

Progetto:

BUDGETpal

Titolo del documento:

Sviluppo dell'applicazione web

Document Info:

Doc. Name	D4-BUDGETpal_SviluppoDellApplicazioneWeb	Doc. Number	F45
Description	Documento di Specifica dello sviluppo dell'applicazione web		

INDICE

1 Scopo del documento	3
2 User Flow.....	4
3 Implementazione e documentazione dell'applicazione	7
4 Struttura del progetto	7
5 Project Dependencies	11
6 Modelli del database	13
7 Project API's	18
8 Sviluppo API	24
9 API documentazione	33
10 Frontend Implementation	36
11 Testing	44
12 Risultati Testing	47
13 GitHub repository e informazioni sul deployment	50

1 Scopo del documento

Il seguente documento fornisce tutte le informazioni necessarie per descrivere lo sviluppo iniziale di una parte dell'applicazione web BUDGETpal.

Nel primo capitolo è presentato lo user flow, che consiste in una descrizione tramite diagramma di tutte le azioni eseguibili all'interno dell'applicazione BUDGETpal.

Si noti che, poiché l'applicazione è ancora in uno stato embrionale, solo una parte è stata implementata.

Nel capitolo vengono descritte le dipendenze installate, i modelli realizzati e le API implementate. Una parte della descrizione delle API implementate è resa attraverso il diagramma di estrazione delle risorse, che individua le risorse estratte.

Nel capitolo successivo viene spiegato l'utilizzo di Swagger per la documentazione delle API.

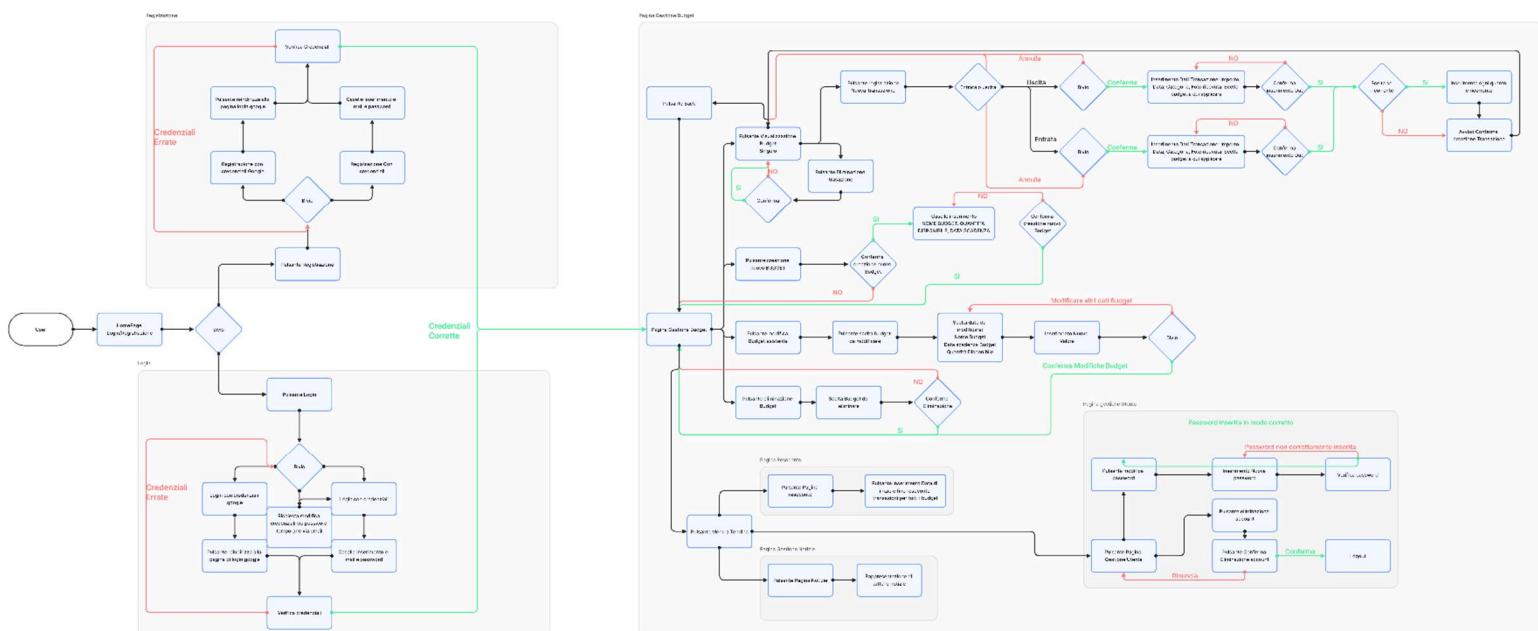
Successivamente, viene fornita una breve descrizione delle pagine implementate e una panoramica del repository di GitHub, comprese le istruzioni per eseguire il deployment.

2 User Flow

In questa sezione viene riportato lo user – flow dell'applicazione, il quale descrive le azioni possibili da fare all'interno del nostra web App, descritta nel dettaglio in questo documento.

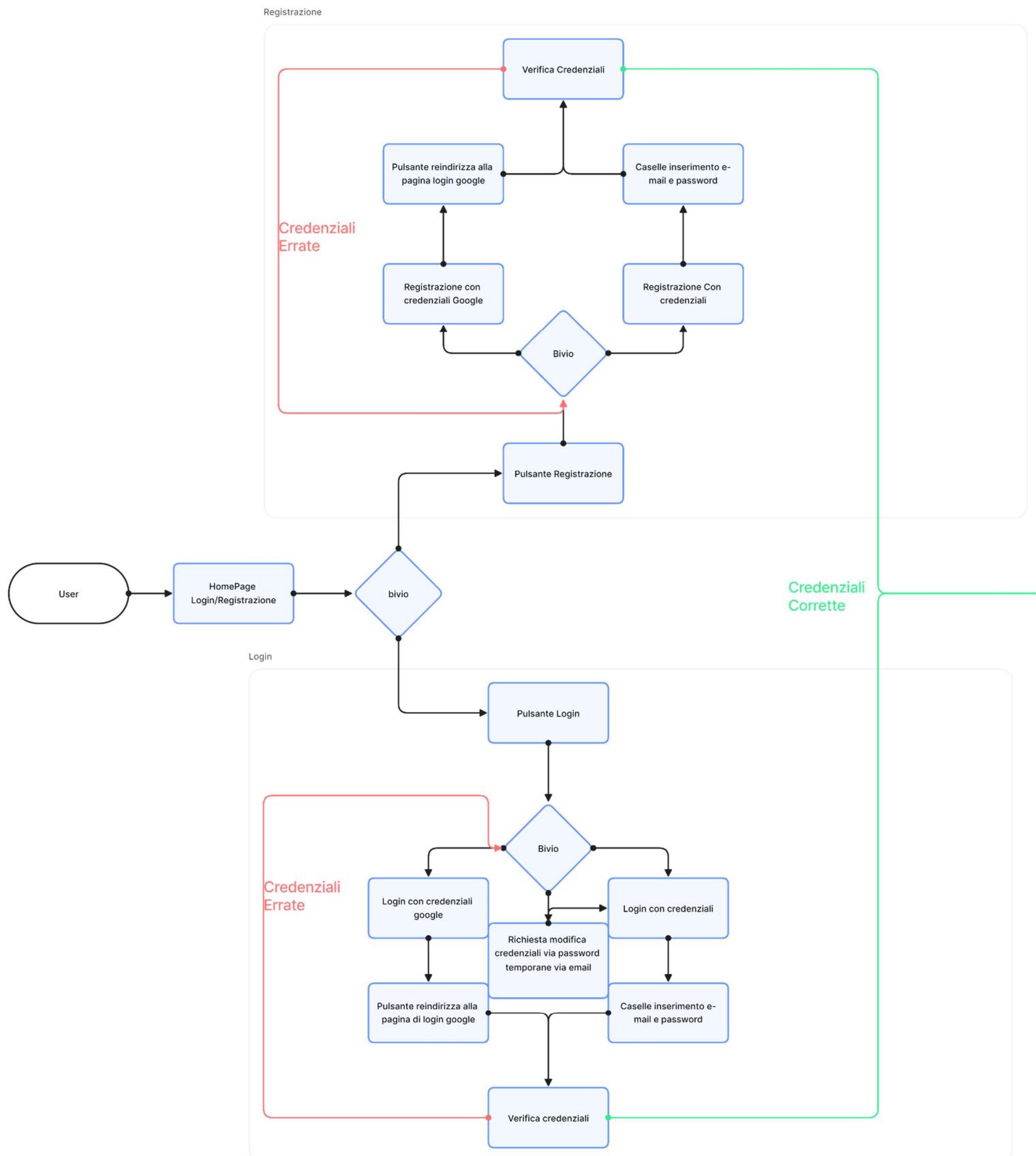
Bisogna notare che, lo user – flow da noi descritto rappresenterà tutte le azioni possibili che abbiamo pensato in fase di progettazione dell'applicazione, ciò vuole sottolineare che l'implementazione software non essendo ancora completa ma ad uno stadio iniziale, non consente di svolgere tutte le azioni descritte seguentemente.

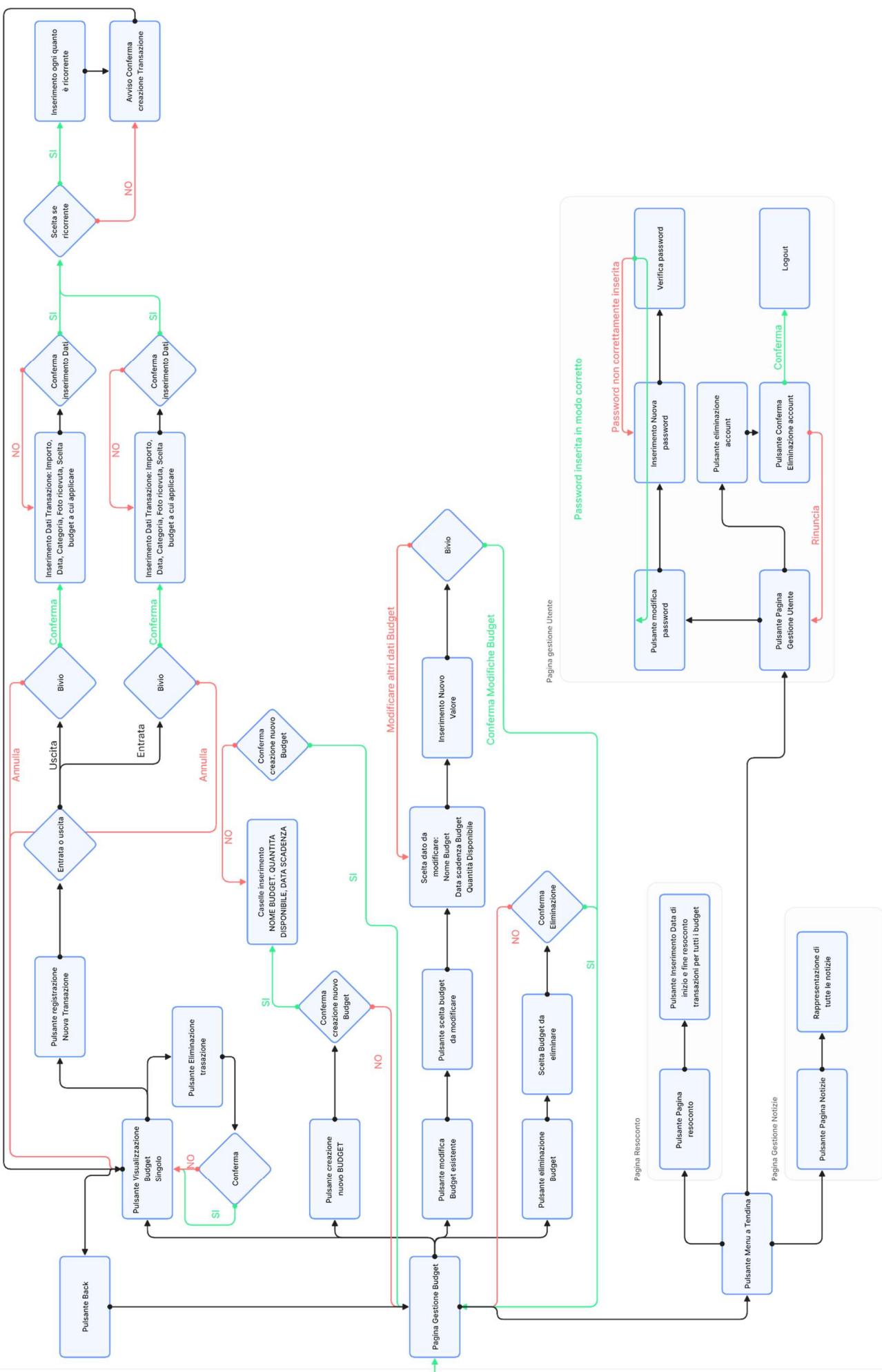
Qui sotto è riportata una didascalia dei vari componenti utilizzati nello user-flow.



Siamo consapevoli della scarsa qualità dell'immagine precedente; tuttavia, abbiamo deciso di includerla per illustrare la struttura del nostro user-flow.

Nelle due pagine successive, sono presenti due sezioni dell'immagine con uno zoom applicato per consentire una visione più dettagliata delle singole componenti che compongono il grafico, mantenendo la struttura illustrata sopra.





3 Implementazione e documentazione dell'applicazione

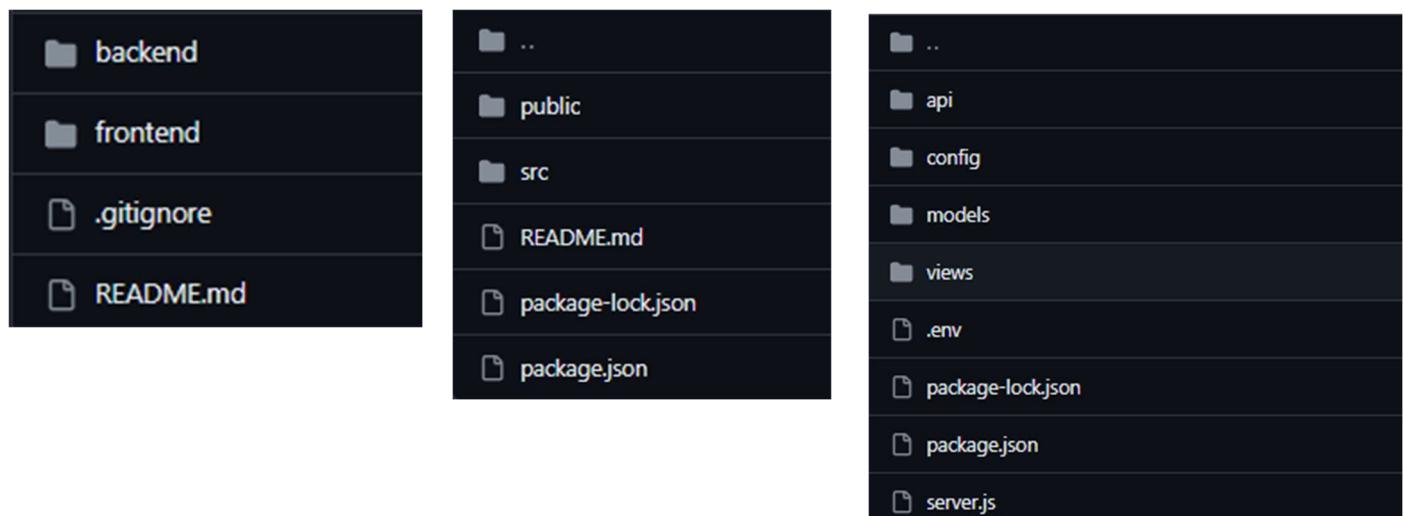
L'applicazione BUDGETpal è stata sviluppata utilizzando principalmente sviluppata con React e NodeJS, utilizzando dei framework di quest'ultimo come per esempio ExpressJS. A livello di backend per la memorizzazione dei vari dati abbiamo utilizzato MongoDB.

Come si potrà osservare in seguito, sono state sviluppate tutte le parti riguardante la registrazione, il login, il ripristino della password e il recupero della password, la logica della gestione dei budget, la logica della gestione delle transazioni, (nello specifico di quest'ultime due, si intende la creazione, l'eliminazione, e l'aggiornamento), la logica del menu. Sono state inoltre implementate delle logiche di backend che però non sono visibili causa mancata implementazione del frontend, come ad esempio la logica della gestione notizie.

È stato inoltre utilizzato anche l'API del sistema esterno *NodeMailer* per gestire l'invio di email agli utenti (riguardante sia il ripristino della password che per la conferma di avvenuta e corretta registrazione).

4 Struttura del progetto

Il codice è stato strutturato attraverso la suddivisione di backend e frontend, attraverso la creazione di due sottocartelle, ognuna fornita opportunamente dei file necessari, tra i quali i **package.json** che vanno a specificare le dependencies utilizzate. La struttura del progetto è rappresentata nella immagine sottostante.



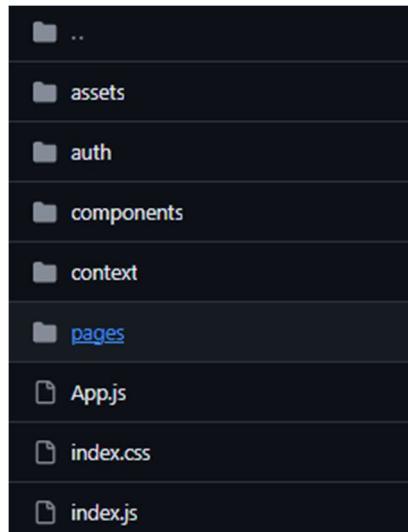
La directory principale è G45_Project ed è costituita in questo modo:

- **file .gitignore** è un file di configurazione utilizzato da Git per specificare i file e le cartelle che devono essere ignorati durante il versionamento. Questo file è utile per evitare di includere nel repository file temporanei, file di configurazione locali, file generati automaticamente o sensibili che non devono essere tracciati dalla gestione delle versioni.

- **file `README.md`** è un documento di testo che fornisce informazioni essenziali sul progetto. Solitamente posizionato nella radice del repository, fornisce istruzioni su come utilizzare il progetto, requisiti di installazione, configurazione e informazioni di base per gli utenti o i collaboratori.
- **cartella `/backend`** contiene tutta la logica di backend del progetto, che include la connessione al server, al database, la definizione dei modelli e delle API. Inoltre, comprende la logica per il collegamento di Swagger.
- **cartella `/frontend`** contiene tutta la logica di frontend del progetto. Qui vengono gestite le rotte, create tutte le pagine web, gestite le pagine stesse e le richieste HTTP che vengono inoltrate ai rispettivi endpoint del backend.
- **file `package.json`** nelle rispettive due sottocartelle definisce le dipendenze specifiche del backend e del frontend, rispettivamente. Queste dipendenze includono librerie, framework e moduli necessari per l'esecuzione e lo sviluppo delle applicazioni backend e frontend.
- **cartella `/(backend/frontend)/node_modules`** è una directory generata automaticamente da npm (Node Package Manager) quando si installano le dipendenze di un progetto. Contiene tutte le librerie di terze parti necessarie per il corretto funzionamento dell'applicazione.
- **cartella `/backend/api`** contiene le definizioni delle API del backend e le relative descrizioni per Swagger associate ad ognuna di esse. Qui sono definite le varie rotte e operazioni che l'applicazione backend supporta, insieme alla documentazione che descrive l'uso di ciascuna API, che è utile per la creazione di documentazione automatica e la gestione delle API.
- **cartella `/backend/config`** contiene la configurazione specifica del backend dell'applicazione. All'interno di questa cartella, il file "db.js" gestisce la connessione al database MongoDB utilizzato dall'applicazione. Questo file è responsabile di caricare le variabili d'ambiente dal file.env utilizzando dotenv, che è una pratica comune per mantenere le informazioni sensibili, come le credenziali del database, al di fuori del codice sorgente. Successivamente, il file utilizza la libreria Mongoose per connettersi al database MongoDB utilizzando l'URL specificato nella variabile d'ambiente MONGODB_URI.
- **cartella `/backend/models`** contiene le definizioni dei modelli dati utilizzati dall'applicazione. Questi modelli rappresentano le entità del dominio dell'applicazione e definiscono la struttura dei dati memorizzati nel database.
- **cartella `/backend/views`** contiene la definizione di un singolo file HTML, visualizzato dopo l'avvenuta conferma del account in fase di registrazione.
- **file `.env`** è un file di configurazione utilizzato per memorizzare le variabili d'ambiente sensibili o configurabili utilizzate dall'applicazione. Contiene informazioni come le credenziali di accesso al database, le chiavi API, le impostazioni di configurazione e altre variabili sensibili, tra cui password e indirizzo email per l'invio dell'Email di conferma, utilizzato da *Nodemailer*.
- **file `/backend/server.js`** contiene il codice principale per avviare e gestire il server backend dell'applicazione. Serve a configurare il server Express per gestire le richieste HTTP, definire le rotte delle API, configurare i middleware per la gestione delle richieste e avviare il server per iniziare ad accettare le richieste dei client. Inoltre, gestisce anche la generazione e la visualizzazione della documentazione delle API utilizzando Swagger.
- **cartella `/frontend/public`** è una cartella speciale all'interno di un'applicazione React. Contiene file statici come HTML, immagini, font, icone e altri asset che saranno serviti al

client. Questi file possono essere accessibili pubblicamente e possono essere utilizzati per la visualizzazione dei contenuti dell'applicazione nel browser.

- **cartella /frontend/src** è una parte essenziale di un'applicazione React. Contiene il codice sorgente principale dell'applicazione, inclusi i componenti React, i file di stile, gli script JavaScript e altri file ausiliari necessari per la logica e la presentazione dell'interfaccia utente. Nello specifico questa cartella è come segue:



- **cartella /frontend/src/assets** è utilizzata per archiviare risorse statiche come immagini, font, icone e altri file multimediali utilizzati nell'applicazione.
- **cartella /frontend/src/auth** contiene il codice relativo alla gestione dell'autenticazione all'interno dell'applicazione frontend. Essa include principalmente il file "userAction.js", che contiene le azioni e le funzioni per le operazioni di autenticazione come il login, la registrazione, il recupero della password e altre funzionalità correlate.
 - Il file "userAction.js" utilizza la libreria Axios per inviare richieste HTTP agli endpoint del backend. Le funzioni all'interno di questo file gestiscono la logica di front-end per elaborare le risposte dal backend, gestire gli errori e aggiornare lo stato dell'applicazione di conseguenza.
- **cartella /frontend/src/components** contiene i componenti React riutilizzabili che vengono utilizzati nell'applicazione per costruire l'interfaccia utente. Questi componenti possono essere composti insieme per creare le pagine web dell'applicazione.
- **cartella /frontend/src/context** contiene il codice relativo alla gestione dello stato globale dell'applicazione utilizzando il Context API di React. Questo API permette di condividere lo stato tra componenti React senza dover passare manualmente le props attraverso ogni livello di gerarchia.
- **file /frontend/src/App.js** viene utilizzato per definire il layout e la struttura generale dell'applicazione, nonché per gestire il Routing tra le diverse pagine e le logiche di stato globale.
- **file /frontend/src/index.css** è un file di stile globale utilizzato per definire regole CSS che si applicano ad alcune componenti dell'applicazione React.
- **file /frontend/src/index.js** è il punto di ingresso principale dell'applicazione React. È il primo file che viene eseguito quando l'applicazione viene avviata e viene utilizzato per rendere il componente radice dell'applicazione nel DOM del browser.

Riassumendo quindi possiamo dire che il progetto è strutturato seguendo un'architettura client-server, dove la parte frontend e la parte backend sono state separate per una migliore organizzazione e scalabilità del codice. Ecco come è strutturato il progetto

Parte Frontend:

- La parte frontend del progetto è responsabile della presentazione dell'interfaccia utente e dell'interazione con l'utente.
- È stata sviluppata utilizzando React.js e girata localmente sulla porta 3000.
- Tutti i file e le cartelle relativi al frontend sono contenuti nella directory "/frontend".
- Questa parte contiene i componenti React, i file di stile, le risorse grafiche e altri asset necessari per la visualizzazione e l'interazione dell'utente.
- Il file "package.json" nella directory **"/frontend"** contiene le dipendenze specifiche del frontend e le istruzioni per l'avvio dell'applicazione frontend.

Parte backend:

- La parte backend del progetto gestisce la logica di business, l'accesso ai dati e altre funzionalità lato server.
- È stata sviluppata utilizzando Node.js con Express.js come framework web e MongoDB come database.
- Girata localmente sulla porta 4500.
- Tutti i file e le cartelle relativi al backend sono contenuti nella directory "/backend".
- Questa parte contiene i file di Routing, i modelli dei dati, le configurazioni del server e altri componenti necessari per la gestione delle richieste client e il recupero dei dati dal database.
- Il file "package.json" nella directory **"/backend"** contiene le dipendenze specifiche del backend e le istruzioni per l'avvio del server backend.

Interazione tra frontend e backend:

- Il frontend comunica con il backend tramite richieste HTTP, inviando dati al server e ricevendo risposte JSON.
- Le chiamate API vengono indirizzate agli endpoint definiti nel backend, come "/user/login" per il login dell'utente o "/api/data" per il recupero dei dati.
- Le risposte del backend vengono elaborate dal frontend per aggiornare l'interfaccia utente di conseguenza.

5 Project Dependencies

I moduli Node utilizzati e aggiunti ai file package.json nel campo dependencies. Poiché frontend e backend sono stati separati ed ognuno possiede le sue dipendenze nella seguente descrizione verranno suddivise in modo opportuno:

backend dependencies:

```
"dependencies": {  
    "bcrypt": "^5.1.1",  
    "cors": "^2.8.5",  
    "dotenv": "^16.3.1",  
    "express": "^4.18.2",  
    "express-ip": "^1.0.4",  
    "mongoose": "^8.0.0",  
    "nodemailer": "^6.9.7",  
    "nodemon": "^3.0.1",  
    "swagger-jsdoc": "^6.2.8",  
    "swagger-ui-express": "^5.0.0",  
    "uuid": "^9.0.1"  
}
```

1. **bcrypt**: Libreria per l'hashing delle password. Viene utilizzata per crittografare e proteggere le password degli utenti all'interno del database.
2. **cors**: Middleware per Express che gestisce le richieste HTTP cross-origin. Consente al server di rispondere alle richieste provenienti da domini diversi.
3. **dotenv**: Carica le variabili d'ambiente da un file .env nel processo Node.js, permettendo di configurare facilmente le variabili di ambiente come le credenziali di accesso al database.
4. **express**: Framework web per Node.js. Viene utilizzato per gestire le richieste HTTP, definire le rotte dell'applicazione e configurare il server backend.
5. **express-ip**: Middleware per Express che consente di ottenere informazioni sull'indirizzo IP del client che ha effettuato la richiesta.
6. **mongoose**: Libreria Node.js per interfacciarsi con il database MongoDB. Fornisce un'interfaccia di tipo ORM (Object-Relational Mapping) per gestire facilmente i dati del database.
7. **nodemailer**: Libreria per Node.js che permette l'invio di email. Viene utilizzata per inviare email di notifica o di conferma all'utente.
8. **nodemon**: Strumento di utilità per lo sviluppo che monitora i file nel progetto Node.js e riavvia automaticamente il server ogni volta che viene rilevato un cambiamento nel codice.
9. **swagger-jsdoc**: Strumento per generare la documentazione delle API utilizzando JSDoc. Viene utilizzato per documentare le API del backend in modo automatico e standardizzato.
10. **swagger-ui-express**: Middleware per Express che fornisce un'interfaccia utente interattiva per esplorare e testare le API documentate con Swagger.

frontend dependencies:

```
"dependencies": {  
    "@testing-library/jest-dom": "^5.17.0",  
    "@testing-library/react": "^13.4.0",  
    "@testing-library/user-event": "^13.5.0",  
    "axios": "^1.6.2",  
    "bootstrap": "^5.3.3",  
    "formik": "^2.4.5",  
    "react": "^18.2.0",  
    "react-bootstrap": "^2.10.2",  
    "react-dom": "^18.2.0",  
    "react-icons": "^4.12.0",  
    "react-loader-spinner": "^5.4.5",  
    "react-redux": "^8.1.3",  
    "react-router-dom": "^6.19.0",  
    "react-scripts": "5.0.1",  
    "redux": "^4.2.1",  
    "redux-react-session": "^2.6.1",  
    "redux-thunk": "^2.4.2",  
    "styled-components": "^6.1.1",  
    "web-vitals": "^2.1.4",  
    "yup": "^1.3.2"  
},
```

1. **@testing-library/jest-dom, @testing-library/react, @testing-library/user-event:** Librerie di testing per React che forniscono utility per testare i componenti React e le interazioni dell'utente.
2. **axios:** Libreria per le chiamate HTTP asincrone. Viene utilizzata per effettuare richieste AJAX al server backend per ottenere o inviare dati.
3. **bootstrap:** Framework CSS per la creazione di interfacce utente responsive e moderne. Viene utilizzato per la progettazione e lo stile dell'interfaccia utente.
4. **formik:** Libreria per la gestione dei moduli nei form React. Semplifica la gestione dello stato dei form e la validazione dei dati inseriti dall'utente.
5. **react, react-dom:** Librerie React per la creazione di interfacce utente reattive e dinamiche.
6. **react-bootstrap:** Implementazione dei componenti Bootstrap per React. Fornisce una serie di componenti React predefiniti per la creazione di interfacce utente.
7. **react-icons:** Libreria di icone React. Fornisce una vasta gamma di icone pronte all'uso per migliorare l'aspetto e la funzionalità dell'applicazione.
8. **react-loader-spinner:** Componente React per mostrare un indicatore di caricamento animato durante le operazioni asincrone.
9. **react-redux:** Libreria per la gestione dello stato in React, utilizzata in combinazione con Redux per la gestione dello stato globale dell'applicazione.
10. **react-router-dom:** Libreria per il routing nel frontend React. Consente di gestire la navigazione tra diverse pagine dell'applicazione in base all'URL.
11. **react-scripts:** Scripts di avvio preconfigurati per React. Fornisce comandi per avviare, testare e costruire l'applicazione React.
12. **redux:** Libreria per la gestione dello stato in applicazioni JavaScript, utilizzata in combinazione con React per mantenere lo stato globale dell'applicazione.
13. **redux-react-session:** Libreria per la gestione della sessione in Redux. Permette di salvare e ripristinare la sessione utente nell'applicazione.
14. **redux-thunk:** Middleware per Redux che consente l'uso di azioni asincrone. È utilizzato per gestire azioni che comportano chiamate asincrone al backend.

15. **styled-components:** Libreria per lo styling CSS in React. Permette di definire stili CSS come componenti React, migliorando la modularità e la manutenibilità dello stile.
16. **web-vitals:** Libreria per la misurazione delle prestazioni web. Viene utilizzata per monitorare e migliorare le metriche di prestazioni dell'applicazione web.
17. **yup:** Libreria per la validazione dei dati in JavaScript. Viene utilizzata per definire schemi di validazione e verificare la correttezza dei dati inseriti dall'utente nei form.

6 Modelli del database

Per gestire in modo efficiente i dati all'interno dell'applicazione, abbiamo definito diversi modelli di dati, partendo dalle classi sviluppate nel class Diagram del documento D3. Le risorse fondamentali da gestire nel nostro sistema hanno richiesto la creazione di sei modelli distinti, ognuno dei quali corrisponde a una specifica collezione nel database.

Può capitare che nella descrizione dei modelli non siano presenti negli attributi la specifica di **required** oppure di **unique**. La loro mancanza è data dalla presenza di opportuni controlli e sufficienti costrutti logici che verificano la loro unicità e obbligatorietà all'interno sia di codice presente nella logica di backend che nella logica di frontend.

Un esempio potrebbe essere il campo email durante la fase di registrazione. Attraverso la libreria **Yup** verifichiamo la validità e correttezza del campo, e in caso di errori, l'utente verrebbe opportunamente notificato impedendone così la scorretta registrazione.

Modello User

Per memorizzare i dati degli utenti che si iscrivono al nostro sito web abbiamo creato il modello User.

```
//in this file we use mongoose to create a model to enable us to communicate with our MongoDB Database
// Import the mongoose library
const mongoose = require('mongoose');

//Define a schema for the "User" documents.
//The schema defines the structure and types of fields for user documents:
const Schema = mongoose.Schema;

//In this schema, I've defined fields for the user's name, email, password, and date of birth, each with its respective data type.
const UserSchema = new Schema({
  name: String,
  email: String,
  password: String,
  dateOfBirth: Date,
  verified: Boolean
});

//Create a Mongoose model based on the "UserSchema". The model is named "User".
//The mongoose.model function creates a model that represents a collection in your MongoDB database. In this case, it represents the "User" collection.
const User = mongoose.model('User', UserSchema);

//Export the "User" model to make it available for use in other parts of your application:
module.exports = User;

//Now, you can use the "User" model to interact with the "User" collection in your MongoDB database.
//You can perform operations like inserting, updating, querying, and deleting user documents in your database using this model
```

Come specificato precedentemente, l'obbligatorietà dei campi e la loro unicità non è specificata nello Schema di Mongoose, perché è garantita dalla logica implementata nel codice del programma.

Si pone necessaria la definizione degli attributi sensibili come *name*, *email* e *dateOfBirth* come specificato nel Requisito **RNF 2.1** nel D1 che richiede esplicitamente di verificare l'identità degli utenti. In questo modo si riesce a raccogliere i dati sensibili degli utenti che confermano la loro reale identità.

L'attributo *verified* di tipo Boolean, impostato come valore di default a false, serve a verificare la veridicità dell'utente e solo dopo apposita conferma tramite sistema di email implementato grazie all'ausilio di *Nodemailer*.

Ogni modello in MongoDB è dotato automaticamente di un campo denominato "**_id**", che viene generato da MongoDB per identificare in modo univoco ciascun record all'interno della collezione. Questo campo "**_id**" è unico per ogni documento e viene creato automaticamente quando viene aggiunto un nuovo documento al database.

La password all'interno del database non è in chiaro, poiché nella logica del nostro codice è stata implementata con l'ausilio della libreria per l'hashing **bcrypt**.

```
_id: ObjectId('66119473847819a47c28b45a')
name: "Gabriele Menestrina"
email: "menestrina.gabriele@gmail.com"
password: "$2b$10$X1f9CecibMwVnzSwzR3OnFH/8mYyS92khMVij2t14QSY1lpPa6i"
dateOfBirth: 2024-04-13T00:00:00.000+00:00
verified: true
__v: 0
```

L'immagine precedente è un esempio di elemento nella collezione nel database. Il campo *verified* è impostato a **true** perché l'utente ha verificato l'autenticità del suo account, durante la fase di registrazione.

Modello Budget

Per memorizzare i Budget creati dagli utenti presenti nella pagina di gestione dei Budget abbiamo creato il modello Budget. In questa immagine mostriamo lo schema del dato.

```
const mongoose = require(`mongoose`);
const Schema = mongoose.Schema;

const BudgetSchema = new Schema({
    userID: String,
    name: String,
    current: Number,
    amount: Number,
    createdAt: Date,
    description: String,
});

const Budget = mongoose.model(`Budget`, BudgetSchema);

module.exports = Budget;
```

Ogni modello in MongoDB è dotato automaticamente di un campo denominato "**_id**", che viene generato da MongoDB per identificare in modo univoco ciascun record all'interno della collezione. Questo campo "**_id**" è

unico per ogni documento e viene creato automaticamente quando viene aggiunto un nuovo documento al database.

L'attributo userID, identifica lo user al quale è associato il relativo Budget presente nel Database. Va notato che lo userID è l'equivalente dell'attributo `_id` del modello User.

L'attributo name rappresenta il nome del Budget assegnatoli dall'utente in fase di aggiunta di un budget tramite l'apposito form.

L'attributo current è utilizzato per tenere traccia delle spese associate al Budget. Ogni spesa aggiunta o eliminata andrà a modificare opportunamente il valore di tale campo attraverso due opportune API di tipo patch.

L'attributo "amount" viene definito durante la registrazione del budget e rappresenta il valore massimo che il budget può raggiungere senza superare il limite stabilito. Nel programma è implementata una logica che calcola la differenza tra l'attributo "amount" e il valore corrente del budget per verificare se il massimo consentito è stato superato. Questa verifica viene eseguita per garantire che il budget rimanga entro i limiti prestabiliti e che eventuali superamenti siano identificati e gestiti correttamente.

I rimanenti attributi servono per delle specifiche aggiuntive.

Un esempio di elemento nella collezione nel database è il seguente:

```
_id: ObjectId('66225b645b1b9ace00e60964')
userID: "66225b185b1b9ace00e60957"
name: "Alimentari"
current: 34.83
amount: 100
createdAt: 2024-04-19T11:54:12.095+00:00
description: "Questo budget serve per gestire le spese relative agli alimentari"
__v: 0
```

Modello Transaction

Per memorizzare le Transazioni create dagli utenti presenti nella pagina di gestione dei Budget abbiamo creato il modello Transaction. In questa immagine mostriamo lo schema del dato.

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const TransactionSchema = new Schema({
    budgetID: String,
    name: String,
    createdAt: Date,
    amount: Number,
    currentAmount: Number,
    description: String,
});

const Transaction = mongoose.model(`Transaction`, TransactionSchema);

module.exports = Transaction;
```

Ogni modello in MongoDB è dotato automaticamente di un campo denominato "`_id`", che viene generato da MongoDB per identificare in modo univoco ciascun record all'interno della collezione. Questo campo "`_id`" è

unico per ogni documento e viene creato automaticamente quando viene aggiunto un nuovo documento al database.

L'attributo budgetID, identifica il Budget al quale è associato la relativa Transaction presente nel Database. Va notato che lo budgetID è l'equivalente dell'attributo `_id` del modello Budget.

L'attributo name rappresenta il nome della Transaction assegnatoli dall'utente in fase di aggiunta di una Transaction tramite l'apposito form.

L'attributo amount, serve al Budget a cui la Transaction è associata per calcolare il valore current.

I rimanenti attributi servono per delle specifiche aggiuntive.

Un esempio di elemento nella collezione nel database è il seguente:

```
_id: ObjectId('66225b9d5b1b9ace00e60970')
budgetID: "66225b645b1b9ace00e60964"
name: "Spesa varia"
createdAt: 2024-04-19T11:55:09.511+00:00
amount: 34.83
currentAmount: 0
description: "Eurospar"
__v: 0
```

Modello UserVerification

Il modello UserVerification è un modello che serve in fase di registrazione per la verifica dell'account. In questa immagine mostriamo lo schema del dato.

```
const mongoose = require(`mongoose`);

const Schema = mongoose.Schema;

const UserVerificationSchema = new Schema({
  userID: String,
  uniqueString: String,
  createdAt: Date,
  expiredAt: Date,
});

const UserVerification = mongoose.model(`UserVerification`, UserVerificationSchema);

module.exports = UserVerification;
```

Ogni modello in MongoDB è dotato automaticamente di un campo denominato "`_id`", che viene generato da MongoDB per identificare in modo univoco ciascun record all'interno della collezione. Questo campo "`_id`" è unico per ogni documento e viene creato automaticamente quando viene aggiunto un nuovo documento al database.

`userID` è un attributo che fa riferimento all'utente associato alla verifica dell'account ed è un identificatore univoco dell'utente nel sistema.

`uniqueString` è attributo che contiene una stringa unica generata per la verifica dell'account ed è utilizzato come token di verifica univoco per associare l'utente al processo di verifica.

`createdAt` è un attributo che indica la data e l'ora in cui è stato creato il record di verifica dell'account.

expiredAt è un attributo che indica la data e l'ora di scadenza del record di verifica dell'account. Dopo questa data e ora, il record verrà considerato scaduto e viene eliminato dal sistema, di conseguenza l'utente dovrà registrarsi nuovamente e verificarsi entro la data limite dettata da questo campo (impostata da 6 ore).

Modello PasswordReset

Il modello PasswordReset è un modello che serve per resettare la password in caso di smarrimento. In questa immagine mostriamo lo schema del dato.

```
const mongoose = require(`mongoose`);

const Schema = mongoose.Schema;

const PasswordResetSchema = new Schema({
  userID: String,
  resetString: String,
  createdAt: Date,
  expiredAt: Date,
});

const PassworsReset = mongoose.model(`PassworsReset`, PasswordResetSchema);

module.exports = PassworsReset;
```

Ogni modello in MongoDB è dotato automaticamente di un campo denominato "**_id**", che viene generato da MongoDB per identificare in modo univoco ciascun record all'interno della collezione. Questo campo "**_id**" è unico per ogni documento e viene creato automaticamente quando viene aggiunto un nuovo documento al database.

L'attributo userID serve per identificare l'utente che vuole cambiare la password.

L'attributo resetString corrisponde alla stringa sostitutiva della vecchia password, che andrà a modificare la password cifrata all'interno del Modello User.

L'attributo createdAt serve ad indicare la data di creazione della richiesta di reset della password e per calcolare il tempo intercorso tra la creazione della richiesta e la data di scadenza.

L'attributo expiredAt serve a delineare il tempo massimo entro il quale si può resettare la password con il modulo di richiesta corrente.

Un esempio di elemento nella collezione nel database è il seguente:

```
_id: ObjectId('65d76531e4761eca80921b25')
userID : "65d7607ce4761eca80921b12"
resetString : "$2b$10$PBcBdao3qsGliwjZX7muT0AhrbtJREkWAFoAn8r0a.hk5kJKks8iK"
createdAt : 2024-02-22T15:16:01.813+00:00
expiredAt : 2024-02-23T00:36:01.813+00:00
__v : 0
```

Modello News

Il modello News è un modello che serve per rappresentare le notizie. In questa immagine mostriamo lo schema del dato.

```
const mongoose = require('mongoose');

const newsSchema = new mongoose.Schema({
  name: {type: String, required: true},
  link: {type: String, required: true},
  date: {type: Date, default: Date.now}
});

const News = mongoose.model('News', newsSchema);
```

Ogni modello in MongoDB è dotato automaticamente di un campo denominato "**_id**", che viene generato da MongoDB per identificare in modo univoco ciascun record all'interno della collezione. Questo campo "**_id**" è unico per ogni documento e viene creato automaticamente quando viene aggiunto un nuovo documento al database.

L'attributo name rappresenta il nome del modello News.

L'attributo link serve per associare la News presente nella WebApp con la effettiva notizia.

L'attributo date serve per tenere traccia della data della notizia.

7 Project API's

Questa sezione del documento descrive le varie API implementate a partire dal diagramma delle classi presente nel documento D3. Verrà utilizzato un diagramma per rappresentare l'estrazione delle risorse dal diagramma delle classi e un altro per rappresentare le risorse sviluppate, creando così una panoramica chiara delle API disponibili e delle loro interazioni.

Estrazione delle risorse dal class Diagram

Questo diagramma mostra come abbiamo estratto le varie risorse sviluppate nel sistema a partire dal class Diagram.

Inizialmente le prime risorse che abbiamo individuato sono stati i 6 modelli che abbiamo descritto nel [Modelli nel database](#).

A partire dalle classi presenti abbiamo individuato i “i tipi di dato” che dovevano essere memorizzati nel database in MongoDB preservandone gli attributi fondamentali. Infatti gli attributi specificati nel class Diagram sono stati riportati anche nel diagramma delle risorse.

Per quanto l'omissione di alcuni attributi, deriva dal fatto che la loro necessità nella costruzione del Database era superflua e non sempre necessaria, abbiamo quindi deciso di ottimizzare le risorse minimizzando gli attributi necessari. Un esempio pratico riguarda la risorsa User, dove all'interno della gestione User è stato inserito l'attributo email e password oltre all'id del record. In questo caso risultava non necessaria la presenza dell'email, poiché la sua gestione avviene interamente tramite userID.

Un altro caso di omissione è la classe Report, difatti l'implementazione non richiede necessariamente la sua presenza in quanto è gestibile tramite semplice funzioni a livello di front-end e API che prendono le informazioni necessarie, come ad esempio le transazioni.

Successivamente abbiamo trasformato alcuni metodi delle classi in ulteriori risorse del nostro sistema.

Le risorse a cui sono collegati i modelli sopra descritti sono nient'altro che le API che abbiamo sviluppato e coincidono con alcuni metodi delle classi. Non tutti i metodi sono chiaramente diventati API in quanto alcuni sono semplicemente delle funzioni ausiliarie e di supporto alle API.

Di queste risorse viene specificato il metodo (che si tratta di GET, POST, PATCH, DSELETE a seconda del loro compito) e i parametri che richiedono per essere eseguiti (si è specificato body nel caso in cui servissero più di tre parametri in input per dire che quella data risorsa richiede tutti, o comunque molti, attributi del modello).

Infine viene specificato anche se l'effetto di quella risorsa ha rilevanza nel front-end oppure nel back-end: per le varie risorse i tipo POST, PATCH e DELETE l'effetto è chiaramente sul back-end perché il loro compito è di salvare, modificare o eliminare una risorsa nel database, non fornendo informazioni in front-end oltre che un messaggio di conferma. Le risorse di tipo GET hanno invece un chiaro effetto sul front-end perché interrogano il database per chiedere alcune risorse e poi le mostrano nel front-end

Nella pagina seguente alleghiamo il diagramma che illustra quanto descritto, mentre una specifica più attenta della API sarà fatta nel [diagramma delle risorse](#)

N.B. nel seguente diagramma può essere che all'interno di alcune API ci siano parametri non riconducibili ai modelli, questo perché sono provenienti da funzioni che restituiscono quei valori per essere utilizzati da quest'ultime.

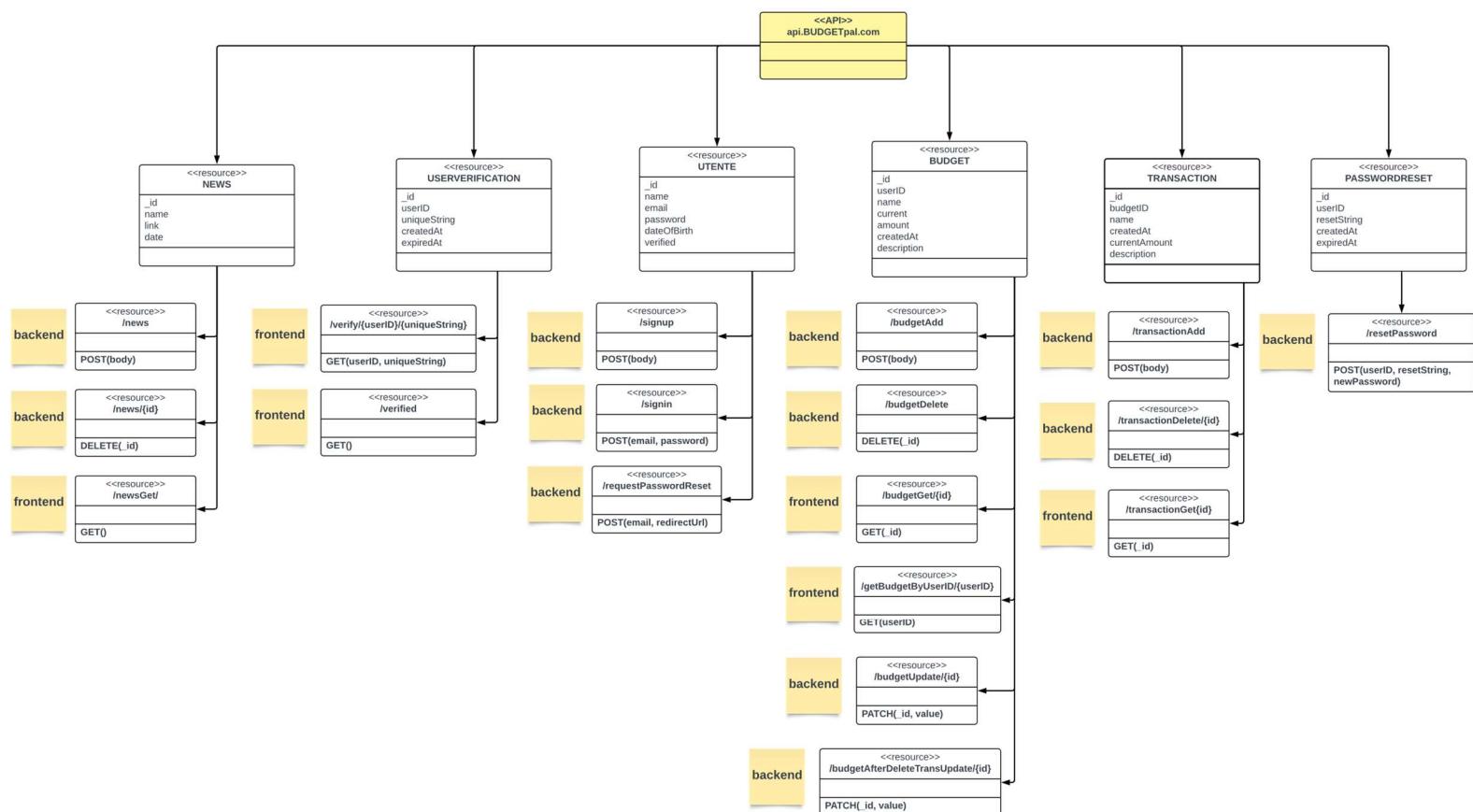
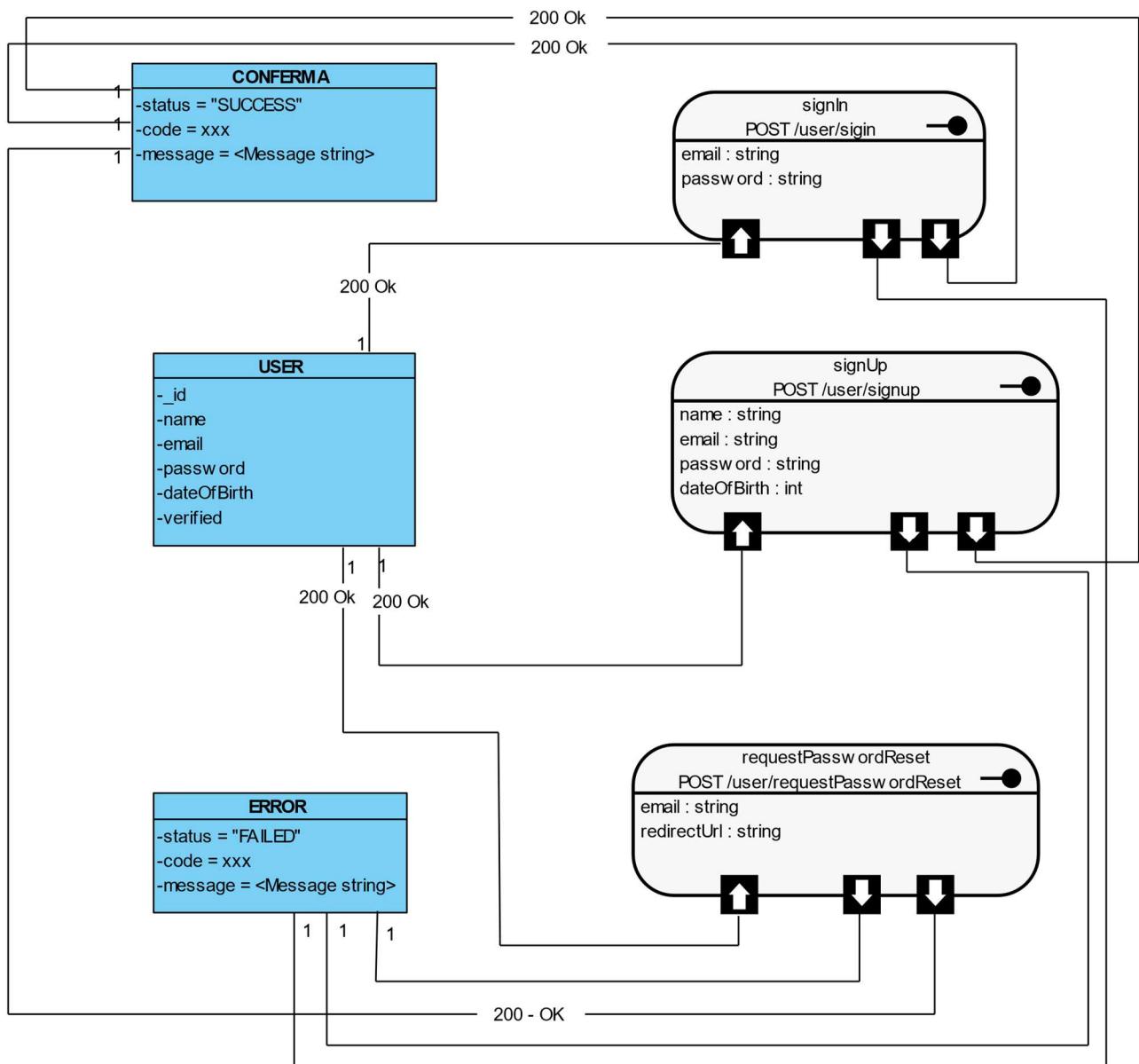


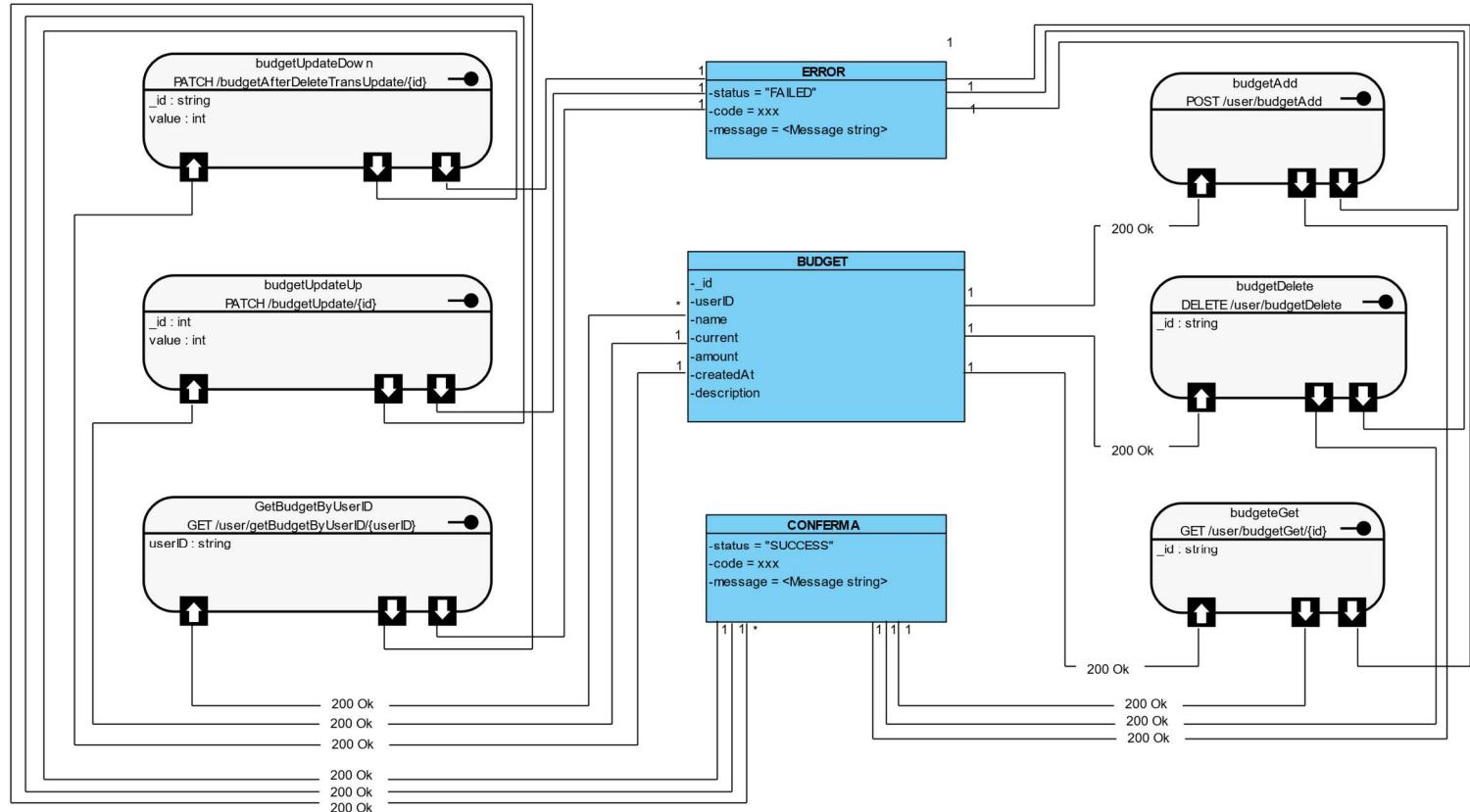
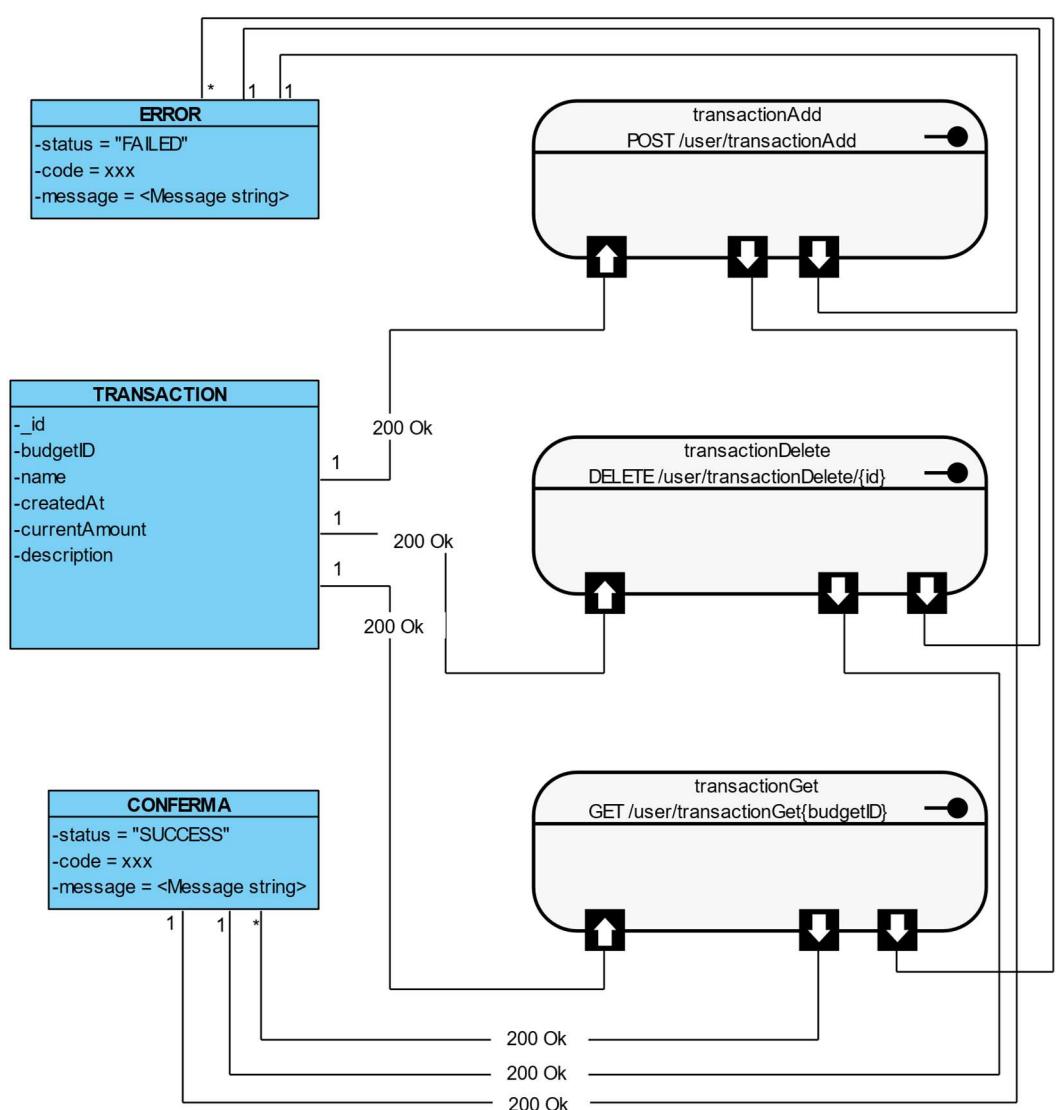
Diagramma delle risorse

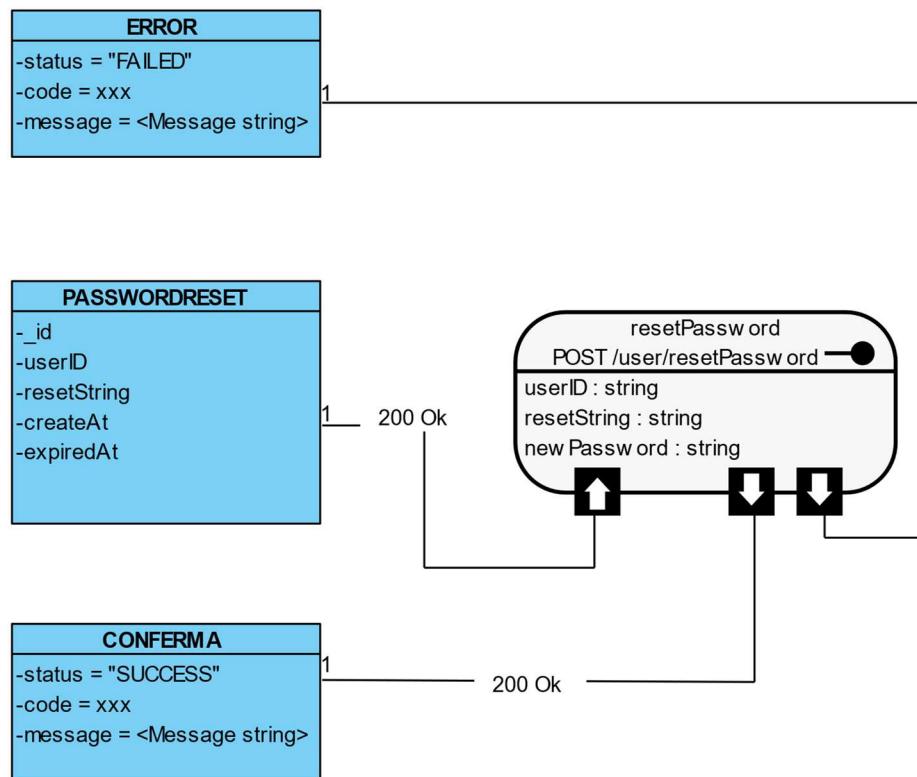
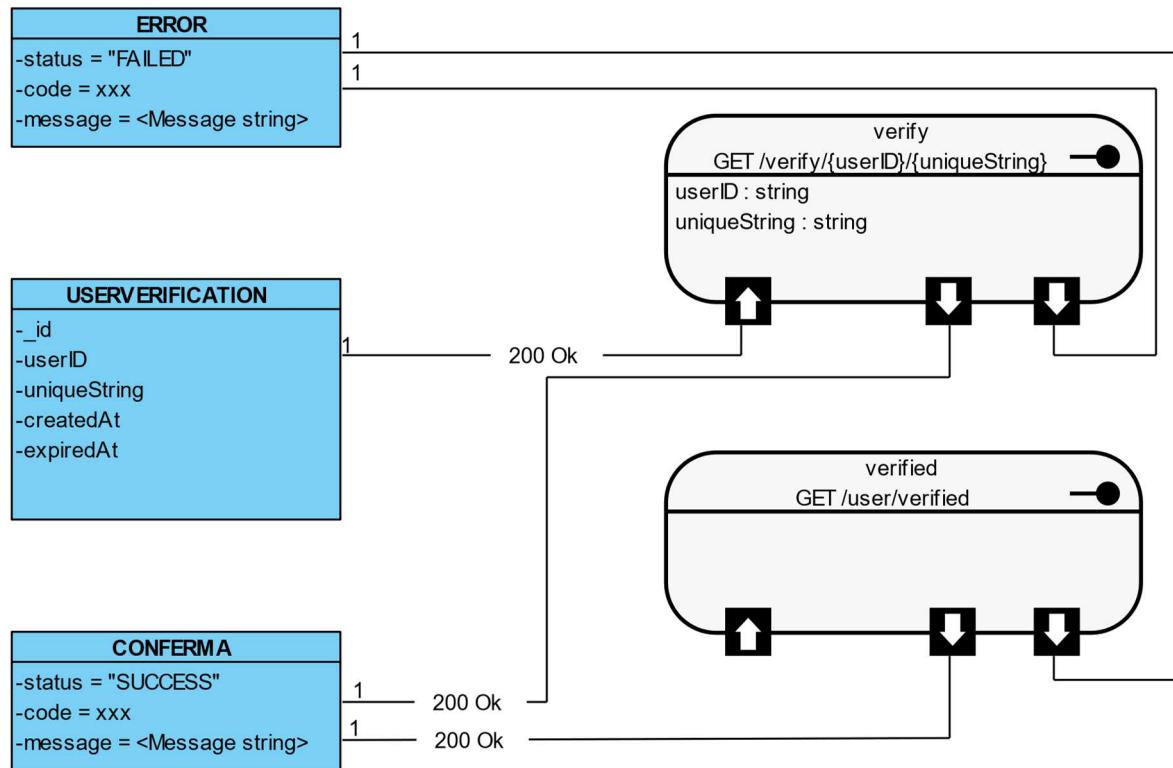
Nel diagramma delle risorse seguente vengono illustrate le varie API sviluppate nel progetto. Per migliorare la chiarezza e la modularità, il diagramma è stato diviso per modello.

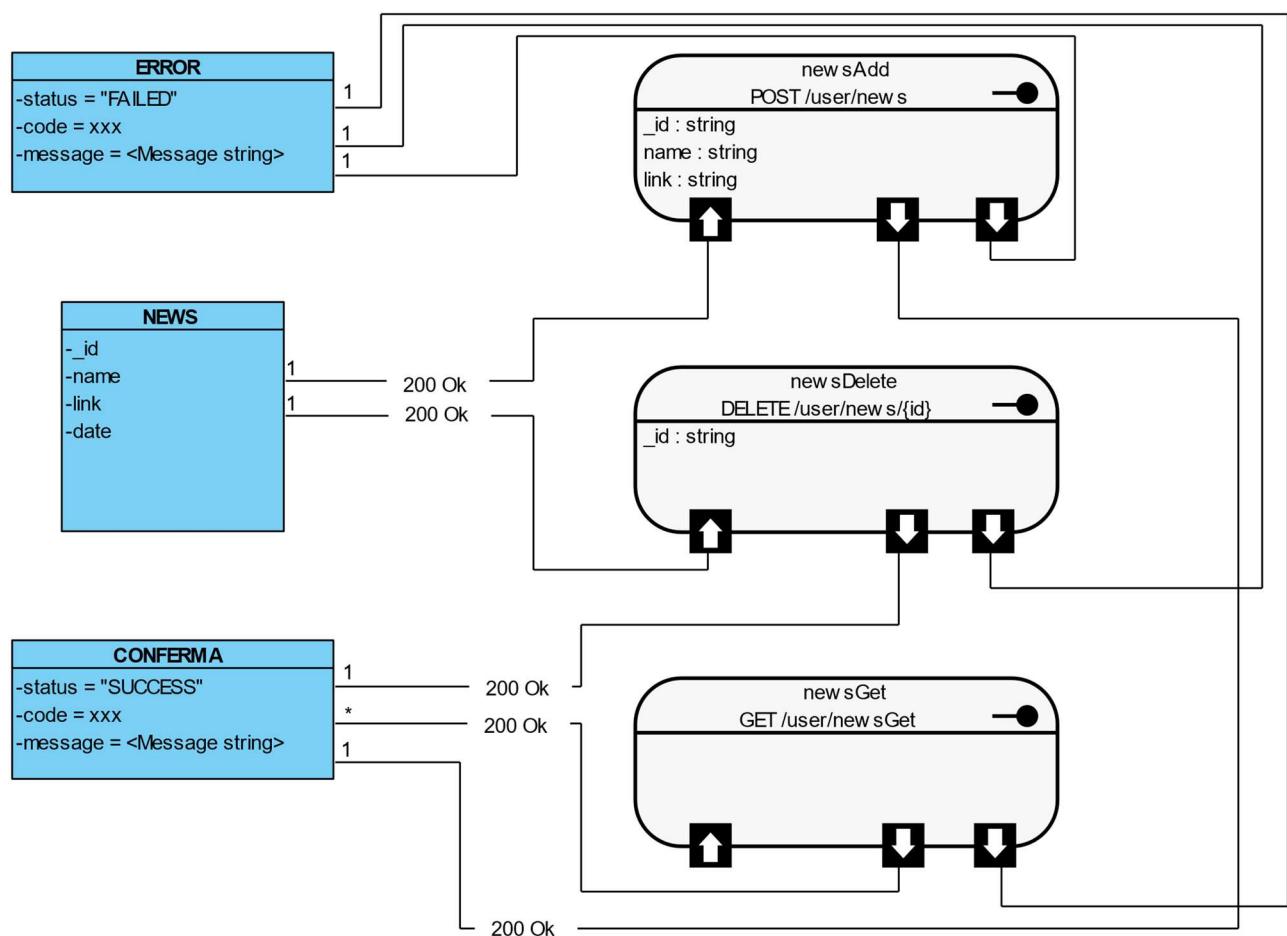
Ogni API è stata descritta con i suoi input e output. Le API possono generare errori di diversa natura, quindi invece di definire un output per ogni possibile errore, è stato creato un output comune per tutti gli errori. In tutte le API, il messaggio di errore è standardizzato: il campo "code", che è diverso per ogni API, il campo "message" che descrive la natura dell'errore e il codice di stato che identifica l'errore corrispondente.

Resource Diagram 1



Resource Diagram 2**Resource Diagram 3**

Resource Diagram 4**Resource Diagram 5**

Resource Diagram 6

8 Sviluppo API

Di seguito verranno descritti i funzionamenti delle API sviluppate nel progetto e le varie funzioni di MongoDB chiamate al loro interno.

Le immagini di ogni API non saranno inserite nel documento, in quanto troppo lungo, ma per chiarezza aggiuntive il codice si trova nella repository [/G45_project/backend/api/](#) e sono contenute nel file “**User.js**”.

Il codice iniziale del file “**User.js**” rappresenta l'inizializzazione di un router Express per gestire le richieste HTTP all'interno di un'applicazione Node.js.

Vengono importati i modelli MongoDB necessari per interagire con il database, come User, UserVerification, PasswordReset, Budget e Transaction.

Inoltre, vengono importati i moduli **Nodemailer** e **uuid** per gestire l'invio di email e la generazione di ID univoci, rispettivamente.

Le credenziali di autenticazione per l'invio delle email vengono recuperate dalle variabili d'ambiente tramite il modulo dotenv.

Viene configurato un trasportatore Nodemailer per l'invio delle email tramite il servizio Gmail, utilizzando le credenziali fornite.

Infine, viene verificata la connessione del trasportatore Nodemailer per garantire che sia pronto per l'invio delle email.

```

1  const express = require(`express`);
2  const router = express.Router();
3
4  // Import the MongoDB user model so we can use it now
5  const User = require(`./../models/User`);
6
7  // Import the MongoDB user verification model so we can use it now
8  const UserVerification = require(`./../models/UserVerification`);
9
10 // Import the MongoDB password reset model so we can use it now
11 const PasswordReset = require(`./../models/PasswordReset`);
12
13 //Import the MongoDB Budget model so we can use it now
14 const Budget = require(`./../models/Budget`);
15
16 //Import the MongoDB transaction model so we can use it now
17 const Transaction = require(`./../models/Transaction`);
18
19 //Import the email handler from node package (nodemailer)
20 const nodemailer = require(`nodemailer`);
21
22 //Imprt the uuid from node package (uuid)
23 const {v4: uuidv4} = require(`uuid`);
24
25 //env credentials
26 require(`dotenv`).config();
27
28
29 // Password handler
30 const bcrypt = require(`bcrypt`);
31 const saltRounds = 10; // Moved saltRounds outside the function
32
33 //path for static verified page
34 const path = require(`path`);
35 const {error} = require("console");
36 const exp = require("constants");
37
38 //nodemailer stuff (transporter)
39 let transporter = nodemailer.createTransport({
40     service: "gmail",
41     host: 'smtp.gmail.com',
42     port: 465,
43     secure: true,
44     auth: {
45         user: process.env.AUTH_EMAIL,
46         pass: process.env.AUTH_PASS,
47     }
48 });
49
50 //testing success
51 transporter.verify((error, success) => {
52     if (error) {
53         console.log(error);
54     } else {
55         console.log("Ready for message");
56         console.log(success);
57     }
58 });

```

API per il modello User

Di seguito descriviamo le API per gestire le varie azioni eseguibili sulla risorsa Utente

POST (“/signup”):

- Questa API rappresenta un endpoint per la procedura di registrazione degli utenti all'interno del sistema. Estraie i parametri fondamentali, come il nome, l'email, la password e la data di nascita, dal corpo della richiesta HTTP. Previa esecuzione di operazioni di sanitizzazione, quali la rimozione degli spazi bianchi, procede alla validazione dei dati in input.
- Successivamente, avvia una serie di controlli per verificare la coerenza dei dati forniti. Si assicura che nessun campo obbligatorio sia vuoto e che i parametri soddisfino i criteri di conformità definiti. Questa fase include la verifica del formato dell'email, la validità della data di nascita e la lunghezza adeguata della password.
- Dopo la convalida dei dati, avvia una interrogazione al database per determinare se l'email fornita sia già associata a un utente registrato. Nel caso in cui l'email sia già presente nel sistema, l'API restituisce una risposta negativa.
- Qualora l'email sia nuova e non risulti registrata, procede alla crittografia della password utilizzando l'algoritmo bcrypt. Successivamente, crea un nuovo documento utente con i dati forniti e lo persiste nel database.
- In seguito, invia un'email di verifica all'indirizzo specificato dall'utente per completare la procedura di registrazione. Se la registrazione e l'invio dell'email di verifica hanno successo, l'API rilascia una risposta affermativa.
- In presenza di errori durante la registrazione, l'API rilascia risposte di errore specifiche che indicano la natura del problema riscontrato, fornendo così una comunicazione accurata e dettagliata con l'utente.

GET(“/verify/:userID/uniqueString”)

- L'API sendVerificationEmail utilizza la libreria Nodemailer per gestire l'invio di email di verifica agli utenti. Questa libreria offre funzionalità integrate per l'invio di email tramite protocolli come SMTP, rendendo più semplice l'invio di email da parte dell'applicazione. La funzione sendVerificationEmail è stata costruita utilizzando Nodemailer per inviare l'email di verifica contenente un link univoco all'utente appena registrato.
- Il funzionamento dell'API sendVerificationEmail può essere descritto nel seguente modo:
 - 1) Viene creato un URL di base che sarà incluso nell'email di verifica. Questo URL è costruito utilizzando l'indirizzo del server locale e l'endpoint di verifica (/verify).
 - 2) Viene generato un valore univoco combinando l'ID utente fornito e un valore generato casualmente utilizzando la funzione `uuidv4()` della libreria `uuid`. Questo valore univoco sarà incluso nel link di verifica.
 - 3) Il valore univoco viene quindi hashato utilizzando la funzione `bcrypt.hash()` per garantire la sicurezza del link di verifica.
 - 4) Viene creato un oggetto `mailOptions` che contiene le informazioni necessarie per inviare l'email di verifica. Questo include il mittente, il destinatario, l'oggetto dell'email e l'HTML contenente il link di verifica.
 - 5) Viene creato un nuovo record di verifica utente nella collezione `UserVerification` del database, contenente l'ID utente e il valore univoco hashato, insieme ad altri dettagli come la data di creazione e di scadenza.

- 6) Viene utilizzata la funzione transporter.sendMail() di Nodemailer per inviare l'email di verifica utilizzando le informazioni specificate in mailOptions.
- L'API verify è responsabile del processo di verifica effettivo dell'account utente. Quando viene chiamato, ricerca nel database per verificare se esiste un record di verifica utente corrispondente all'ID utente e al valore univoco forniti. Se il record viene trovato e non è scaduto, l'utente viene contrassegnato come verificato nel database e il record di verifica viene eliminato. Viene quindi restituita una risposta di successo con uno stato 200 e un messaggio di conferma.
 - Se il record di verifica non viene trovato, se è scaduto o se il valore univoco fornito non corrisponde a quello nel database, viene restituita una risposta di errore con un messaggio appropriato e uno stato 404.
 - L'API sendVerificationEmail non richiede un diagramma delle risorse separato in quanto è strettamente legata all'API verify e serve a facilitare il processo di verifica dell'account utente.

GET("/verified")

- L'API GET /verified è responsabile della gestione della richiesta di visualizzazione della pagina di verifica dell'account utente. Essa viene chiamata quando un utente desidera visualizzare la pagina di conferma dell'avvenuta verifica dell'account.
- Il funzionamento di questa API può essere descritto nel seguente modo:
 - Quando viene effettuata una richiesta GET a /verified, l'API risponde inviando il file HTML corrispondente alla pagina di verifica dell'account utente.
 - Utilizza la funzione res.sendFile() per inviare il file HTML verified.html presente nella directory delle views del server.
 - Il percorso del file HTML viene costruito utilizzando la funzione path.join() per garantire la corretta risoluzione del percorso, specificando __dirname come riferimento al percorso assoluto della directory corrente e aggiungendo il percorso relativo della cartella views e del file verified.html.
 - Una volta che il file HTML è stato inviato con successo, la pagina di verifica dell'account utente viene visualizzata sul browser dell'utente che ha effettuato la richiesta GET a /verified.

POST ("signin")

- L'API signin è responsabile della gestione dell'accesso degli utenti al sistema. Essa riceve le credenziali dell'utente tramite una richiesta HTTP POST e verifica se corrispondono a un utente registrato nel sistema.
- Il funzionamento di questa API può essere descritto nel seguente modo:
 - Le credenziali dell'utente, ovvero l'email e la password, vengono estratte dal corpo della richiesta HTTP e vengono successivamente sottoposte ad una fase di trim per rimuovere eventuali spazi vuoti.
 - Se uno o entrambi i campi email e password risultano essere vuoti, l'API restituisce una risposta con uno stato "FAILED" e un messaggio che indica la presenza di credenziali vuote.
 - Nel caso in cui entrambi i campi siano popolati, l'API procede con la verifica dell'esistenza dell'utente nel database utilizzando l'email fornita.
 - Se l'utente viene individuato nel database, l'API verifica se l'account associato è stato precedentemente verificato. Nel caso contrario, viene restituita una risposta "FAILED" con un messaggio che segnala la necessità di completare il processo di verifica dell'email.
 - Se l'account è stato verificato, l'API confronta la password fornita dall'utente con quella memorizzata nel database, utilizzando la funzione bcrypt.compare() per confrontare le password hashate.

- Se la password fornita coincide con quella memorizzata nel database, l'API restituisce una risposta "SUCCESS" indicando l'avvenuta autenticazione, fornendo inoltre i dati dell'utente.
- Nel caso in cui la password fornita non coincida con quella memorizzata nel database, l'API restituisce una risposta "FAILED" con un messaggio che informa l'utente dell'invalidità della password inserita.
- Se l'email fornita non corrisponde a nessun utente registrato nel database, viene restituita una risposta "FAILED" con un messaggio che segnala la mancata corrispondenza delle credenziali inserite.
- Infine, se si verifica un errore durante il processo di verifica dell'utente nel database, l'API restituisce una risposta "FAILED" con un messaggio che indica l'accadimento di un errore durante tale operazione.

POST("/requestPasswordReset")

- L'API POST /requestPasswordReset gestisce la richiesta di reimpostazione della password da parte dell'utente. Questo endpoint viene chiamato quando un utente desidera reimpostare la propria password e invia una richiesta POST contenente l'email associata all'account e l'URL di reindirizzamento per il reset della password.
- Il funzionamento di questa API può essere descritto come segue:
 - Riceve i dati della richiesta POST contenenti l'email dell'utente e l'URL di reindirizzamento per il reset della password.
 - Verifica se l'email fornita esiste nel database. Se non esiste, restituisce una risposta con stato "FAILED" e un messaggio indicante che non esiste un account associato all'email fornita.
 - Verifica se l'utente associato all'email è già stato verificato. Se l'utente non è stato verificato, restituisce una risposta con stato "FAILED" e un messaggio che indica che l'email non è stata ancora verificata e l'utente deve controllare la propria casella di posta.
 - Se l'utente esiste ed è verificato, chiama la funzione sendResetEmail() per inviare l'email di reimpostazione della password.
- La funzione sendResetEmail() si occupa di inviare l'email di reimpostazione della password e gestire i record di reset della password nel database. Ecco come funziona:
 - Genera una stringa univoca di reimpostazione della password combinando un UUID generato casualmente con l'ID utente associato all'account.
 - Cancella tutti i record di reset della password esistenti associati all'utente nel database.
 - Hasha la stringa di reimpostazione della password e salva i nuovi dati di reset della password nel database.
 - Invia un'email all'utente contenente un link per il reset della password. L'email include l'URL di reindirizzamento specificato nella richiesta POST, insieme alla stringa univoca di reimpostazione della password.
 - Restituisce una risposta con stato "PENDING" e un messaggio che conferma l'invio dell'email di reimpostazione della password.
- In caso di errori durante il processo di reset della password, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati.

POST (“/resetPassword”)

- L'API POST /resetPassword gestisce il reset della password per un utente. Questo endpoint viene chiamato quando un utente desidera reimpostare la propria password dopo aver ricevuto un'email di reset. La richiesta POST include l'ID utente, la stringa di reset e la nuova password.
- Il funzionamento di questa API può essere descritto come segue:
 - Riceve i dati della richiesta POST contenenti l'ID utente, la stringa di reset e la nuova password.
 - Verifica se esiste un record di reset della password nel database associato all'ID utente fornito. Se non esiste, restituisce una risposta con stato "FAILED" e un messaggio indicante che la richiesta di reset della password non è stata trovata.
 - Se il record di reset della password esiste, verifica se la stringa di reset non è scaduta confrontando la data corrente con la data di scadenza memorizzata nel record. Se la stringa è scaduta, elimina il record di reset della password e restituisce una risposta con stato "FAILED" e un messaggio che indica che il link di reset della password è scaduto.
 - Se la stringa di reset è valida, confronta la stringa fornita nella richiesta con la stringa hashata memorizzata nel record di reset della password. Se le stringhe corrispondono, hasha la nuova password e aggiorna la password dell'utente nel database.
 - Dopo aver aggiornato con successo la password dell'utente, elimina il record di reset della password dal database.
 - Restituisce una risposta con stato "SUCCESS" e un messaggio che conferma il reset della password avvenuto con successo.
- In caso di errori durante il processo di reset della password, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati.

POST (“/budgetAdd”)

- L'API POST /budgetAdd gestisce l'aggiunta di un nuovo budget per un determinato utente. Questo endpoint viene chiamato quando un utente desidera creare un nuovo budget. La richiesta POST include l'ID dell'utente, il nome del budget, l'importo previsto e una descrizione facoltativa.
- Il funzionamento di questa API può essere descritto come segue:
 - Riceve i dati della richiesta POST contenenti l'ID dell'utente, il nome del budget, l'importo previsto e la descrizione.
 - Verifica che il nome e la descrizione non siano vuoti e che l'importo sia un valore positivo. Se uno di questi controlli non viene superato, restituisce una risposta con stato "FAILED" e un messaggio appropriato, incluso un codice di errore specifico.
 - Verifica se esiste già un budget con lo stesso nome per lo stesso utente nel database. Se esiste, restituisce una risposta con stato "FAILED" e un messaggio che indica che un budget con lo stesso nome già esiste, insieme a un codice di errore specifico.
 - Se il nome del budget è unico, crea un nuovo budget con i dati forniti e lo salva nel database.
 - Se il salvataggio nel database è riuscito, restituisce una risposta con stato "SUCCESS", un messaggio di conferma e i dati del budget appena creato.
 - In caso di errori durante il processo di creazione del budget, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati, insieme a codici di errore specifici.

DELETE (“/budgetDelete/:id”)

- L'API DELETE /budgetDelete/:id gestisce l'eliminazione di un budget esistente tramite il suo ID. Questo endpoint viene chiamato quando un utente desidera eliminare un budget specifico. La richiesta DELETE include l'ID del budget da eliminare come parametro nella URL.
- Il funzionamento di questa API può essere descritto come segue:
 - Estraie l'ID del budget dall'URL dei parametri della richiesta.
 - Verifica se l'ID del budget è valido utilizzando il metodo isValid di Mongoose. Se l'ID non è valido, restituisce una risposta con stato "FAILED" e un messaggio che indica un ID di budget non valido, insieme a un codice di errore specifico.
 - Cerca il budget nel database utilizzando il metodo findByIdAndDelete di Mongoose. Se il budget non viene trovato, restituisce una risposta con stato "FAILED" e un messaggio che indica che il budget non è stato trovato, insieme a un codice di errore specifico.
 - Se il budget viene trovato e eliminato con successo, restituisce una risposta con stato "SUCCESS", un messaggio di conferma e i dati del budget eliminato.
 - In caso di errori durante il processo di eliminazione del budget, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati, insieme a codici di errore specifici.

GET (“/budgetGet/:id”)

- L'API GET /budgetGet/:id gestisce il recupero di un budget esistente tramite il suo ID. Questo endpoint viene chiamato quando un utente desidera ottenere i dettagli di un budget specifico. La richiesta GET include l'ID del budget da recuperare come parametro nell'URL.
- Il funzionamento di questa API può essere descritto come segue:
 - Estraie l'ID del budget dall'URL dei parametri della richiesta.
 - Verifica se l'ID del budget è valido utilizzando il metodo isValid di Mongoose. Se l'ID non è valido, restituisce una risposta con stato "FAILED" e un messaggio che indica un ID di budget non valido, insieme a un codice di errore specifico.
 - Cerca il budget nel database utilizzando il metodo findById di Mongoose. Se il budget non viene trovato, restituisce una risposta con stato "FAILED" e un messaggio che indica che il budget non è stato trovato, insieme a un codice di errore specifico.
 - Se il budget viene trovato con successo, restituisce una risposta con stato "SUCCESS", un messaggio di conferma e i dati del budget trovato.
 - In caso di errori durante il processo di recupero del budget, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati, insieme a codici di errore specifici.

GET (“/getBudgetsByUserID/:id”)

- L'API GET /getBudgetsByUserID/:id gestisce il recupero dei budget associati a un determinato utente, identificato dal suo ID. Questo endpoint viene chiamato quando un utente desidera ottenere tutti i budget associati al proprio account. La richiesta GET include l'ID dell'utente come parametro nell'URL.
- Il funzionamento di questa API può essere descritto come segue:
 - Estraie l'ID dell'utente dall'URL dei parametri della richiesta.
 - Verifica se l'ID dell'utente è valido utilizzando il metodo isValid di Mongoose. Se l'ID non è valido, viene restituita una risposta con stato "FAILED" e un messaggio che indica un ID utente non valido, insieme a un codice di errore specifico.
 - Cerca i budget associati all'ID dell'utente nel database utilizzando il metodo find di Mongoose. Se non vengono trovati budget associati all'utente, viene restituita una risposta con stato "FAILED" e un messaggio che indica che i budget non sono stati trovati, insieme a un codice di errore specifico.

- Se vengono trovati budget associati all'utente, viene restituita una risposta con stato "SUCCESS", un messaggio di conferma e i dati dei budget trovati.
- In caso di errori durante il processo di recupero dei budget, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati, insieme a codici di errore specifici.

PATCH("/budgetUpdate")

- L'API PATCH /budgetUpdate/:budgetID gestisce l'aggiornamento di un budget esistente, identificato dal suo ID. Questo endpoint viene chiamato quando un utente desidera modificare il valore corrente di un budget specifico. La richiesta PATCH include l'ID del budget da aggiornare come parametro nell'URL e il nuovo valore da aggiungere nel corpo della richiesta.
- Il funzionamento di questa API può essere descritto come segue:
 - Estraе l'ID del budget e il valore da aggiungere dalla richiesta.
 - Verifica che il valore da aggiungere sia valido e un numero utilizzando il metodo isNaN. Se il valore non è valido, viene restituita una risposta con stato "FAILED" e un messaggio che indica un valore non valido, insieme a un codice di errore specifico.
 - Cerca il budget nel database utilizzando l'ID fornito utilizzando il metodo findById di Mongoose. Se il budget non viene trovato, viene restituita una risposta con stato "FAILED" e un messaggio che indica che il budget non è stato trovato, insieme a un codice di errore specifico.
 - Se il budget viene trovato, aggiorna il campo "current" del budget aggiungendo il valore fornito.
 - Salva le modifiche nel database utilizzando il metodo save. Se le modifiche vengono salvate con successo, viene restituita una risposta con stato "SUCCESS", un messaggio di conferma e i dati del budget aggiornato.
 - In caso di errori durante il processo di aggiornamento del budget, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati, insieme a codici di errore specifici.

PATCH ("budgetAfterdeleteTransUpdate/:budgetID")

- L'API PATCH /budgetAfterdeleteTransUpdate/:budgetID gestisce l'aggiornamento del campo "current" di un budget dopo l'eliminazione di una transazione associata. Questo endpoint viene chiamato quando una transazione viene eliminata e il suo valore deve essere sottratto dal campo "current" del budget corrispondente.
- Il funzionamento di questa API può essere descritto come segue:
 - Estraе l'ID del budget e il valore da sottrarre dalla richiesta.
 - Verifica che il valore da sottrarre sia valido e un numero utilizzando il metodo isNaN. Se il valore non è valido, viene restituita una risposta con stato "FAILED" e un messaggio che indica un valore non valido, insieme a un codice di errore specifico.
 - Cerca il budget nel database utilizzando l'ID fornito utilizzando il metodo findById di Mongoose. Se il budget non viene trovato, viene restituita una risposta con stato "FAILED" e un messaggio che indica che il budget non è stato trovato, insieme a un codice di errore specifico.
 - Se il budget viene trovato, aggiorna il campo "current" del budget sottraendo il valore fornito.

- Salva le modifiche nel database utilizzando il metodo save. Se le modifiche vengono salvate con successo, viene restituita una risposta con stato "SUCCESS", un messaggio di conferma e i dati del budget aggiornato.
- In caso di errori durante il processo di aggiornamento del budget, vengono restituite risposte con stato "FAILED" e messaggi di errore appropriati, insieme a codici di errore specifici.

POST ("*/transactionAdd*")

- L'endpoint API POST /transactionAdd gestisce l'aggiunta di nuove transazioni associate a specifici budget. Qui di seguito viene descritto il suo funzionamento:
 - Estrazione dei dettagli della transazione: I dettagli della transazione, inclusi budgetID, name, amount e description, vengono estratti dal corpo della richiesta HTTP.
 - Normalizzazione dei dati: Eventuali spazi bianchi nei campi name e description vengono rimossi.
 - Validazione dei dati: Vengono eseguite diverse verifiche sui dati ricevuti:
 - Verifica che il campo name non sia vuoto.
 - Verifica che l'amount sia un valore positivo.
 - Verifica che il budgetID sia un identificatore valido e non vuoto.
 - Gestione delle eccezioni: Se una delle verifiche fallisce, viene restituita una risposta con uno stato "FAILED", un codice di errore specifico e un messaggio descrittivo che indica il problema riscontrato.
 - Creazione della transazione: Se tutti i controlli hanno esito positivo, viene creata un'istanza di Transaction con i dettagli forniti e salvata nel database utilizzando il metodo save().
 - Risposta di conferma: Se il salvataggio nel database ha successo, viene restituita una risposta con stato "SUCCESS", un codice di conferma e i dettagli della transazione appena creata.
 - Gestione degli errori: Se si verifica un errore durante il salvataggio della transazione nel database, viene restituita una risposta con stato "FAILED", un codice di errore specifico e un messaggio che indica il fallimento dell'operazione.

DELETE ("*/transactionDelete/:id*")

- L'endpoint API DELETE /transactionDelete/:id gestisce l'eliminazione di transazioni esistenti in base al loro ID. Di seguito viene fornita una descrizione del funzionamento di questo endpoint:
 - Estrazione dell'ID della transazione: L'ID della transazione da eliminare viene estratto dai parametri della richiesta HTTP.
 - Validazione dell'ID della transazione: Viene verificato se l'ID della transazione è un identificatore valido utilizzando la funzione mongoose.Types.ObjectId.isValid(). Se l'ID non è valido, viene restituita una risposta con stato "FAILED", un codice di errore specifico e un messaggio che indica che l'ID della transazione non è valido.
 - Ricerca e eliminazione della transazione: Viene effettuata una ricerca nel database della transazione corrispondente all'ID fornito utilizzando il metodo findByIdAndDelete(). Se la transazione viene trovata e eliminata con successo, viene restituita una risposta con stato "SUCCESS", un codice di conferma e i dettagli della transazione eliminata.
 - Gestione degli errori: Se si verifica un errore durante il processo di ricerca ed eliminazione della transazione, viene restituita una risposta con stato "FAILED", un codice di errore specifico e un messaggio che indica il fallimento dell'operazione, insieme all'eventuale messaggio di errore dettagliato.

GET (“/transactionGet/:id”)

- L'endpoint API GET /transactionGet/:id consente di recuperare le informazioni relative a una transazione specifica in base al suo ID dell'utente. Ecco una descrizione dettagliata del funzionamento di questo endpoint:
 - Estrazione dell'ID della transazione: L'ID della transazione da recuperare viene estratto dai parametri della richiesta HTTP.
 - Validazione dell'ID della transazione: Viene verificato se l'ID della transazione è un identificatore valido utilizzando la funzione mongoose.Types.ObjectId.isValid(). Se l'ID non è valido, viene restituita una risposta con stato "FAILED", un codice di errore specifico e un messaggio che indica che l'ID della transazione non è valido.
 - Ricerca della transazione nel database: Viene effettuata una ricerca nel database delle transazioni che corrispondono all'ID del budget fornito utilizzando il metodo find(). Se viene trovata almeno una transazione, viene restituita una risposta con stato "SUCCESS", un codice di conferma e i dettagli delle transazioni trovate.
 - Gestione degli errori: Se si verifica un errore durante il processo di ricerca delle transazioni, viene restituita una risposta con stato "FAILED", un codice di errore specifico e un messaggio che indica il fallimento dell'operazione, insieme all'eventuale messaggio di errore dettagliato.

POST (“/news”)

- L'endpoint API POST /news consente di aggiungere una nuova notizia al sistema. Di seguito è riportata una descrizione del suo funzionamento:
 - Estrazione dei dati dalla richiesta: I dati della nuova notizia, come il nome e il link, vengono estratti dal corpo della richiesta HTTP.
 - Validazione dei dati: Viene verificato che il nome e il link della notizia siano forniti nella richiesta. Se uno o entrambi i campi sono mancanti, viene restituita una risposta con uno stato 400 (Bad Request) e un messaggio di errore che indica che il nome e il link sono obbligatori.
 - Creazione e salvataggio della notizia: Viene creata una nuova istanza di News utilizzando i dati estratti dalla richiesta. Successivamente, la nuova notizia viene salvata nel database utilizzando il metodo save().
 - Risposta con successo: Se la notizia viene salvata con successo nel database, viene restituita una risposta con stato 201 (Created) contenente i dettagli della notizia aggiunta.
 - Gestione degli errori: Se si verifica un errore durante il processo di aggiunta della notizia, l'errore viene passato alla funzione next() per la gestione centralizzata degli errori. La gestione degli errori può essere configurata per restituire una risposta appropriata all'utente finale.

DELETE (“/news/:id”)

- L'endpoint API DELETE /news/:id consente di eliminare una notizia dal sistema. Ecco una descrizione del suo funzionamento:
 - Identificazione della notizia da eliminare: L'ID della notizia da eliminare viene estratto dai parametri della richiesta HTTP.

- Eliminazione della notizia: Viene effettuata una query per trovare la notizia nel database utilizzando l'ID fornito. Se la notizia viene trovata, viene eliminata utilizzando il metodo `findByIdAndDelete()` di Mongoose.
- Gestione della notizia non trovata: Se la notizia non viene trovata nel database (ovvero se la query non restituisce alcun risultato), viene restituita una risposta con uno stato 404 (Not Found) e un messaggio di errore che indica che la notizia non è stata trovata.
- Risposta con successo: Se la notizia viene eliminata con successo dal database, viene restituita una risposta con uno stato 200 (OK) e un messaggio che conferma che la notizia è stata eliminata con successo.
- Gestione degli errori: Se si verifica un errore durante il processo di eliminazione della notizia, l'errore viene passato alla funzione `next()` per la gestione centralizzata degli errori. La gestione degli errori può essere configurata per restituire una risposta appropriata all'utente finale.

9 API documentazione

Le API locali dell'applicazione BUDGETpal sono state documentate tramite l'utilizzo del modulo Swagger UI Express di NodeJS. Questa documentazione fornisce una panoramica esaustiva delle API, rendendo accessibili tutte le informazioni pertinenti ai potenziali utenti. Gli sviluppatori possono accedere alla documentazione tramite un endpoint specifico, come ad esempio `/api-docs`.

Nel caso in cui venga scaricata la repository **G45_project** e si crei il server locale sulla propria macchina, l'endpoint da invocare per raggiungere la documentazione è:

<http://localhost:4500/api-docs/>

Le API inizialmente vengono mostrate in sottogruppi suddivisi secondo la loro logica di utilizzo, la rappresentazione quindi è la seguente:

The screenshot shows the Swagger UI interface for the G45 project's API documentation. At the top, there's a dark header bar with the "Swagger" logo and the text "Documentazione delle API del progetto G45". Below this is a light-colored main content area. In the top left of the content area, there's a "Servers" dropdown menu set to "http://localhost:4500/". The main content area is titled "API Documentation" with version "1.0.0" and "OAS 3.0" status indicators. Below the title, there's a sub-header "Documentazione delle API del progetto G45". The main content is organized into four main categories: "USER", "BUDGET", "TRANSACTIONS", and "NEWS", each with a small downward arrow icon to its right, indicating they can be expanded.

Selezionando uno dei Tag possibili vengono mostrate le API presenti nel rispettivo sottogruppo. Le API implementate sono di quattro tipologie: **POST, GET, DELETE, PATCH**.

POST: viene utilizzato per inviare dati al server per creare una nuova risorsa.

GET: viene utilizzato per richiedere dati dal server e ottenere informazioni sulle risorse esistenti.

DELETE: viene utilizzato per richiedere al server l'eliminazione di una risorsa specifica.

PATCH: viene utilizzato per apportare modifiche parziali a una risorsa esistente.

Di seguito un esempio di rappresentazione di alcune **API** del modello **BUDGET**:

The screenshot shows a list of API endpoints for the BUDGET category. Each endpoint is represented by a row with a color-coded button for the method (POST, DELETE, GET, PATCH) and a link to the endpoint description. The descriptions provide a brief overview of the functionality.

BUDGET	
POST	/user/budgetAdd Crea un nuovo budget.
DELETE	/user/budgetDelete/{id} Elimina un budget esistente
GET	/user/budgetGet/{id} Ottiene un budget dal database.
GET	/user/getBudgetsByUserID/{id} Ottiene i budget per un determinato utente
PATCH	/user/budgetUpdate/{budgetID} Aggiorna il valore di un budget in positivo
PATCH	/user/budgetAfterDeleteTransUpdate/{budgetID} Aggiorna il valore di un budget in negativo

Tutti le API sono ben descritte con il loro endpoint e la loro funzionalità, specificando eventuali errori e stati di ritorno.

Presentano ognuna di esse una descrizione breve della loro funzionalità e successivamente forniscono il tipo di dato che si aspettano in input, associando anche un esempio valido di possibili valori in input. Nella parte finale sono elencati tutti i vari codici di ritorno possibili della API. Un esempio è il seguente **GET (/getBudgetsByUserID/{id})**:

This screenshot shows the detailed description of the **GET /user/getBudgetsByUserID/{id}** API endpoint. It includes the method, endpoint, description, parameters, and a 'Try it out' button.

GET /user/getBudgetsByUserID/{id} Ottiene i budget per un determinato utente

Ottiene i budget associati all'ID dell'utente specificato.

Parameters

Name	Description
id <small>* required</small>	ID dell'utente per cui ottenere i budget <i>Example : 5fc8e9a85af3950017c6d2a2</i>

Try it out

GET /user/getBudgetsByUserID/{id} Ottiene i budget per un determinato utente

Ottiene i budget associati all'ID dell'utente specificato.

Parameters

Name **Description** **Try it out**

id <small>* required</small>	ID dell'utente per cui ottenere i budget. <i>Example : </i> 5fc8e9a85af3950017c6d2a2
<small>(path)</small>	5fc8e9a85af3950017c6d2a2

Responses

Code	Description	Links
200	Successo, restituisce i budget associati all'utente.	No links
	Media type	
	application/json	
	Controls Accept header.	
	Example Value Schema	
	<pre>{ "status": "SUCCESS", "code": 9, "message": "Budgets found successfully" }</pre>	
400	Errore dovuto a un ID utente non valido.	No links
	Media type	
	application/json	
	Example Value Schema	
	<pre>{ "status": "FAILED", "code": "9XX", "message": "Invalid user ID" }</pre>	
404	Nessun budget trovato per l'utente specificato.	No links
	Media type	
	application/json	
	Example Value Schema	
	<pre>{ "status": "FAILED", "code": "9XX", "message": "Budgets not found" }</pre>	
500	Errore del server durante la ricerca dei budget.	No links
	Media type	
	application/json	
	Example Value Schema	
	<pre>{ "status": "FAILED", "code": "9XX", "message": "An error occurred while retrieving the budgets", "error": "Internal Server Error" }</pre>	

10 Frontend Implementation

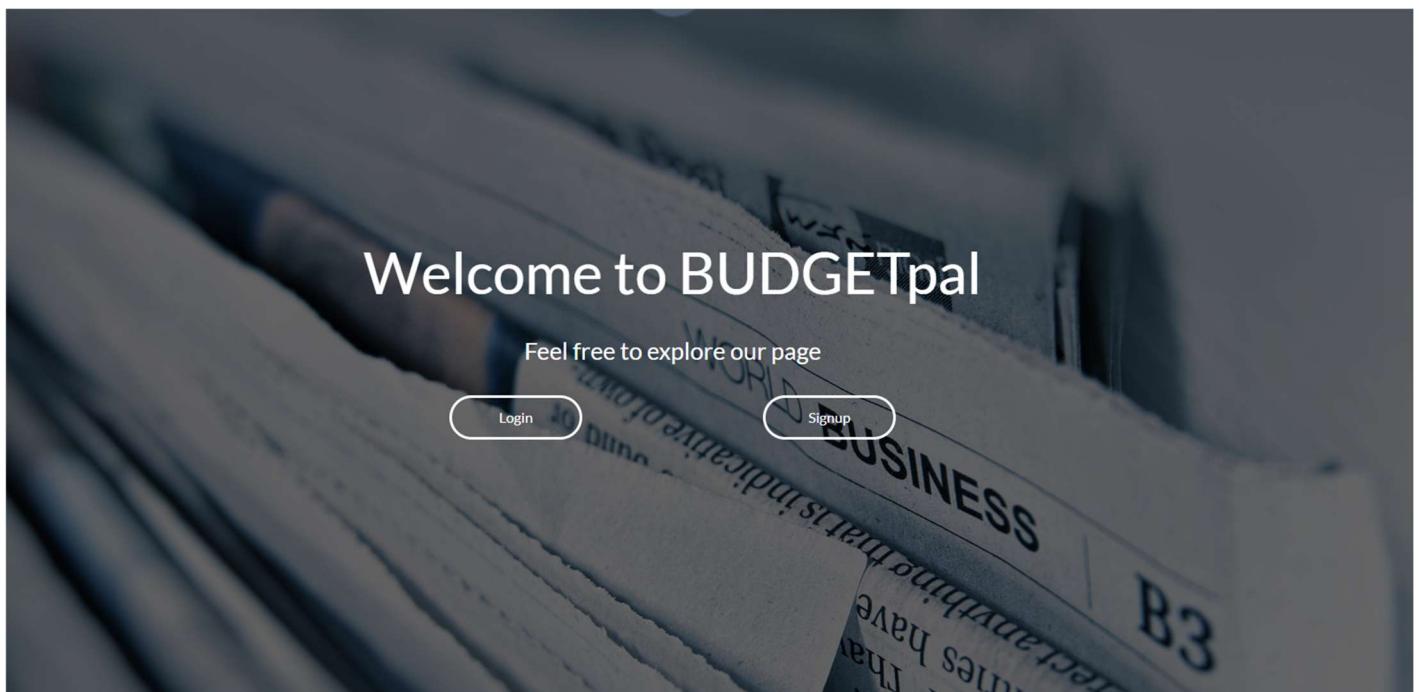
In questa sezione verranno descritte in dettaglio le schermate del frontend che sono state implementate con le rispettive azioni che possono essere eseguite su di esse.

Schermata iniziale

iniziale

La schermata iniziale, è la schermata attraverso la quale l'utente può accedere al suo account personale oppure nel caso non sia mai stato eseguito un accesso l'utente ha la possibilità di registrarsi, quindi sono presenti due bottoni che permettono di svolgere le due azioni.

Scegliendo arbitrariamente uno bottone, si aprirà un apposito form nel quale inserire le credenziali per accedere al sito web.



Schermata di registrazione

La schermata di registrazione presenta un formulario contenente campi per le credenziali personali dell'utente, tra cui **Full Name**, **Email Address**, **Date of Birth**, **Password** e **Repeat Password**.

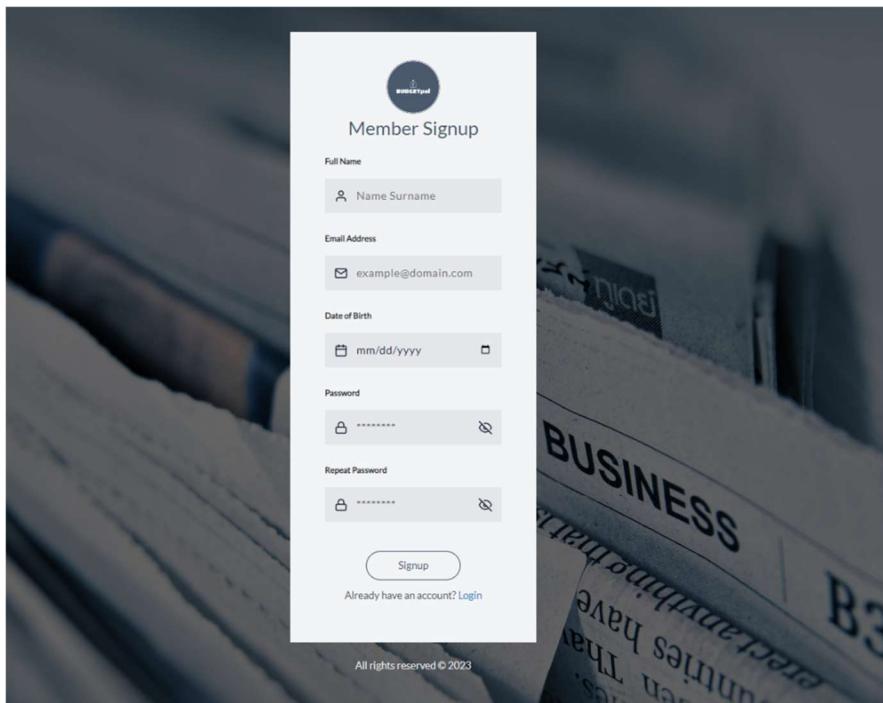
Ogni campo è soggetto a validazione e logica specifica che definisce i tipi di valori accettabili, l'obbligatorietà e l'unicità dei valori.

Le password devono rispettare una lunghezza minima di 8 caratteri, mentre l'email deve seguire uno standard specifico.

Il campo **Date of Birth** include un calendario a tendina per la selezione della data, mentre **Password** e **Repeat Password** sono dotati di mascheramento per la sicurezza.

In caso di errore, l'utente riceverà notifiche adeguate.

Alla compilazione del formulario, premendo il pulsante Signup, l'utente può creare l'account e sarà reindirizzato direttamente al formulario di login con l'**email** già compilata.



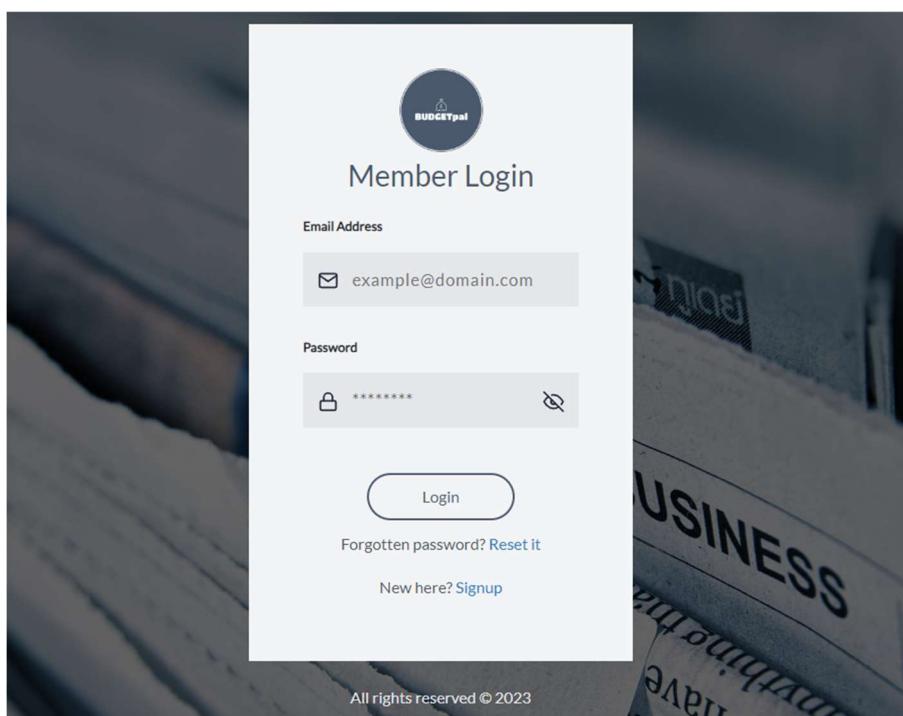
Schermata di login

La schermata di login presenta un formulario con campi per l'**email** e la **password** dell'utente.

La logica di frontend gestisce la validazione dei campi, assicurando che siano compilati correttamente prima di consentire il login. In caso di errori, vengono fornite notifiche appropriate all'utente.

Dopo aver inserito le credenziali e premuto il pulsante Login, l'applicazione verifica le credenziali dell'utente nel backend.

Se le credenziali sono corrette, l'utente viene autenticato e reindirizzato alla dashboard dell'applicazione. In caso contrario, viene visualizzato un messaggio di errore indicando che le credenziali non sono valide.



Schermata recupero Password

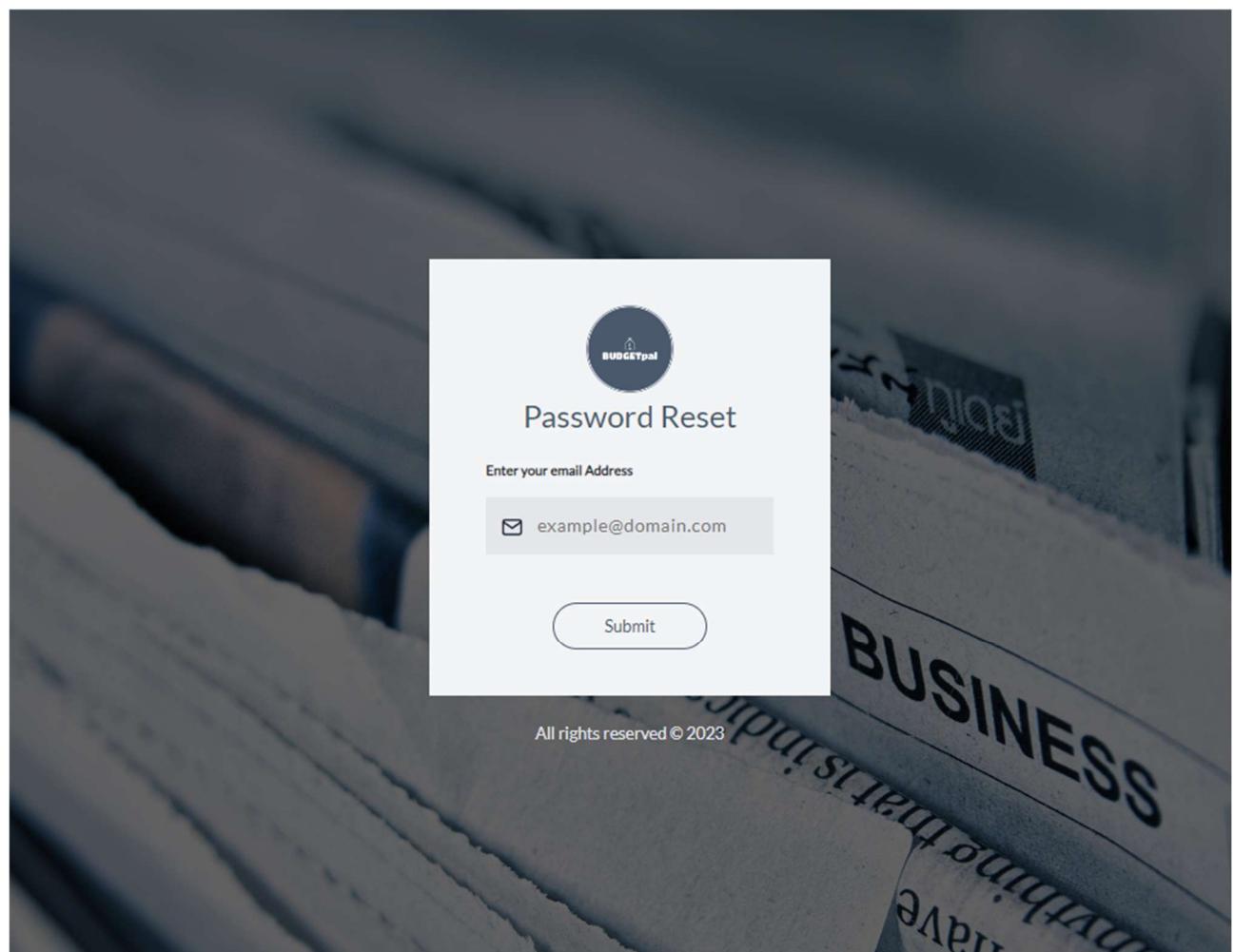
Nella schermata di **Login**, il form include un bottone "**Reset it**" posizionato nella parte inferiore. Questo bottone permette agli utenti di recuperare la propria password.

Cliccando su questo pulsante, gli utenti vengono reindirizzati a un altro form dove è possibile inserire l'email associata al proprio account.

La validazione dei campi nel **frontend** assicura che tutti i campi siano compilati correttamente prima di inviare l'email. In caso di errori, vengono fornite notifiche appropriate agli utenti, permettendo loro di correggere i campi necessari.

Sul lato **backend**, è implementata una logica denominata "**sendResetEmail**" che si occupa di inviare agli utenti un'email contenente un link per accedere al form dedicato al reset della password.

Questo link permette agli utenti di procedere con il reset della password in modo sicuro e protetto.

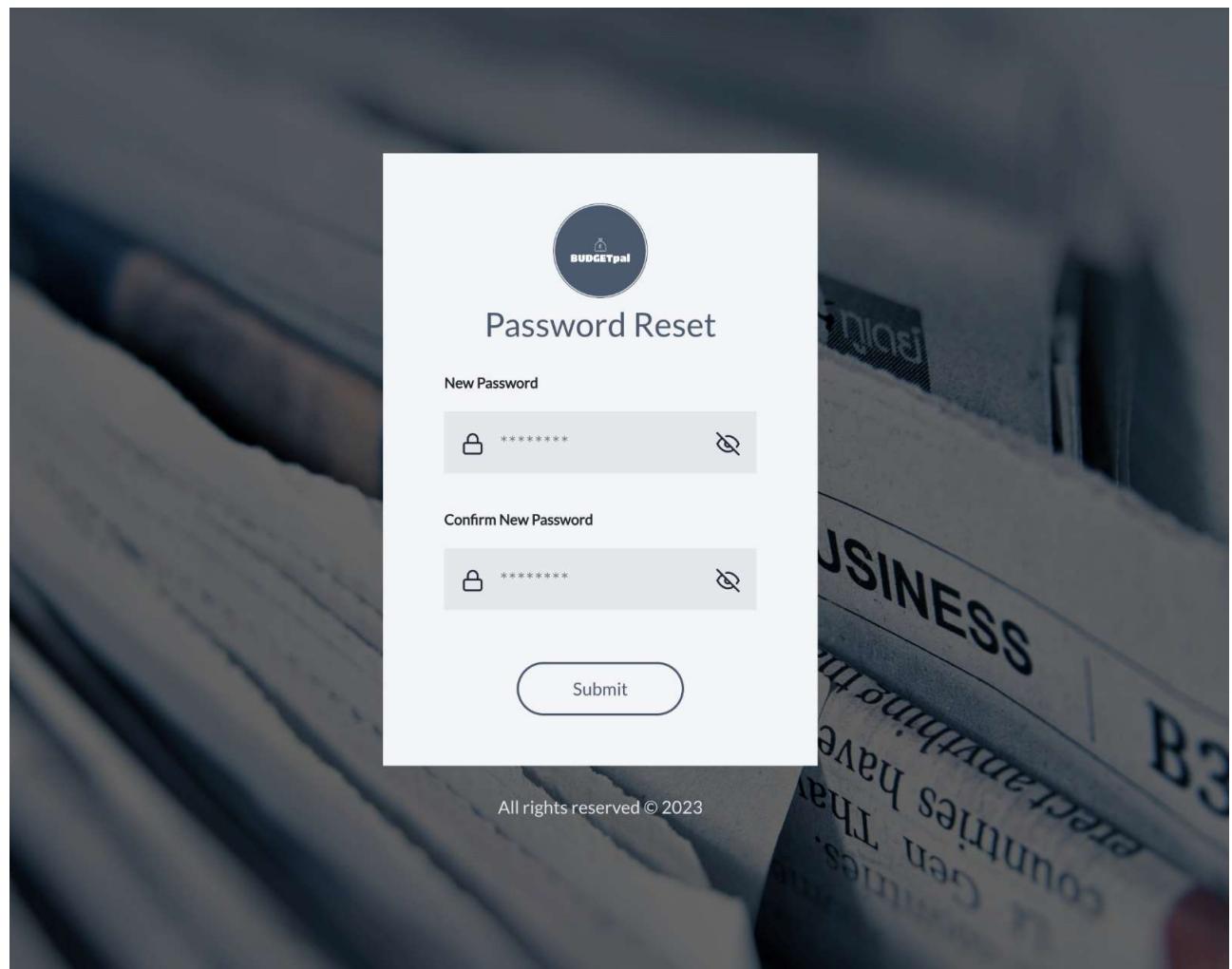


Schermata di Reset Password

La pagina di reset della password presenta due campi: "**Password**" e "**Repeat password**". Attraverso il frontend, viene eseguita la validazione dei campi per garantire che entrambi i campi siano compilati correttamente e che le password corrispondano.

Una volta che l'utente ha compilato correttamente i campi e ha inviato la richiesta di reset della password, la logica di backend si occupa di gestire questa richiesta. Questa logica invia una richiesta per cambiare la password nel database, utilizzando la nuova password fornita dall'utente.

Dopo aver cambiato con successo la password nel database, l'utente viene reindirizzato a una pagina di attesa. Durante questo periodo, il sistema verifica che la nuova password sia stata correttamente inserita nel database. Una volta confermato che la password è stata aggiornata con successo, all'utente viene data la possibilità di procedere ulteriormente premendo su "**Continua**". Questo permette all'utente di accedere nuovamente alla pagina di Login utilizzando la nuova password.



Schermata di gestione Budget

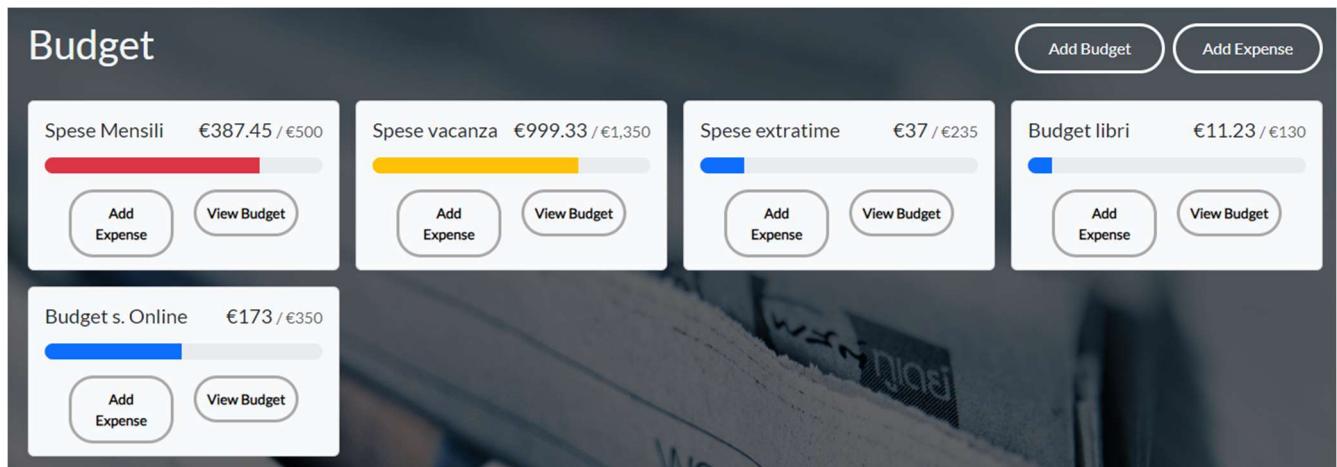
Nella piattaforma di **gestione dei Budget**, gli utenti hanno accesso alla visualizzazione di tutti i Budget precedentemente creati e associati ai rispettivi account. La pagina offre la possibilità di aggiungere una spesa direttamente tramite la card del Budget o attraverso un apposito pulsante posizionato nell'angolo in alto a destra, denominato "**Add Expense**", che consente di selezionare il Budget appropriato.

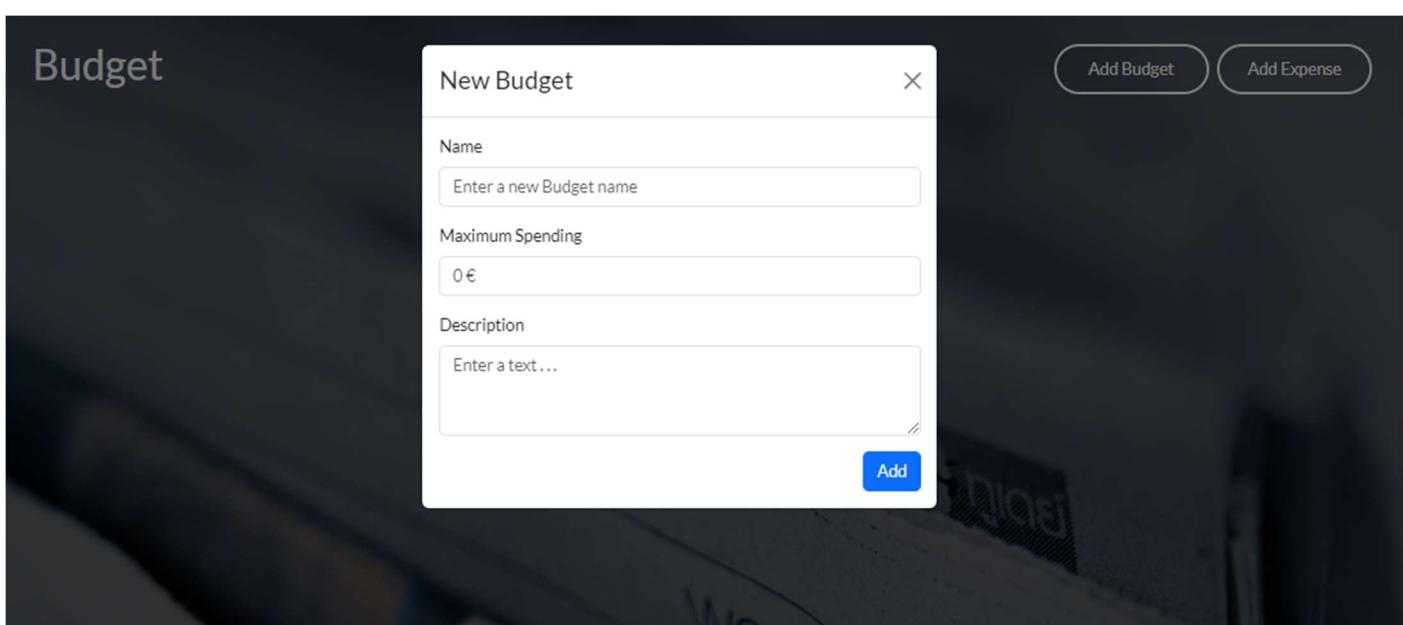
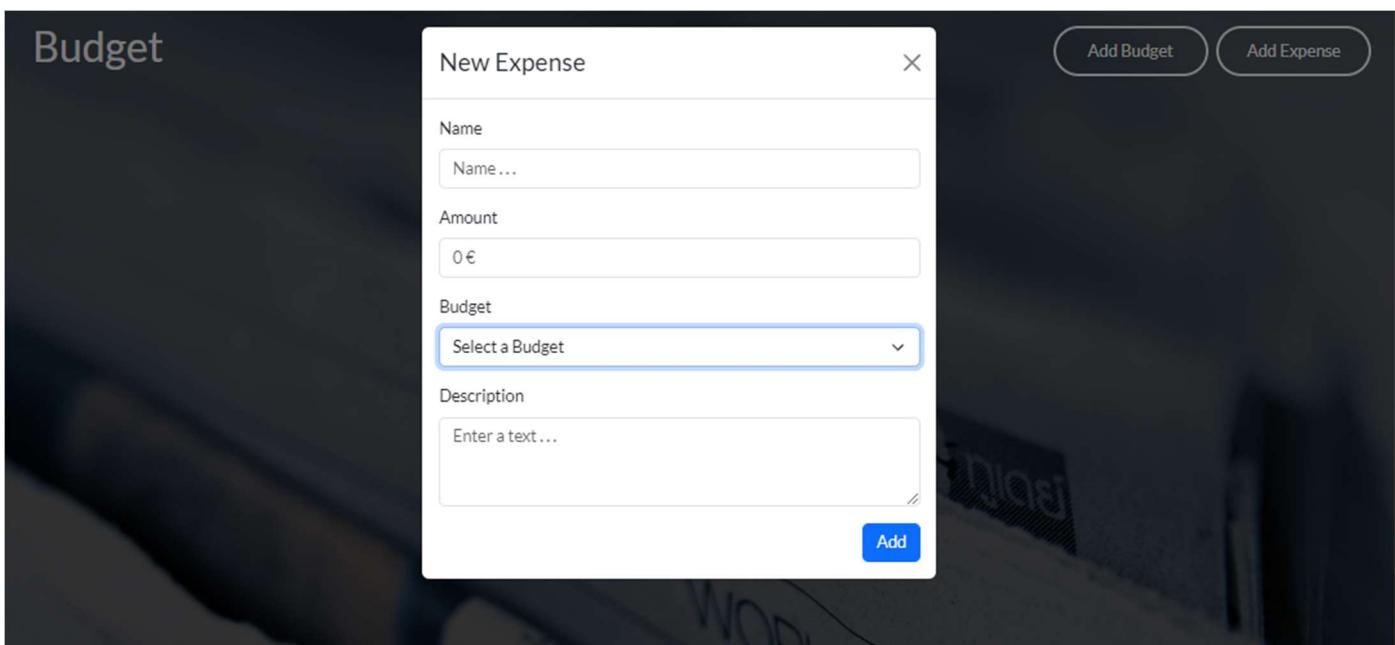
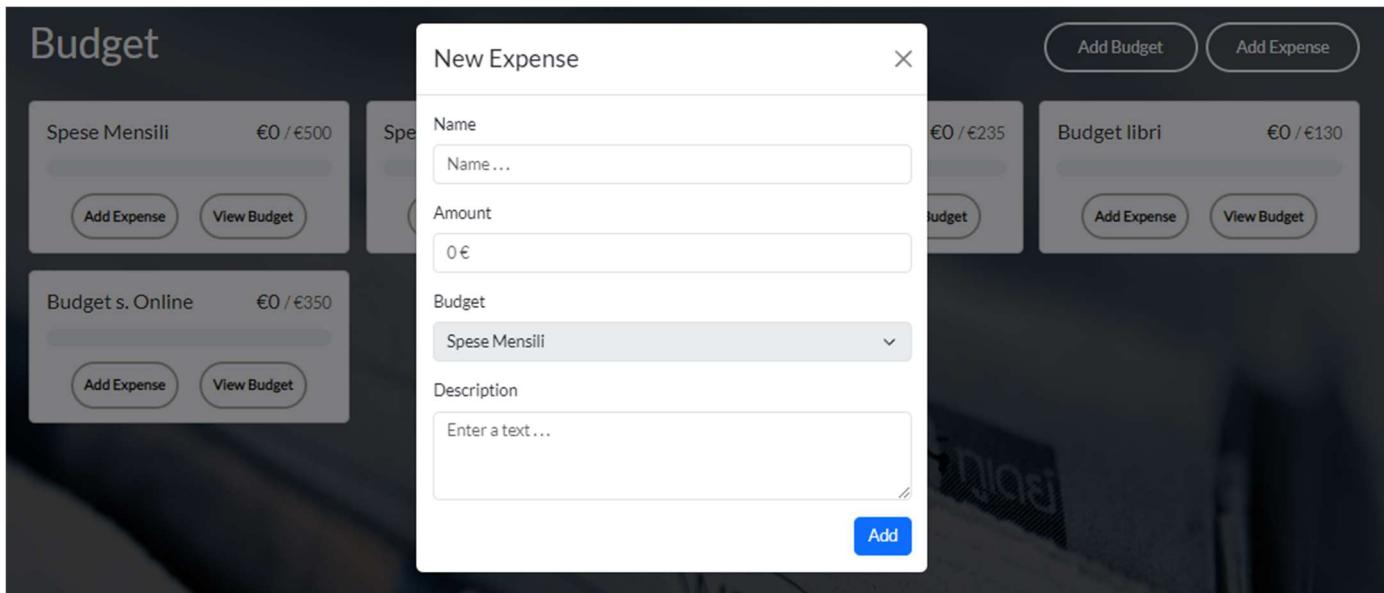
Inoltre, è disponibile la funzionalità di creare un nuovo Budget mediante il pulsante "**Add Budget**" situato nella stessa posizione. Cliccando su questo pulsante, gli utenti possono definire tutti i parametri necessari per il nuovo Budget.

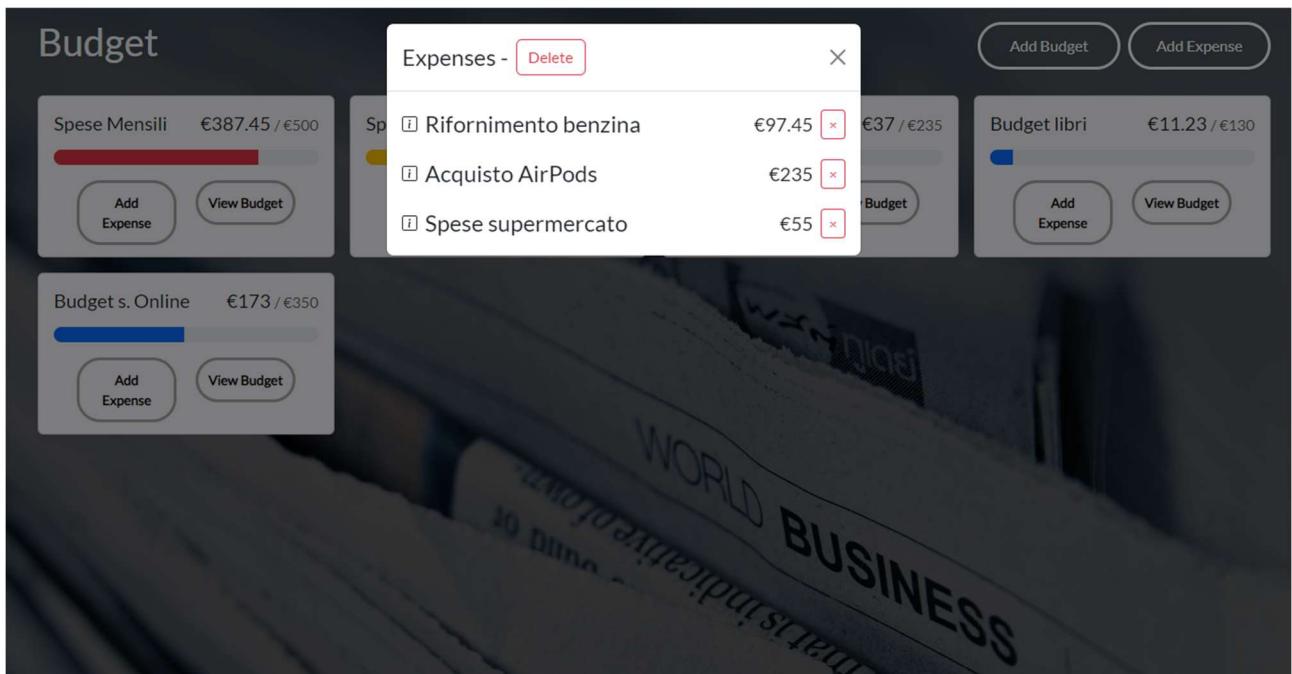
L'aggiunta di nuove spese associate a un Budget comporta l'aggiornamento automatico della barra di progresso relativa all'ammontare corrente del Budget.

Per esaminare nel dettaglio le spese di ciascun Budget, gli utenti possono selezionare l'opzione "**View Model**", che consente di visualizzare un modello espanso contenente informazioni dettagliate sulle transazioni effettuate e i relativi Budget. Inoltre, tramite questo modello è possibile eliminare sia le singole spese associate al Budget che il Budget stesso.

Sia le transazioni che i Budget sono dotati di un insieme completo di informazioni rilevanti per la gestione finanziaria. Tutto il sistema è supportato da una solida logica sia nel frontend che nel backend, che garantisce il corretto funzionamento delle varie funzionalità offerte dalla pagina di gestione dei Budget.





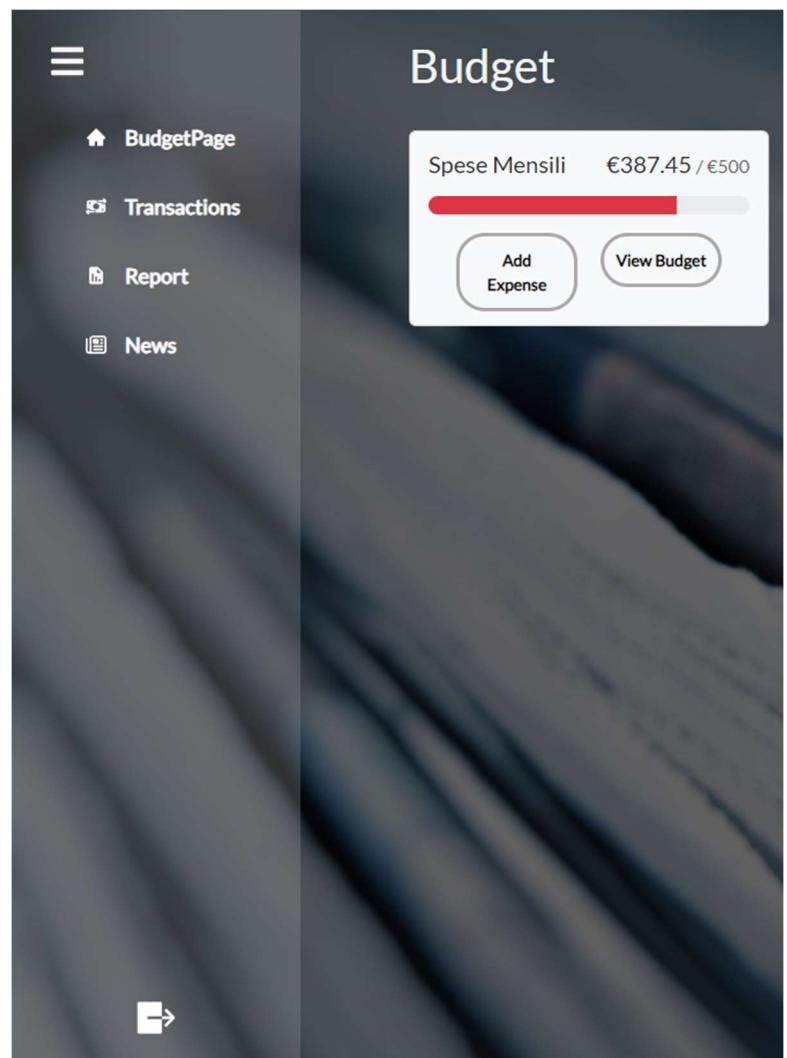


Schermata di Navigazione

Ogni pagina dell'applicazione presenta un pulsante laterale che attiva una **navbar** contenente le varie opzioni disponibili all'interno dell'applicazione.

Questa **navbar** offre una panoramica completa delle funzionalità offerte agli utenti.

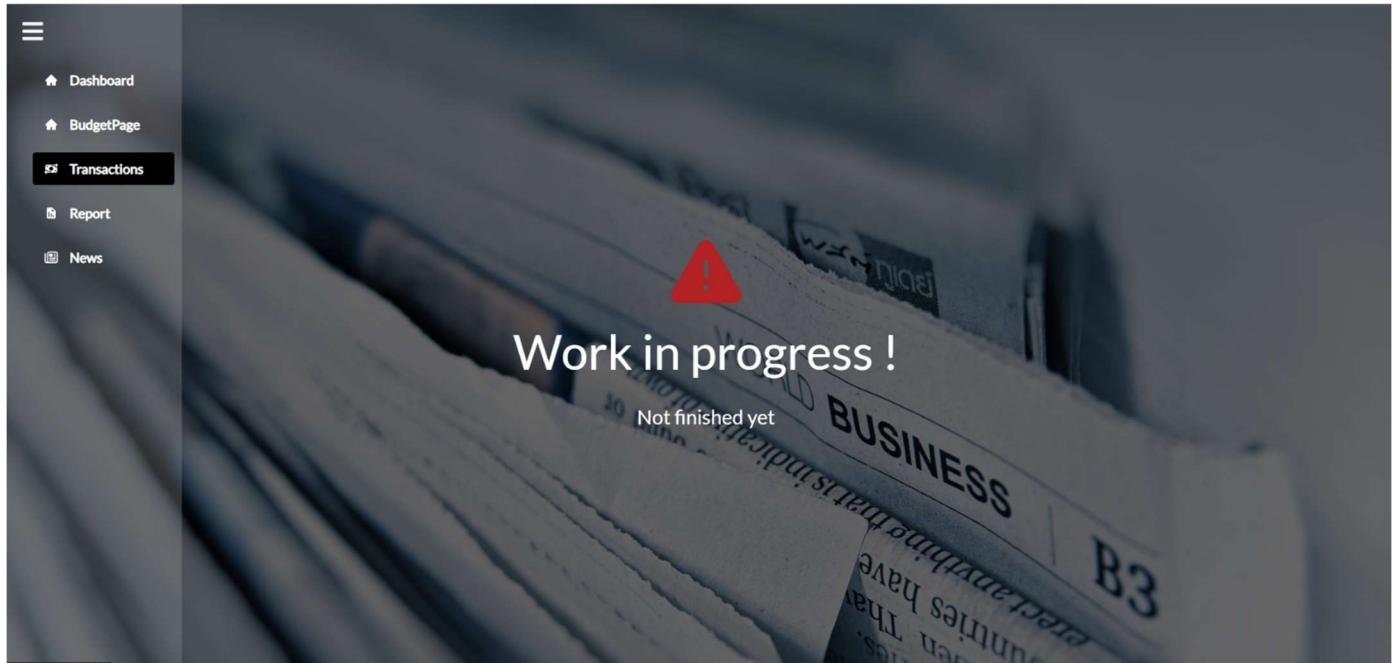
Inoltre, all'estremità inferiore della **navbar** è presente un pulsante dedicato che consente agli utenti di eseguire il logout dall'applicazione.



Resto delle Pagine del frontend

Come precedentemente specificato, nonostante lo sviluppo avanzato della logica di backend, non è stato ancora possibile completare l'implementazione del frontend per tutte le pagine dell'applicazione.

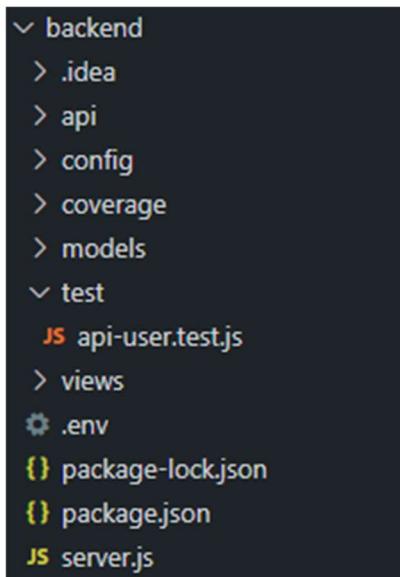
Di conseguenza, ogni pagina che non è ancora stata completamente sviluppata presenterà un avviso apposito, informando gli utenti dell'indisponibilità delle funzionalità correlate.



11 Testing

Per eseguire i test, abbiamo utilizzato la libreria Jest insieme al modulo SuperTest per l'invocazione delle API.

Tutti i test sono stati organizzati nella cartella "test/".



Il file `api-user.test.js` all'interno della directory "test" è essenzialmente strutturato per condurre il testing delle API che abbiamo sviluppato nel nostro software. Questo file è fondamentale per garantire che le funzionalità del nostro sistema rispondano correttamente e mantengano la coerenza con le specifiche stabilitate.

All'interno di ciascuna sezione, i singoli test sono definiti utilizzando le funzioni di Jest, come ad esempio `test()`. Ogni test si focalizza su uno specifico comportamento o scenario che l'API dovrebbe gestire correttamente. Questi test possono includere verifiche sulla corretta risposta delle API, sulla gestione degli errori, sulla validazione dei dati di input e su altri aspetti cruciali delle funzionalità testate.

Nel corpo di ciascun test, vengono eseguite le chiamate alle API utilizzando il modulo SuperTest, il quale permette di simulare le richieste HTTP verso il nostro server e di ottenere le risposte corrispondenti. Le risposte ricevute vengono quindi confrontate con i risultati attesi per verificare se le API funzionano come previsto.

Infine, all'interno del file di test, sono inclusi anche eventuali preparativi o operazioni di pulizia necessarie prima o dopo l'esecuzione dei test, come ad esempio l'inizializzazione del database o la cancellazione dei dati di prova.

Complessivamente, il file `api-user.test.js` è un componente cruciale del nostro processo di sviluppo del software, poiché ci consente di garantire che le nostre API siano affidabili, corrette e conformi ai requisiti specificati.

```
backend > test > JS api-user.test.js > ...
1  const request = require('supertest');
2  const app = require('../server');
3  const User = require('../models/User');
4  const Budget = require('../models/Budget');
5  const Transaction = require('../models/Transaction');
6  const News = require('../models/News');
7
8  const baseURL = "http://localhost:4500";
9
10
11 > describe('Testing the "/signup" endpoint', () => {
135 });
136
137 > describe('Testing the "/user/signin" endpoint', () => {
239 });
240
241 > describe('Testing the "/budgetAdd" endpoint', () => {
347 });
348
349 > describe('Testing the "/user/budgetDelete/:id" endpoint', () => {
401 });
402
403 > describe('Testing the "/budgetGet/:id" endpoint', () => {
458 });
459
460 > describe('Testing the "/getBudgetsByUserID/:id" endpoint', () => {
516 });
517
518 > describe('Testing the "/budgetUpdate/:budgetID" endpoint', () => {
581 });
582
583 > describe('Testing the "/budgetAfterdeleteTransUpdate/:budgetID" endpoint', () => {
646 });
647
648 > describe('Testing the "/transactionAdd" endpoint', () => {
742 });
743
744 > describe('Testing the "/transactionDelete/:id" endpoint', () => {
801 });
802
803 > describe('Testing the "/transactionGet/:id" endpoint', () => {
848 });
849
850 > describe('Testing the "/newsAdd" endpoint', () => {
890 });
891
892 > describe('Testing the "/newsDelete/:id" endpoint', () => {
934 });
```

Il file api-user.test.js è responsabile dell'esecuzione dei test per le API del nostro software. Inizialmente, il file richiede il modulo **supertest** per la simulazione delle richieste HTTP, il modulo app per accedere all'applicazione server e i modelli necessari (**User, Budget, Transaction, News**) per interagire con il database.

All'interno del file, è definito il **baseURL** che rappresenta l'URL di base delle API in fase di test.

Il file è strutturato in una serie di blocchi di test, ciascuno dei quali riguarda un'API specifica o un insieme di API correlate. Ad esempio, la sezione di test "Testing the '/signup' endpoint" contiene test relativi alla funzionalità di registrazione degli utenti.

Per ogni blocco di test, viene utilizzata la funzione **describe()** di Jest per raggruppare i test correlati. All'interno di ciascun blocco di test, sono presenti i test effettivi utilizzando la funzione **test()** di Jest.

All'interno dei singoli test, vengono eseguite le richieste HTTP verso le API specifiche utilizzando il modulo **request** di supertest. Le risposte ricevute vengono quindi verificate per garantire che corrispondano ai risultati attesi, utilizzando le asserzioni di Jest come **expect()**.

Prima di ogni test, è definita una funzione **beforeEach()** per creare un utente fittizio nel database. Allo stesso modo, dopo ogni test, viene definita una funzione **afterEach()** per eliminare l'utente fittizio dal database. Questo approccio garantisce che ogni test sia eseguito su uno stato del database prevedibile e coerente.

```

11  describe('Testing the "/signup" endpoint', () => {
12
13    const existingUser = {
14      name: "Existing User",
15      email: "existing@example.com",
16      password: "password123",
17      dateOfBirth: "1990-01-01"
18    };
19
20    beforeEach(async () => {
21      // Creare un utente fittizio nel database prima di ogni test
22      await User.create(existingUser);
23    });
24
25    afterEach(async () => {
26      // Eliminare l'utente fittizio dal database dopo ogni test
27      await User.deleteOne({ email: existingUser.email });
28    });
29
30    test("API call with existing email", async () => {
31      // Creare un nuovo utente con lo stesso indirizzo email
32      const newUser = {
33        name: "New User",
34        email: "existing@example.com",
35        password: "newpassword123",
36        dateOfBirth: "1990-01-01"
37      };
38
39      const response = await request(baseURL)
40        .post('/user/signup')
41        .send(newUser);
42
43      // Verificare che la risposta indichi che un utente con lo stesso indirizzo email esiste già
44      expect(response.statusCode).toEqual(200);
45      expect(response.body.status).toEqual('FAILED');
46      expect(response.body.code).toEqual(105);
47      expect(response.body.message).toEqual('A user with the provided email already exists');
48    });
49  > test("API call with empty input fields", async () => { ... });
50  > test("API call with invalid name containing numbers", async () => { ... });
51  > test("API call with invalid email format", async () => { ... });
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

La sezione di test per l'endpoint "/signup" è organizzata all'interno del file api-user.test.js utilizzando la funzione describe() di Jest. Questa funzione è utilizzata per raggruppare insieme i test relativi alla registrazione degli utenti in un contesto logico e comprensibile.

All'interno del blocco describe('Testing the "/signup" endpoint', () => { ... }), sono definiti una serie di test specifici che coprono vari scenari di utilizzo e potenziali risultati dell'endpoint di registrazione.

Ogni test è implementato utilizzando la funzione test() di Jest e rappresenta un caso di test specifico. Ad esempio, possiamo avere test che verificano la gestione di input vuoti, la validità dei dati inseriti, la presenza di un utente con lo stesso indirizzo email, la lunghezza della password, ecc.

Ogni test esegue una richiesta HTTP POST verso l'endpoint "/signup" simulando l'invio di dati dal client al server. I dati inviati includono informazioni come nome, email, password e data di nascita dell'utente.

Dopo aver inviato la richiesta, il test verifica la risposta ricevuta dal server per garantire che corrisponda alle aspettative. Questo può includere la verifica dello status code della risposta, il controllo del corpo della risposta per determinati messaggi di errore o conferme di successo, e altro ancora.

Inoltre, prima di eseguire ogni test, viene utilizzata la funzione beforeEach() per creare un utente fittizio nel database. Allo stesso modo, dopo ogni test, viene utilizzata la funzione afterEach() per eliminare l'utente fittizio dal database. Questo assicura che ogni test venga eseguito su un ambiente pulito e prevedibile.

Complessivamente, la struttura di describe() e i test specifici all'interno di api-user.test.js forniscono un approccio organizzato e dettagliato per testare l'endpoint "/signup" del nostro software.

12 Risultati Testing

Per effettuare il testing è stato opportunamente settato i file package.json all'interno della cartella /backend come segue.

```
"scripts": {
  "test": "jest --coverage"
},
```

Permettendo così di effettuare il comando **npm test** e far eseguire il nostro file **api-user.test.js**

Il flag --coverage nell'esecuzione del comando npm test viene utilizzato per generare un report di copertura del codice durante l'esecuzione dei test.

Quando questo flag è attivato, Jest tracerà quali parti del codice sono state eseguite durante l'esecuzione dei test e genererà un report dettagliato che indica la percentuale di copertura del codice per ciascun file del progetto.

In sostanza, l'utilizzo del flag --coverage durante l'esecuzione dei test consente di valutare la qualità e l'estensione dei test implementati nel nostro progetto.

Di seguito mostriamo i risultati del “npm test”:

```
PASS  test/api-user.test.js (6.101 s)
Testing the "/signup" endpoint
  ✓ API call with existing email (893 ms)
  ✓ API call with empty input fields (68 ms)
  ✓ API call with invalid name containing numbers (68 ms)
  ✓ API call with invalid email format (69 ms)
  ✓ API call with invalid date of birth format (67 ms)
  ✓ API call with password too short (68 ms)
Testing the "/user/signin" endpoint
  ✓ API call with unverified email (36 ms)
  ✓ API call with valid credentials (109 ms)
  ✓ API wrong (108 ms)
  ✓ API call with empty credentials (2 ms)
  ✓ API call with invalid credentials (30 ms)
Testing the "/budgetAdd" endpoint
  ✓ API success add budget (155 ms)
  ✓ API call with empty input fields (66 ms)
  ✓ API call with negative amount (67 ms)
  ✓ API call with existing budget name (123 ms)
Testing the "/user/budgetDelete/:id" endpoint
  ✓ API call with valid budget ID (35 ms)
  ✓ API call with invalid budget ID (3 ms)
  ✓ API call with non-existing budget ID (31 ms)
Testing the "/budgetGet/:id" endpoint
  ✓ API call with valid budget ID (33 ms)
  ✓ API call with invalid budget ID (3 ms)
  ✓ API call with non-existing budget ID (31 ms)
Testing the "/getBudgetsByUserID/:id" endpoint
  ✓ API call with valid user ID (40 ms)
  ✓ API call with invalid user ID (3 ms)
  ✓ API call with non-existing user ID (32 ms)
Testing the "/budgetUpdate/:budgetID" endpoint
  ✓ API call with valid value (141 ms)
  ✓ API call with invalid value (69 ms)
  ✓ API call with non-existing budget ID (1365 ms)
Testing the "/budgetAfterDeleteTransUpdate/:budgetID" endpoint
  ✓ API call with valid value (136 ms)
  ✓ API call with invalid value (81 ms)
  ✓ API call with non-existing budget ID (97 ms)
Testing the "/transactionAdd" endpoint
  ✓ API call with valid input fields (97 ms)
  ✓ API call with empty input fields (66 ms)
  ✓ API call with negative amount (63 ms)
  ✓ API call without selecting a valid budget (63 ms)
Testing the "/transactionDelete/:id" endpoint
  ✓ API call to delete an existing transaction (158 ms)
  ✓ API call to delete a non-existing transaction (98 ms)
  ✓ API call with invalid transaction ID (69 ms)
Testing the "/transactionGet/:id" endpoint
  ✓ API call with valid budget ID (97 ms)
  ✓ API call with invalid budget ID (70 ms)
Testing the "/newsAdd" endpoint
  ✓ API call with valid data (73 ms)
  ✓ API call with missing data (118 ms)
Testing the "/newsDelete/:id" endpoint
  ✓ API call to delete existing news (97 ms)
  ✓ API call to delete non-existing news (95 ms)
Testing the "/newsGet" endpoint
  ✓ API call to retrieve all news (33 ms)
  ✓ API call to retrieve all news – Error handling (4 ms)

-----|-----|-----|-----|-----|-----|
File      | % Stmtns | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 62.01   | 80.2     | 32.65   | 62.01   |
backend   | 100      | 100      | 100     | 100     |
server.js | 100      | 100      | 100     | 100     |
backend/api | 55.73   | 80.2     | 31.57   | 55.73   |
User.js   | 55.73   | 80.2     | 31.57   | 55.73   | 57,202-243,256-315,374-457,4
backend/config | 80       | 100      | 50      | 80      |
db.js     | 80       | 100      | 50      | 80      | 11
backend/models | 100      | 100      | 100     | 100     |
Budget.js | 100      | 100      | 100     | 100     |
News.js   | 100      | 100      | 100     | 100     |
PasswordReset.js | 100      | 100      | 100     | 100     |
Transaction.js | 100      | 100      | 100     | 100     |
User.js   | 100      | 100      | 100     | 100     |
UserVerification.js | 100      | 100      | 100     | 100     |

Test Suites: 1 passed, 1 total
Tests:       45 passed, 45 total
Snapshots:   0 total
Time:        6.165 s, estimated 8 s
Ran all test suites.
```

La test suites contenente i 45 test definiti da **test()**, sono risultati essere eseguiti con successo e quindi risultano passati.

Il report relativo ai risultati dei test è localizzato nella directory coverage/ situata nella cartella /backend, seguita dalla sottodirectory lcov-report/. Al suo interno, il file principale index.html costituisce l'interfaccia attraverso la quale è possibile esaminare e valutare in dettaglio i risultati conseguiti. Tale pagina web fornisce una panoramica esaustiva e interattiva della copertura del codice, consentendo l'analisi approfondita delle singole unità, la determinazione delle linee di codice coperte e non coperte, e la presentazione di statistiche chiare sulla copertura complessiva del codice per l'intero progetto.



Nei casi non inseriti all'interno dei test sicuramente ci sono errori indipendenti da noi che possono succedere (ad esempio errori del server, errori di connessione al dB, o sincronizzazione). Un esempio è il seguente:

```

225          .catch((err) => {
226            res.json({
227              status: "FAILED",
228              code: 106,
229              message: "An error occurred while saving the user account"
230            });
231          });
232        });
233      );
234    );
235    .catch(err => {
236      res.json({
237        status: "FAILED",
238        code: 108,
239        message: "An error occurred while hashing the password"
240      });
241    });
242    );
243    .catch(err => {
244      console.log(err);
245      res.json({
246        status: "FAILED",
247        code: 109,
248        message: "An error occurred while checking for an existing user"
249      });
250    });

```

13 GitHub repository e informazioni sul deployment

Vi invitiamo a prendere visione della nostra repository principale del progetto, accessibile tramite il seguente link: https://github.com/G45-UNITN/G45_Project.git. Qui troverete sia il front-end che il back-end, come precedentemente descritto. Di seguito forniamo dettagli sul deployment.

Il deployment è stato effettuato utilizzando Heroku per il backend e GitHub Pages per il frontend.

Abbiamo creato una specifica cartella per il deployment del solo front-end, reperibile al link: <https://github.com/G45-UNITN/frontend.git>.

L'URL del backend è il seguente: <https://agile-tundra-48116-00eb7327fba8.herokuapp.com/>, mentre per il frontend è: <https://g45-unitn.github.io/frontend/>.

Da notare: sebbene il deployment del back-end funzioni senza problemi - abbiamo infatti testato con successo ogni API tramite Postman - il deployment del front-end potrebbe causare problemi, potendo non visualizzare correttamente il codice. Pertanto, consigliamo vivamente di eseguire la webApp in locale (sia il front-end che il back-end).

Per eseguire il server sulla propria macchina, seguite le istruzioni dettagliate nel file README.md presente nella radice della repository: README.md della repository principale.

Nella cartella principale del back-end troverete già il file .env, contenente tutte le variabili necessarie per l'esecuzione della webApp. Nel caso desideriate sostituire il DB, vi consigliamo di inserire il vostro MONGODB_URI.

Per visualizzare la documentazione Swagger in locale, avviate il server e visitate l'indirizzo <http://localhost:4500/api-docs>, dove potrete testare regolarmente tutte le API.