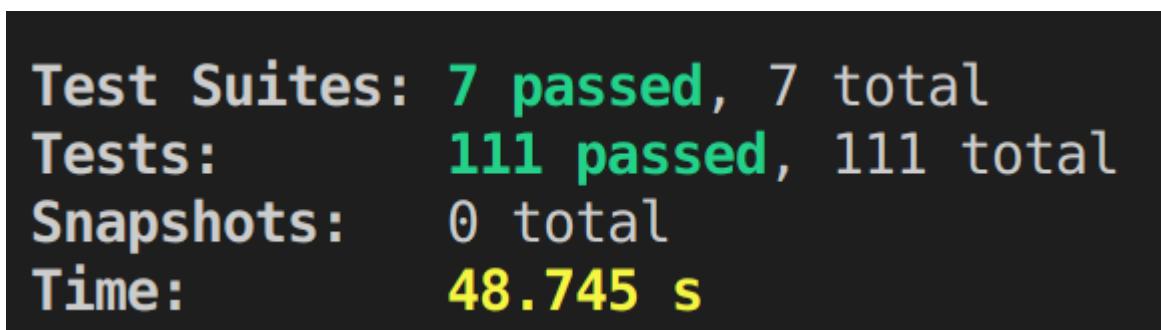
Per poter testare adeguatamente il nostro sistema, abbiamo creato uno script apposito, messo poi nel file `package.json`:

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 performato:



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.

File	Statements	Branches	Functions	Lines
backend	97.43%	38/39	50%	75% 3/4
backend/controller	87%	261/300	91.13%	75.49% 77/102
backend/models	100%	20/20	100%	100% 0/0
backend/router	100%	58/58	100%	100% 0/0
backend/utils	100%	14/14	100%	100% 5/5

Il grosso dei casi di codice non coperti sono errori fuori dalla nostra portata, dove chiediamo di ritornare un errore 500. Ci siamo resi conto che sono errori lanciati dalle varie funzioni di controllo di errori di [MongoDB](#).

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 riusciamo a coprire, tutta l'API è testata e controllata dalla nostra suite di 111 test cases, divisi in 7 sub suite.

GitHub, deployment e codice in 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**. Il progetto risulta attivo e può essere visionato al seguente link:

<https://gog-ilya-emeliyanov.vercel.app/>

Note per la versione online dell'applicazione

Abbiamo hostato il frontend e il backend su 2 servizi differenti e quindi di conseguenza anche su 2 domini differenti. Perciò volevamo avvertire che potrebbero verificarsi latenze nella trasmissione e ricezione dei pacchetti.

Questi problemi risultano purtroppo al di fuori del nostro controllo e ci scusiamo a priori per il disagio.

Nonostante questo, esiste sempre la possibilità di eseguire la versione di Money Expense in locale.

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 è stato configurato correttamente, al lancio del comando **npm run dev**, il server partirà. Attendere finché non appaiono le scritte "Server started at <http://localhost:3001>" e "Connected to MongoDB". A questo punto l'API è funzionante.
5. Aprire un nuovo terminale, ed entrare nella cartella **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 potrà accedere alla piattaforma.

Note per la documentazione locale

Nel caso si volesse consultare la documentazione del backend in locale, si può 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 si volesse visualizzare la documentazione anche hostata online, si può visitare il seguente link. <https://g09-72hkdodcm-matteopossamai.vercel.app/api-docs/>

Si utilizzi HTTPS per visitare tale link.