UNIVERSITÀ DI TRENTO

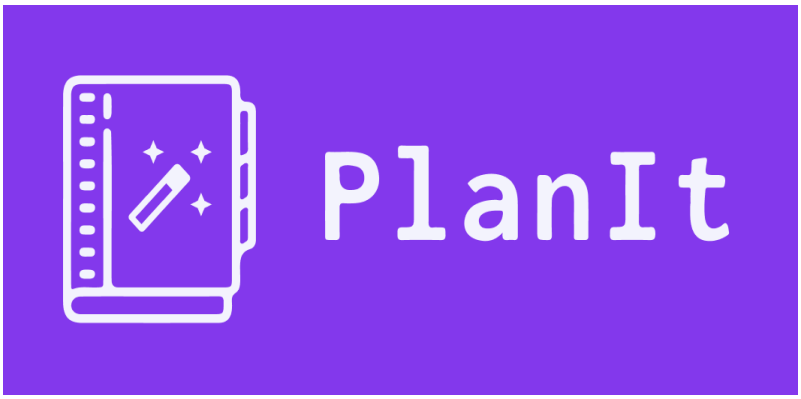
Dipartimento d'Ingegneria e
Scienze dell'informazione

Progetto:

PlanIt

Titolo del documento:

Sviluppo Applicazione



Gruppo T56
Gabriele Lacchin, Denis Lucietto ed Emanuele Zini

Indice

Scopo del documento

da sistemare, copia e incollato

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di una parte dell'applicazione PlanIt. In particolare, presenta tutti gli artefatti necessari per realizzare i servizi di creazione e modifica di eventi, calendari e utente autenticato nel sito. Partendo dalla descrizione degli user flows legati, soprattutto, al ruolo dell'utente autenticato dell'applicazione, il documento prosegue con la presentazione delle API necessarie (tramite l'API Model e il Modello delle risorse) per poter visualizzare, creare e modificare sia gli eventi che i calendari necessari per implementare l'applicativo PlanIt, che ricordiamo essere un applicativo di gestione calendari. Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati. Per i test effettuatati, verrà anche fornito un documento html con il resoconto di questi. Infine una sezione e' dedicata alle informazioni del Git Repository e il deployment dell'applicazione stessa.
Di seguito sono presenti gli argomenti che verranno trattati in questo documento.

- User Flows;
- Application Implementation and Documentation;
- API Documentation;
- FrontEnd Implementation;
- GitHub Repository and DeploymentInfo;
- Testing;

1 User Flows

In questa sezione del documento di sviluppo riportiamo gli “user flows” riguardanti, soprattutto, il ruolo dell'utente autenticato nel nostro sito. Questa figura descrive gli user flows relativi alle funzionalità che abbiamo reso disponibili in questo prototipo dell'applicativo PlanIt. Come si può notare dall'user flow, l'utente, una volta autenticato, visualizza la schermata "Calendario", schermata principale della nostra piattaforma. La procedura di autenticazione da parte dell'utente è piuttosto intricata; infatti l'utente ha la possibilità sia di accedere/registrarsi mediante proprie credenziali sia mediante un proprio account Google. Nel caso in cui si fosse registrato mediante credenziali e avesse dimenticato la propria password, l'utente ha la possibilità di fare il "reset password" per recuperare il proprio account indicando a quale indirizzo email inviare l'email di recupero.
Una volta effettuato l'accesso, l'utente, divenuto utente autenticato, visualizza la schermata "Calendario" da cui:

- può accedere alla seconda schermata implementata, ovvero "Evento";
- può fare diverse funzionalità direttamente da questa schermata, come: visualizzare periodi temporali diversi nel calendario, filtrare i calendari che si visualizzano, creare eventi e calendari direttamente da questa schermata, anziché andando nella schermata "Evento", dove, però, c'è anche la possibilità di modificare quest'ultimi.

Si specifica che per "compilazione evento", azione presente sia quando si modifica che crea un evento, si intende la definizione di vari campi, ovvero:

- titolo, campo da completare obbligatoriamente per far andare a buon fine la procedura di creazione o modifica evento;
- data, campo da completare obbligatoriamente per far andare a buon fine la procedura di creazione o modifica evento;
- priorità, campo facoltativo;
- difficoltà, campo facoltativo;
- calendario, ovvero calendario a cui appartiene l'evento che si sta modificando o creando. Questo è un campo da completare obbligatoriamente per far andare a buon fine la procedura di creazione o modifica evento;
- notifiche, ovvero quando ricevere la notifica di tale evento.

Infine per "compilazione calendario", azione presente sia nella "creazione" che "modifica" calendario, sin intende la definizione di vari campi, ovvero:

- nome del calendario, campo che deve essere definito per forza per l'esito positivo della procedura di creazione o modifica;
- colore, campo che deve essere definito per forza per l'esito positivo della procedura di creazione o modifica;
- fuso orario, campo facoltativo;
- impostazioni predefinite degli eventi, ovvero definizione di alcuni campi con cui si vanno a precompilare gli eventi appartenenti a quel calendario. Ovviamente, nella creazione o modifica di un evento, questi campi precompilati potranno essere modificati.

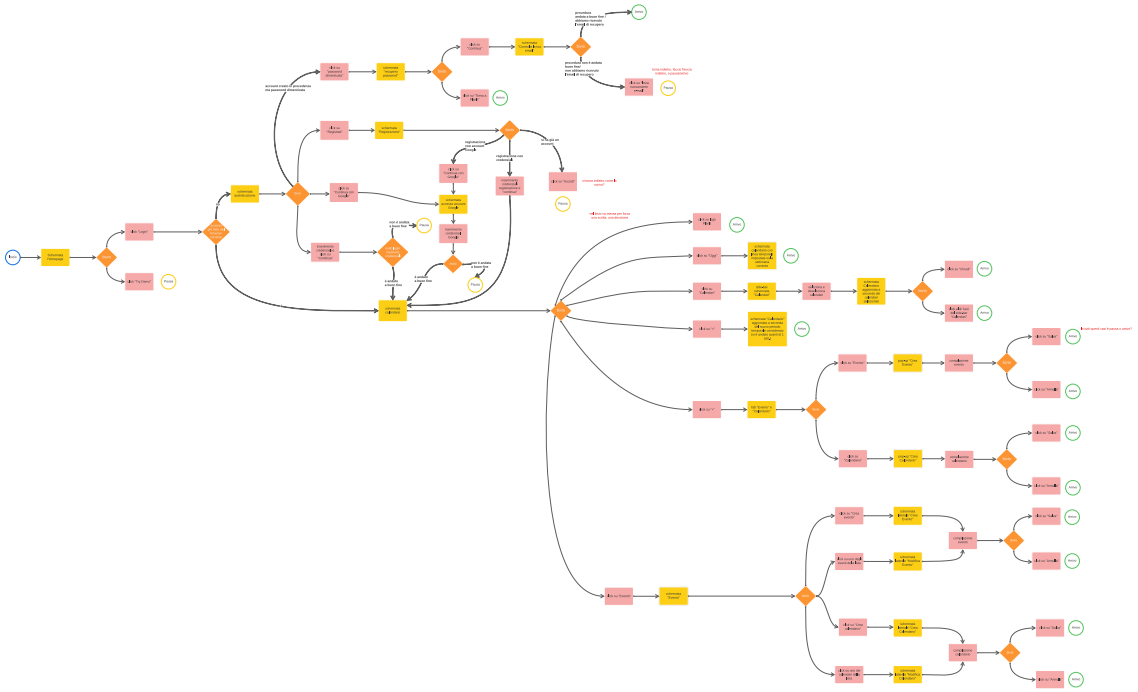


Immagine PNG/SVG User Flows

2 Application Implementation and Documentation

Nelle sezioni precedenti abbiamo individuato tutte le procedure e funzionalità che sono implementate nel prototipo del sito PlanIt e di come l'utente autenticato può utilizzarle nel suo flusso applicativo mostrando un diagramma di user flows. L'applicazione è stata sviluppata utilizzando NextJS (anche TypeScript ma solo in piccola parte) per la parte di API, invece per la parte di UI (user interface) sono stati utilizzati CSS e React. Per la gestione dei dati abbiamo utilizzato MongoDB. Per la gestione di autenticazione all'interno del sito abbiamo utilizzato il sistema esterno Auth0. Per la produzione della documentazione delle APIs abbiamo usato la libreria Swagger di NextJS, invece per il testing, sempre delle APIs, è stato usato il framework di JavaScript Jest.

APD1 : **Project Structure**

La struttura del progetto è presentata nelle seguenti foto ed è composto di una cartella "api", dove sono presenti tutte le funzioni per la gestione delle APIs locali, divisi in base a quale resource gestiscono, di una cartella "pages", al cui interno oltre che la cartella api, sono presenti anche i file di estensione React indispensabili per comporre le interfacce grafiche di ciascuna schermata presente nel sito PlanIt, di una cartella "lib", in cui sono presenti i file usati per andare a stabilire la connessione con il database MongoDB, e, infine, di una cartella "models", dove ci sono i modelli di oggetti implementati per la gestione di dati del nostro sito. Si evidenzia che i modelli definiti sono presenti nel database MongoDB.

Sottolineiamo che le uniche "pages" che sono state sviluppate in maniera completa sono "calendario" e "eventi". Evidenziamo che le varie componenti che vanno a comporre queste schermate sono state sviluppate in React nella cartella "components", che può essere sempre visualizzata nella seguente foto.

Inoltre, nella cartella "styles" sono presenti i file "css" sviluppati per andare a definire i stili dei moduli presenti nell'interfaccia grafica che saranno visualizzabili dall'utente che entra nel sito PlanIt.

APD2 : **Project Dependencies**

I seguenti moduli sono stati utilizzati e aggiunti al file "package.json", il cui contenuto è anche visualizzabile nella seguente foto:

- MongoDB, per cui per connettersi è stata utilizzata la libreria "mongoose";
- NextJS, framework utilizzato per lo sviluppo delle APIs e del FrontEnd del sito PlanIt;
- Swagger, tool utilizzato per la documentazione delle APIs;
- React, libreria di JavaScript per l'implementazione dell'UI del sito;
- Jest, framework di JavaScript utilizzato per andare a fare il testing delle APIs;
- PostCSS, è uno strumento di sviluppo software che utilizza plug-in basati su JavaScript per automatizzare le operazioni CSS di routine e quindi lo sviluppo dei moduli presenti nel FrontEnd del nostro sito;
- TypeScript, estensione di JavaScript, usata solo in minima parte nello sviluppo delle APIs, ovvero nello sviluppo degli esempi per la documentazione di queste.

```
{
  "name": "life-planner",
  "version": "0.1.0",
  "private": true,
  ▷ Debug
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "test": "jest --watch --silent=false --runInBand --coverage",
    "lint": "next lint"
  },
  "dependencies": {
    "@auth0/nextjs-auth0": "2.0.1",
    "daisyui": "^2.42.1",
    "mongodb": "3.5.9",
    "mongoose": "^6.8.0",
    "next": "13.0.6",
    "next-swagger-doc": "^0.3.6",
    "react": "18.2.0",
    "react-dom": "18.2.0",
    "swagger-ui-react": "^4.15.5"
  },
  "devDependencies": {
    "@shelf/jest-mongodb": "^4.1.4",
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@types/node": "18.11.12",
    "autoprefixer": "^10.4.13",
    "jest": "^29.3.1",
    "jest-environment-jsdom": "^29.3.1",
    "jest-html-reporter": "^3.7.0",
    "node-mocks-http": "^1.12.1",
    "postcss": "^8.4.19",
    "tailwindcss": "^3.2.4",
    "typescript": "4.9.4"
  }
}
```

Immagine PNG Project Dependencies

APD3 : Project Data or DB

Per la gestione dei dati utili all'applicazione abbiamo definito tre principali strutture dati come illustrato nelle seguenti immagini. Infatti, le strutture dati che abbiamo deciso di avere in questo prototipo del nostro sito PlanIt, sono:

- "UtenteAutenticato", struttura dati che individua l'utente che si è autenticato ed accede al sito;
- "Evento", componente fondamentale del calendario, che sta alla base della logica del nostro sito ideato.
- "Calendario", contenitore di eventi, struttura dati sopra citata.

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
Calendario	1	371B	371B	36KB	1	36KB	36KB
Evento	2	859B	430B	36KB	1	36KB	36KB
UtenteAutenticato	3	447B	149B	36KB	1	36KB	36KB

Collections delle strutture dati usate nell'applicazione


```
_id: ObjectId('63a709dadad816515f857b37')
userId: "User1"
email: "prova@example.com"
username: "Username1"
__v: 0
```

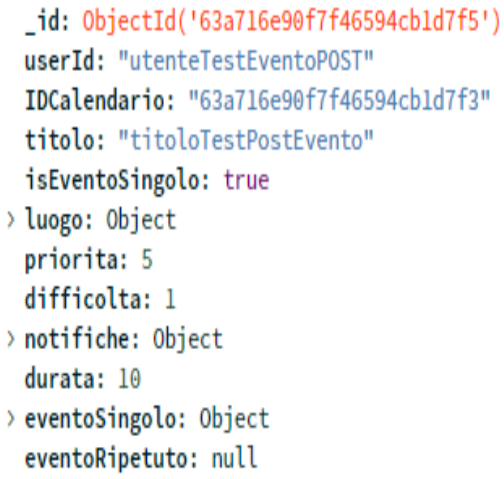

Tipo di dato "Utente Autenticato"


```
_id: ObjectId('63a71058d0009bdc566244f7')
nome: "calendarioTestProva1"
› fusoOrario: Object
colore: "#7C36B9"
› partecipanti: Array
principale: false
› impostazioniPredefiniteEventi: Object
__v: 0
```


Tipo di dato "Calendario"

Immagine PNG Collections
Immagine PNG Tipo di dato "Utente Autenticato"
Immagine PNG Tipo di dato "Calendario"

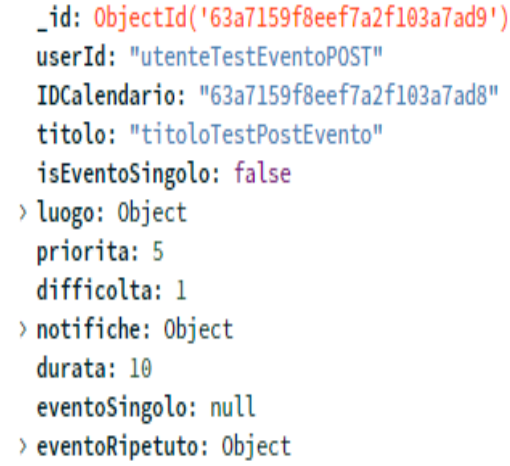
Tipo di dato "Calendario" con gli oggetti contenuti "aperti"



Tipo di dato "Evento", in questo caso è un evento singolo, che quindi è presente solo in un giorno specifico

Immagine PNG Tipo di dato "Calendario" con gli oggetti contenuti aperti
Immagine PNG Tipo di dato "Evento" singolo

Tipo di dato "Evento" singolo con gli oggetti contenuti "aperti"



Tipo di dato "Evento", in questo caso è un evento ripetuto, che quindi è presente su più giorni

Immagine PNG Tipo di dato "Evento" singolo con gli oggetti contenuti aperti
Immagine PNG Tipo di dato "Evento" ripetuto

Tipo di dato "Evento" ripetuto con gli oggetti contenuti "aperti"

Immagine PNG Tipo di dato "Evento" ripetuto con gli oggetti contenuti aperti

APD4 : Project APIs

In questo capitolo, verranno presentate le APIs che sono state sviluppate in questo prototipo del sito PlanIt. Infatti, si passerà dall'esposizione dell' Extract Diagram, diagramma che serve come collegamento tra il Class Diagram alla APIs, e Resources Models, diagramma in cui sono presenti tutte le risorse implementate, alla descrizione del vero e proprio codice delle APIs presentati in questi primi diagrammi.

4.1 : Resources Extraction from the Class Diagram

Questo diagramma si può dire che faccia da vero e proprio "ponte" tra Class Diagram, presentato nel documento D3, e le Resources che andremo ad implementare nelle nostre APIs.
Come si può notare, abbiamo individuato dodici Resources che riteniamo fondamentali per sviluppare almeno la logica che sta alla base del nostro sito PlanIt.

- Partendo dalla classe "GestioneChiamateMongoDB" e dalle sue funzioni e attributi, presentati in D3 DCL13, abbiamo individuato cinque Resources, ovvero:
  - CreaAccount, di tipo POST. Risorsa che ha lo scopo di creare un account all'interno del database MongoDB e per questo motivo ha l'etichetta BackEnd; infatti l'effetto principale di questa API avviene nel BackEnd con la creazione dell'account nel database.
    L'account è una struttura dati già presenta nel capitolo precedente e l'input delineato è un "body" in quanto è formato da più di tre parametri, i quali verrano specificati e descritti nel diagramma Resources Model, presentato nel successivo capitolo.
  - ModificaAccount, di tipo PUT. Risorsa che ha al fine di dare la possibilità di andare a modificare un account già presente nel nostro database. L'etichetta BackEnd e l'input body sono preseti per gli stessi motivi definiti per la resource "CreaAccount".
  - GetDatiAccount, di tipo GET. API che ha lo scopo di ottenere i dati di un "UtenteAutenticato" conoscendo l'userId di tale utente. L'effetto ultima di questa resource è nel FrontEnd, in quanto i dati appariranno all'utente e quindi si è deciso di mettere come sua etichetta "FrontEnd"
  - GetDatiAccount_email, di tipo GET. Risorsa analoga a quella precedente, con la sola differenza che i dati account non sono definite mediante l'userId, stringa che identifica univocamente un utente, ma mediante la sua email.
  - getEventi, di tipo GET. Risorsa mediante cui si ottengo gli eventi di un calendario, dato l'userId dell'utente che ha tale calendario. L'etichetta FrontEnd è stata messa in quanto questa API è fondamentale per mostrare all'utente finale nell'user interface gli eventi di un determinato calendario.
  - getCalendari, di tipo GET. API che ha lo scopo di ottenere tutti i calendari di un determinato UtenteAutenticato, dato il suo userId. Questa risorsa è fondamentale per andare a mostrare i calendari appartenenti all'utente, che potrà, quindi, visualizzare.

  Si specifica che gli attributi "link", "password" e "schema" sono necessari per poter effettuare la connessione con il database MongoDB, come specificato in D3 DCL13, e tutte le resources individuate riguardano a funzioni già presentate sempre in D3 DCL13.
- Dopo abbiamo individuato altre tre Resources, partendo dalla classe "Evento, già presentata in D3 DCL6.2, ovvero:
  - creaEvento, di tipo POST. Resource che ha lo scopo di andare a creare un evento che viene salvato nel database MongoDB. L'input che viene preso da questa API è un oggetto di tipo "Evento" e l' "userId" di un utente, struttura dati già definita nel capitolo precedente, che in quanto piuttosto complicata, non siamo andati a specificare in questo diagramma ma nel Resources Model e qua ci siamo fermati nel scrivere come input solo "body". In quanto ha un effetto sul BackEnd, con il salvataggio dell'evento nel database, abbiamo indicato come etichetta BackEnd.
  - modificaEvento, di tipo PUT; grazie alla sua esecuzione abbiamo la modifica di un determinato evento già presente al database appartenente ad un utente.

In quanto ha un effetto finale nel database, gli abbiamo attribuito l'etichetta BackEnd.

- eliminaEvento, di tipo DELETE. Lo scopo di questa resource è quello di dare la possibilità di eliminare un "Evento" dato il suo "objectId" e l'"userId" dell'utente autenticato che ha questo determinato "Evento" che vuole eliminare. In quanto ha un effetto finale nel database, gli abbiamo attribuito l'etichetta BackEnd.

Si specifica che tutte le resources individuate riguardano a funzioni già presentate in D3 DCL6.2.

- Infine abbiamo sviluppato tre resources per la classe "Calendario", tipo di dato già presentato nel precedente capitolo, le cui funzionalità e attributi sono stati descritti in D3 DCL6.1. Le tre resources sono:
  - creaCalendario, di tipo POST. API che ha lo scopo di creare un calendario all'interno del database MongoDB, e per questo motivo ha come etichetta BackEnd. In quanto, per andare a creare un "Calendario" all'interno del database serve un oggetto di tipo "Calendario" e l' "userId" dell'utente che vuole creare tale calendario, come input abbiamo indicato body. L'input verrà descritto maggiormente nel Resources Model che viene presentato nel successivo capitolo.
  - modificaCalendario, di tipo PUT; grazie alla sua esecuzione abbiamo la modifica di un determinato calendario già presente al database appartenente ad un utente. In quanto ha un effetto finale nel database, gli abbiamo attribuito l'etichetta BackEnd.
  - eliminaCalendario, di tipo DELETE. Lo scopo di questa resource è quello di dare la possibilità di eliminare un "Evento" dato il suo "objectId" e l'"userId" dell'utente autenticato che ha questo determinato "Calendario" che vuole eliminare. In quanto ha un effetto finale nel database, gli abbiamo attribuito l'etichetta BackEnd.

Si specifica che tutte le resources individuate riguardano a funzioni già presentate in D3 DCL6.1.

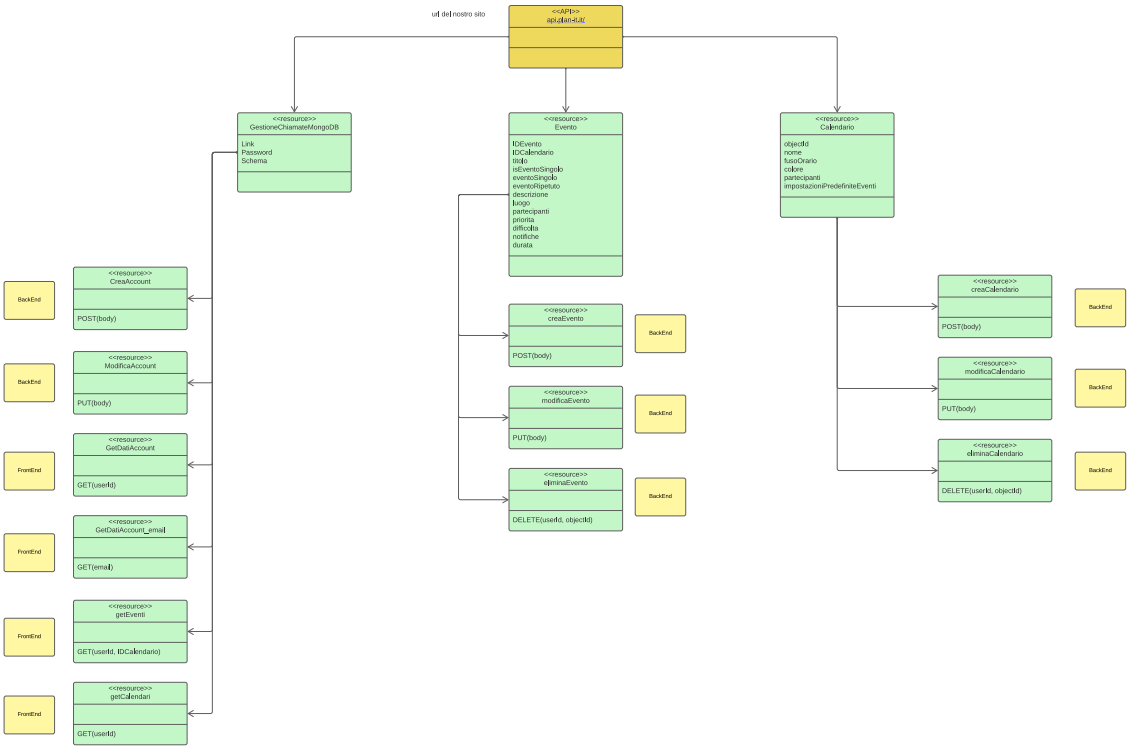


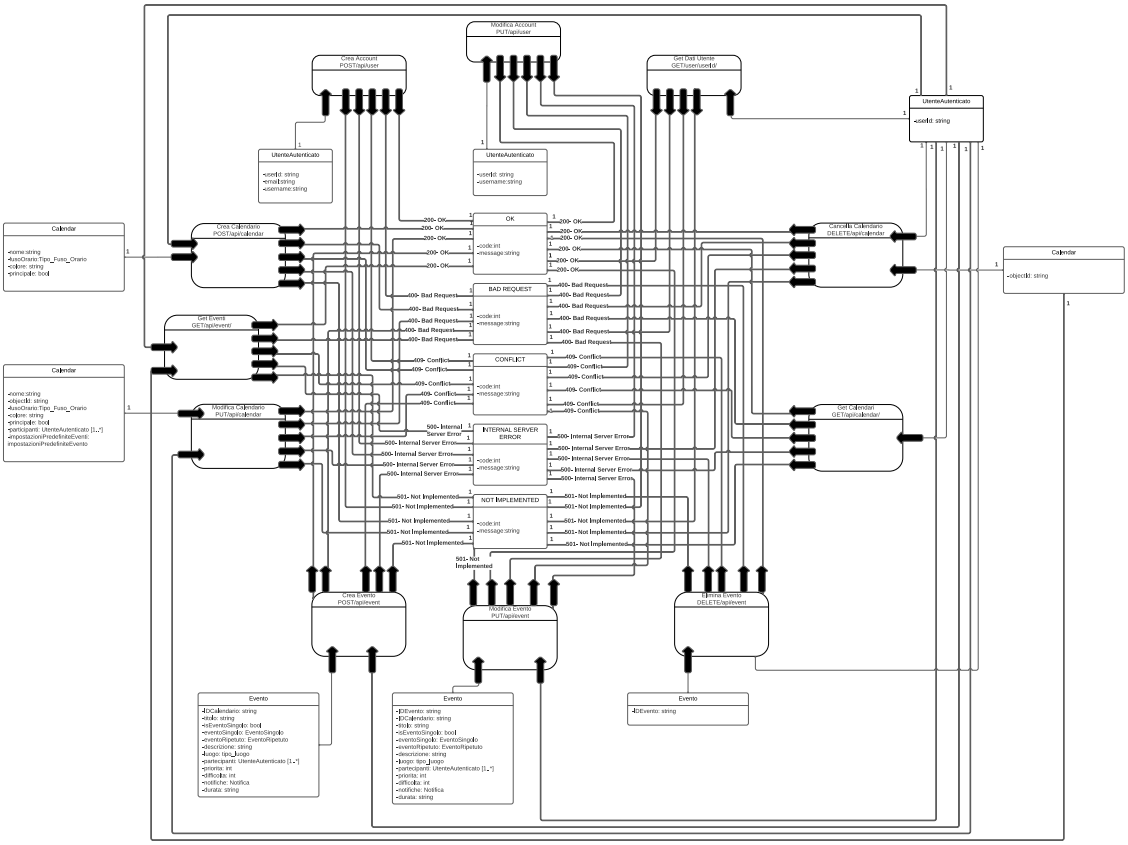
Immagine PNG/SVG Extract Diagram

4.2 : Resources Models

La seguente immagine mostra tutte le resources sviluppate nel nostro prototipo del sito PlanIt che sono state già presentate in parte nel precedente capitolo (APD4.1), ma in questo capitolo e nelle seguenti figure si va più nello specifico di come funzionano queste APIs andando a specificare in maniera dettagliata gli input e gli output possibili di ciascuna risorsa.
Abbiamo deciso di non presentare solo il diagramma totale di tutti i resources model, ovvero quello successivo, ma anche altri tre resources model, con lo scopo di far visualizzare con più facilità tutte le risorse. Le resources sono divise nei tre diversi diagrammi in base a come sono state già divise nell' Extract Diagram (APD4.1). C'è una cosa comune che si può notare prima di andare nel dettaglio della descrizione di ciascuna risorsa, ovvero che praticamente tutte le risorse danno le stesse possibile risposte/output, che sono:

- 200 OK. Serve ad indicare che la procedura effettuata dall'API è andata a buon fine e abbiamo ottenuto ciò che volevamo con successo. "200 OK" è la risposta standard per le richieste HTTP andate a buon fine.
- 400 BAD REQUEST. Serve ad indicare che la richiesta non può essere soddisfatta a causa di errori di sintassi, quindi abbiamo ottenuto un errore dall'esecuzione dell' API. Per errori di sintassi si intendono errori che riguardano soprattutto l'input inserito, il quale ha un formato sbagliato e quindi la procedura che deve effettuare l'API non può andare a buon fine e si riceve un messaggio di errore del tipo: "Wrong format for (nome del parametro che è stato inserito con il formato errato)". Oppure, non si sono inseriti tutti i parametri necessari affinché l'API possa funzionare e quindi riceviamo un messaggio di errore del tipo: "Parameter missing".
- 409 CONFLICT. Serve ad indicare che la richiesta non può essere portata a termine a causa di un conflitto con lo stato attuale della risorsa. Un caso che potrebbe portare ad un conflitto è ad esempio l'invio di un dato in input che è un duplicato di un dato già presente nel database, come un evento, calendario ed utente. Gli eventi, calendari e utenti duplicati inviati in input non possono essere creati e messi nel database, in quanto c'è il dato duplicato che crea conflitto. Questo errore lo otteniamo anche quando un determinato dato che stiamo passando non è presente nel database o non appartiene all'utente che lo sta inviando, questo può essere il caso in cui l' "userId" di un utente non esiste, oppure il "Calendario" o "Evento" passati con il loro "id" non appartengono a quell'utente identificato dall' "userId" inviato.
- 500 INTERNAL SERVER ERROR. Serve ad indicare un errore che riguardi il server interno. Questo errore non ha fatto andare a buon fine l'esecuzione dell' API, come ad esempio la sconnessione dal database MongoDB, dove sono inseriti i vari oggetti che vengono creati.
- 501 NOT IMPLEMENTED. Questo errore indica che non si è stati in grado di soddisfare il metodo della richiesta. Quindi si può descrivere come un errore generico, che non specifica nessun dettaglio sul perché non sia andata a buon fine l'esecuzione dell' API.

Si specifica che da questo momento in poi, verranno scritti assieme i codici di risultato con i messaggi di risposta per pura comodità. Infatti invece che scrivere "Tale API invia "400" come codice di risposta e messaggio di errore "Wrong format for (something)", verrà scritto direttamente "Tale API invia come messaggio di errore "400 - Wrong format for (something)".



Resources Model completo

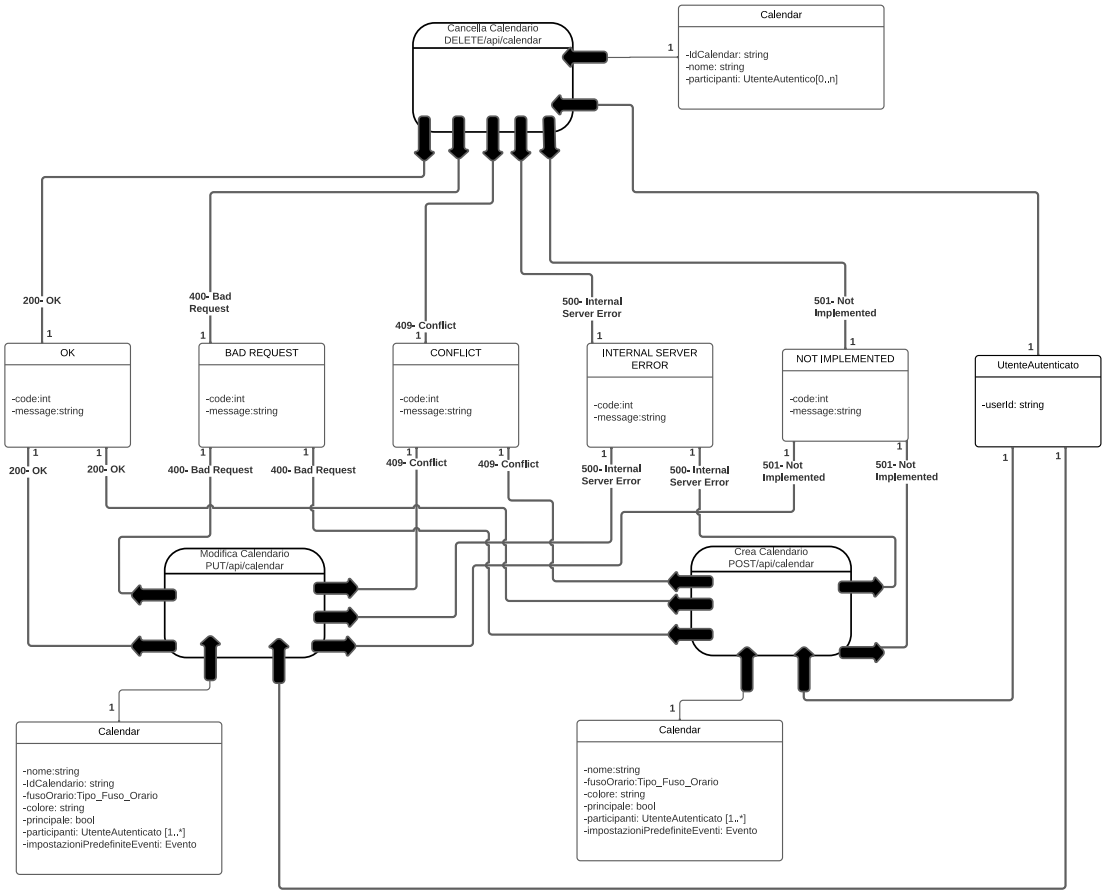
Immagine PNG/SVG Resources Model completo

4.2.1 : Resource Models "Calendario"

Nel seguente diagramma vengono presentate le APIs che riguardano la struttura dati "Calendario"; le risorse individuate per questa struttura dati, sono:

- "Elimina Calendario", resource di tipo DELETE. Questa API, già descritta in APD4.1, ha la funzione di eliminare un calendario specifico, dato il suo "IDCalendario" e l' "userId" dell'utente autenticato che vuole eliminare quel determinato calendario. Gli output sono quelli già descritti in precedenza, ma specifichiamo quali sono i casi in cui si ottiene l'errore "400". Otteniamo questo errore di risposta, quando manca un parametro (messaggio di errore "Parameter missing"), ovvero o "userId" o "IDCalendario". Invece, l'errore "409" si ottiene quando l' "userId" è un duplicato (messaggio di errore "There are too many users with that userId") o non esiste (messaggio di errore "There is no user with that userId") o l' "IDCalendario", del calendario che si vuole eliminare, non appartiene alla lista di calendari di quell'utente specificato con l' "userId" (messaggio di errore "You do not own the calendar").
- "Modifica Calendario", resource di tipo PUT. Questa API, già descritta in parte in APD4.1, ha la funzione di modificare un determinato calendario, dati tutti gli attributi che definiscono questa struttura dati modificata, e l' "userId" dell'utente che vuole modificare tale calendario. Specifichiamo, che si ottiene l'errore "400" ogniqualvolta non si è inserito uno degli attributi della struttura dati "Calendario", che si possono osservare nell'oggetto "Calendario" in input a questa API, con il messaggio "Parameter missing".
  Infine, questa resource ha anche gli errori che verranno specificati nella descrizione di "Crea Calendario", descritta qua sotto.
- "Crea Calendario", resource di tipo POST. Questa API, già descritta in parte in APD4.1, ha la funzione di creare e salvare nel database MongoDB il calendario che viene inviato in input per l'utente specificato con il parametro "userId". A differenza della resource PUT per il calendario, questa può avere anche dei campi vuoti in input senza che si ottenga un errore, ovvero questi campi: "fusoOrario", "colore", "principale" e "GestioneImpostazioniPredefiniteEventi". Invece i parametri "nome" e "userId" devono essere per forza non vuoti. Però, è da specificare che dagli attributi "fusoOrario", "colore" e "principali" si possono ottenere altri errori, ovvero:
  - errore "400" nel caso in cui il colore inserito avesse un formato sbagliato ("400" con messaggio di errore "Wrong format for color");
  - errore "400" nel caso in cui il fusoOrario inserito avesse un formato sbagliato, ovvero "GMTOffset" deve stare tra -12 e +12 e attributo "localita" deve essere diverso da "null" ("400" con messaggio di errore "Wrong format for fusoOrario");
  - errore "409" nel caso in cui principale fosse uguale a "true", ovvero stiamo creando il calendario principale di un utente, ma questo utente ha già un calendario principale. Infatti non si possono avere più di un calendario principale, quindi otteniamo come risposta: "409 - There are too many primary calendars";

  Infine, l'errore "409" si potrebbe ottenere anche quando l' "userId" dell'utente che sta creando il calendario è un duplicato o non esiste nel database. Si ottiene come messaggio di errore: "There are too many users with that userId" o "There is no user with that userId" rispettivamente.



Resources Model riguardo solo il "Calendario"

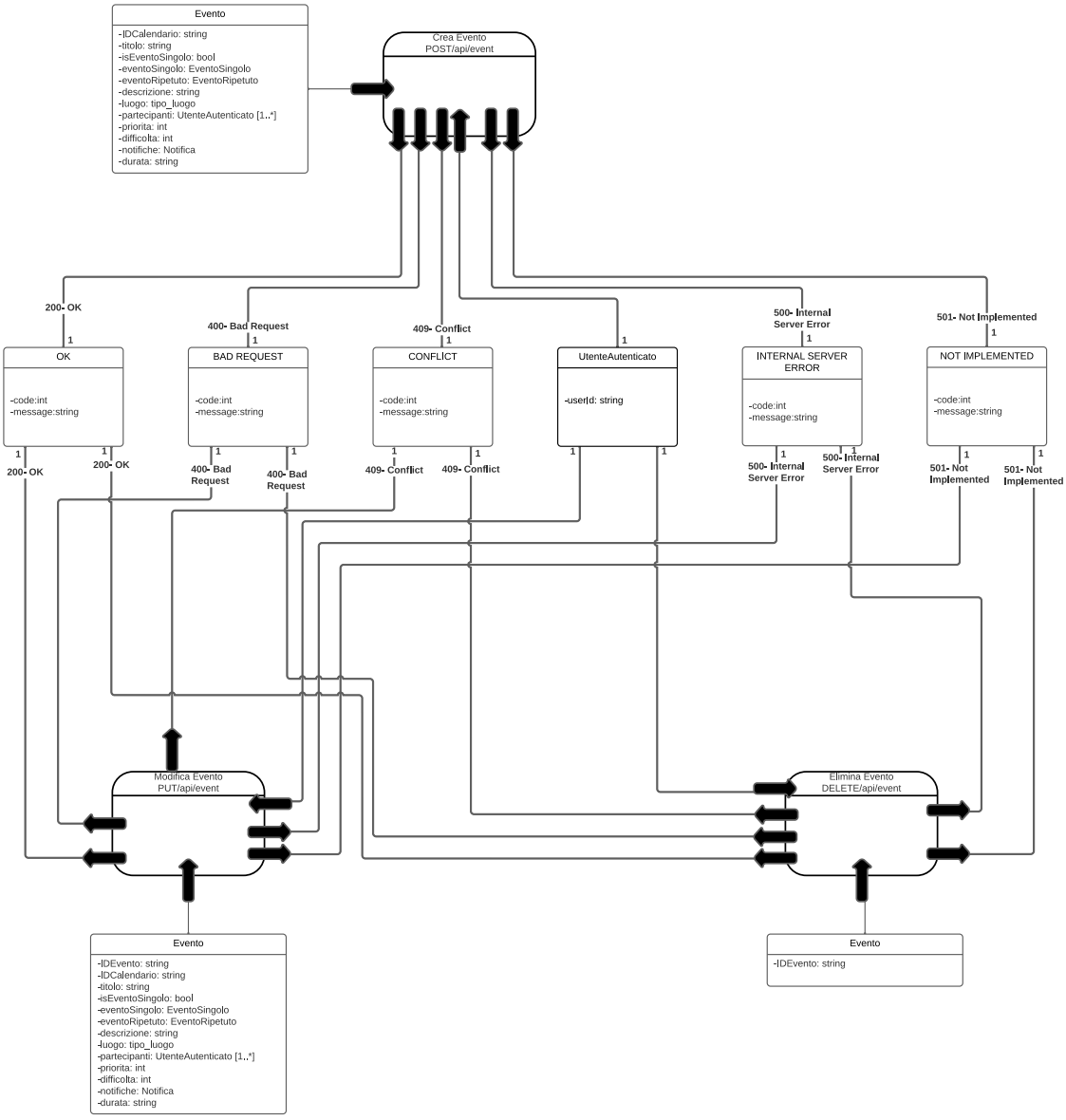
4.2.2 : Resource Models "Evento"

Nel seguente diagramma vengono mostrate le APIs che riguardano la struttura dati "Evento", già presentate in parte in APD4.1; le APIs individuate per "Evento" sono:

- "Crea Evento", resource di tipo POST. Questa API ha lo scopo di andare a salvare un evento dato in input. Gli output sono sempre quelli citati in generale in precedenza, ma si specifica che i parametri che sono obbligatori, per non ottenere l'errore "400 - IDCalendario or titolo or evento details missing", sono: "IDCalendario", id del calendario in cui stiamo aggiungendo questo evento, "titolo", titolo dell'evento, "userId", id dell'utente che sta creando l'evento, "isEventoSingolo", booleano che indica se l'evento che si sta creando sia un evento singolo o ripetuto. Inoltre se "isEventoSingolo" è uguale a true si deve avere l'oggetto "eventoSingolo" diverso da "null", invece se questo è uguale a false, "eventoRipetuto" deve essere diverso da "null".
- "Modifica Evento", resource di tipo PUT. Questa resource ha lo scopo di andare a modificare un evento specifico appartenente ad un utente autenticato. A differenza dell' API "Crea Evento", questa API ha bisogno di tutti gli attributi che costituiscono la struttura dati "Evento" eccetto eventoSingolo o eventoRipetuto, che possono essere uguali a "null", a seconda del valore del booleano isEventoSingolo, per i casi già citati per "Crea Evento". Dunque per non ricevere l'errore "400 - Parameter missing", questa resource ha bisogno di tutti gli attributi di "Evento" che si possono osservare nel diagramma successivo. Gli altri errori che si possono ottenere da questa resource sono gli stessi già citati in precedenza in generale; ovviamente visto che i parametri da inserire obbligatoriamente sono di più, è più facile che si possa sbagliare e ottenere l'errore "400 - Wrong formato for (inserted parameter uncorrectly)".

Immagine PNG/SVG Resources Model riguardo solo il "Calendario"

- "Elimina Evento", resource di tipo DELETE. Questa API ha lo scopo di eliminare un specifico evento di un utente autenticato, per questo motivo i parametri necessari sono: "IDEvento", "id" dell'evento che si vuole eliminare, e "userId", "id" dell'utente autenticato che vuole eliminare un suo evento. Se manca uno dei due parametri, ovviamente si ottiene l'errore "400 - Parameter missing". Gli altri errori che si possono ottenere da questa API, sono quelli già descritti, con particolare attenzione al caso in cui viene inviato l' "IDEvento" di un evento che non appartiene a quel dato "userId"; in questo caso, si riceve l'errore "400 - You do not own the event".



Resources Model riguardo solo l' "Evento"

4.2.3 : Resource Models "GestioneChiamateMongoDB"

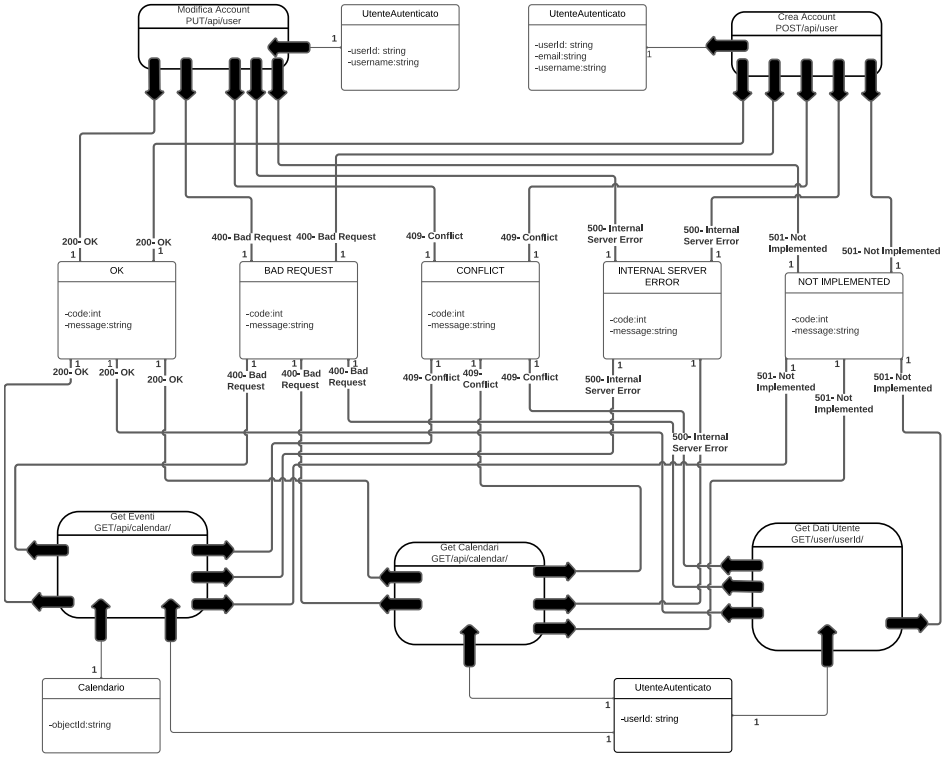
Nel seguente diagramma vengono mostrate le APIs che riguardano la classe "GestioneChiamateMongoDB", già presentate in parte in APD4.1; le APIs individuate per "GestioneChiamateMongoDB" sono:

- "Modifica Account", resource di tipo PUT. La funzione di questa API è di modificare l' "username" di un utente autenticato, dato il suo "userId" e il nuovo "username"; ovviamente, per effettuare tale azione e non riceve il messaggio di errore "400 - Parameter missing", i parametri sopra citati devono essere sempre

Immagine PNG/SVG Resources Model riguardo solo l' "Evento"

presenti. Gli altri errori che si possono riscontrare sono sempre i soliti presentati in generale all'inizio di questo capitolo.

- "Crea Account", resource di tipo POST. Lo scopo di questa resource è quella di creare un account all'interno del database, dato l' "email", l' "username" e l' "userId" dell'utente autenticato; è importante specificare che tutti questi tre parametri devono essere diversi da "null" per non ottenere il messaggio di errore "400 - Parameter missing". Gli altri errori e risposte che si possono riscontrare sono gli stessi già citati.
- "Get Dati Utente" e "Get Dati Utente_email", APIs di tipo GET. Si descrivono assieme queste APIs, in quanto hanno la stessa funzione, ovvero ottenere i dati dell'account di un utente ("email", "userId", "username"), ma dando parametri diversi: per "Get Dati Utente" basta inviare l' "userId" dell'utente autenticato, invece per "Get Dati Utente_email" basta la propria "email". Si è deciso di mettere entrambi i GET, in quanto entrambi gli attributi "email" e "userId" sono univoci per ciascun utente autenticato. Ovviamente, per non ottenere il messaggio di errore "400 - Parameter missing", il parametro che deve essere sempre presente e diverso da "null" è l' "userId" e l' "email" rispettivamente. Gli altri errori e messaggi che si possono ricevere sono gli stessi già descritti in generale all'inizio del capitolo.
- "Get Calendari", resource di tipo GET. Resource che ha lo scopo di ottenere tutti i calendari appartenenti ad un utente, dato il suo "userId": questo, ovviamente, deve essere non vuoto per non riceveere il messaggio di errore "400 - Parameter missing". Gli altri errori e messaggi che si possono ricevere sono gli stessi già descritti in generale all'inizio del capitolo.
- "Get Eventi", resource di tipo GET. API che ha la funzione di ritornare tutti gli eventi di un dato calendario. I parametri necessari, affinché questa resource possa effettuare il suo lavoro, sono l' "IDCalendario", "id" del calendario di cui vogliamo ottenere gli eventi, e l' "userId" dell'utente che ha tale calendario. Questi devono essere diversi da "null" per non ricevere il messaggio di errore "400 - Parameter missing". Gli altri errori e messaggi che si possono ricevere sono gli stessi già descritti in generale all'inizio del capitolo.



Resources Model riguardo solo "GestioneChiamateMongoDB"

APD5 : Sviluppo API

In questo capitolo verranno presentate le API delle resources già sopra descritte, però andando più nello specifico mostrando il codice di come queste resources sono state implementate.

5.1 : Crea Calendario

Mediante questa API, l'applicativo salva nel database MongoDB un calendario con gli attributi uguali ai parametri ricevuti in input, per l'utente autenticato identificato dal suo "userId", parametro preso in input da questa funzione. Come già detto in precedenza nella descrizione della resource corrispondente, nel caso in cui mancasse l'attributo "nome" o l' "userId" non sarebbe possibile creare un calendario e quindi l'API mandarebbe come messaggio di risposta il seguente errore: "400 - Name missing". Inoltre, come già descritto, questa API controlla che il "colore" e "fusoOrario" inseriti rispettino degli standard affinché siano ritenuti accettabili: in caso non li rispettassero, riceveremmo rispettivamente i seguenti errori: "400 - Wrong format for color", "400 - Wrong format for fusoOrario".
Mediante la funzione "find()", l' API va a cercare nel database, gli utenti autenticati che hanno l' "userId" inserito; nel caso in cui trovasse più di un utente con questo "userId", si riceve un messaggio di errore del tipo "400 - There are too many users with that userId"; invece, nel caso in cui trovasse nessun utente autenticato con tale "userId" si riceve il messaggio di errore "400 - There is no user with that userId".
Infine, l'ultimo controllo che si fa sugli attributi ricevuti in input riguarda l'attributo "principale", booleano che indica se il calendario che si sta creando sia quello principale o meno. Infatti, ogni utente autenticato può avere al massimo di un calendario principale, per questo motivo nel caso in cui si provasse a creare un calendario principale già esistente per quell' "userId", si riceve il messaggio di errore: "409 - There are too many primary calendars".
Grazie, alla funzione "create()", viene creato nel database un calendario con i valori dei campi che lo formano uguali ai parametri inseriti in input; nel caso ci fossero degli errori nell'esecuzione di questa funzione, otteniamo l'errore "500 - Not inserted". Se, invece l'esecuzione di questa funzione è andata a buon fine, otteniamo il messaggio di successo "200 - Calendar inserted correctly". C'è un eventuale altro controllo se ci fosse un altro errore generico durante l'esecuzione di creaCalendario, quindi otteniamo un messaggio di errore del tipo: "501 - Generic error".

Immagine PNG/SVG Resources Model riguardo solo "GestioneChiamateMongoDB"

```
export async function creaCalendario(req, res) {
  await dbConnect();
  try {
    const { nome, fusoOrario, colore, principale } = req.query;
    const { userId } = req.query;

    if (nome == null || userId == null) {
      res.status(400).json({ error: "Name missing" }); //TODO or userID
      return;
    }

    if (colore != null && !/^#([0-9a-f]{3}){1,2}$/i.test(colore)) {
      res.status(400).json({ error: "Wrong format for color" });
      return;
    }

    let tempFusoOrario

    if (fusoOrario != null) {
      try{
        tempFusoOrario = JSON.parse(fusoOrario)
      }catch{
        tempFusoOrario = fusoOrario
      }

      if (
        tempFusoOrario.GMTOffset == null ||
        tempFusoOrario.localita == null ||
        tempFusoOrario.GMTOffset > 12 ||
        tempFusoOrario.GMTOffset < -12
      ) {
        res.status(400).json({ error: "Wrong format for time zone" });
        return;
      }
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }

    if (principale === "true" || principale == true) {
      const calendariPrincipali = await Calendario.find({
        $and: [{ partecipanti: userId }, { principale: true }],
      });
      if (Object.keys(calendariPrincipali).length >= 1) {
        res.status(409).json({
          error: "There are too many primary calendars",
        });
        return;
      }
    }

    Calendario.create(
      {
        nome: nome,
        fusoOrario: fusoOrario == null ? undefined : tempFusoOrario,
        colore: colore == null ? undefined : colore,
        partecipanti: [userId],
        principale: principale == null ? false : principale,
        impostazioniPredefiniteEventi: undefined,
      },
      function (err, calendar) {
        if (err) {
          res.status(500).json({ error: "Not inserted" });
          return;
        }
      },
    );

    res.status(200).json({ success: "Calendar inserted correctly" });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - creaCalendario

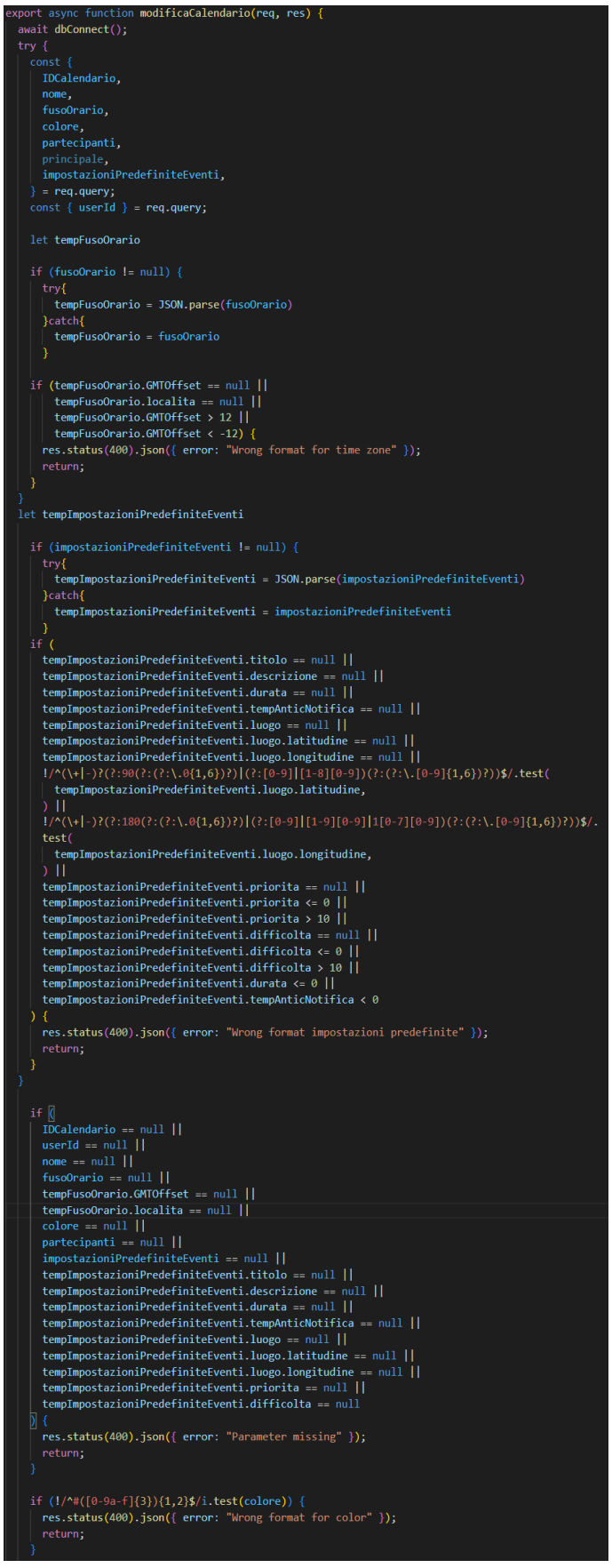
Immagine PNG API - creaCalendario

5.2 : **Modifica Calendario**

Grazie a questa API, PlanIt riesce a modificare le impostazioni di un calendario già salvato per un utente autenticato. Per fare tale operazione, oltre all' "IDCalendario", "id" del calendario che si vuole modificare, e l' "userId", "id" dell'utente autenticato che vuole modificare un proprio calendario, sono necessari anche tutti gli altri attributi che costituiscono la struttura dati "Calendario", già molte volte mostrata, con i quali si andrà a modificare il calendario. Nel caso in cui mancasse uno di questi attributi, si riceve il messaggio di errore: "400 - Parameter missing".
Come si può notare dal codice, in "modificaCalendario" sono presenti gli stessi controlli già descritti in "creaCalendario", ma con l'aggiuta di qualcuno di nuovo. Infatti per l'oggetto "impostazioniPredefiniteEventi", che ricordiamo essere l'insieme di valori con cui vengono precompilati gli eventi appartenenti ad un calendario specifico, valori che possono essere modificati sia durante la creazione che modifica di un evento in fase di compilazione, vengono fatti ulteriori accertamenti. In primo luogo, vengono controllati che ciascun campo che lo forma sia diverso da "null", inoltre:

- la "latitudine" e "longitudine" devono avere dei valori coerenti con la realtà (la longitudine deve essere nell'intervallo [-180,+180], invece la latitudine [-90,+90]);
- la "priorità" e "difficoltà" deve avere un valore tra 0 e 10;
- la "durata" deve essere maggiore di 0
- "tempAnticNotifica", attributoche indica quanto prima inviare la notifica di un evento, deve essere maggiore e uguale a 0.

Nel caso in cui uno delle restrizioni sopra citate non fossero rispettate per "impostazioniPredefiniteEventi", si riceverebbe il messaggio di errore "400 - Wrong format impostazioni predefinite".
Un altro controllo che viene fatto, riguarda se il calendario che si vuole modificare appartiene alla lista di calendari di quell'utente autenticato identificato dal suo "userId". Questo controllo viene fatto grazie alla funzione "find" che va a trovare nel database il calendario con l' "IDCalendario" inserito, e dopo aver ottenuto questo oggetto, si controlla che il suo proprietario (primo utente presente nella lista di partecipanti al calendario, attributo che indica chi partecipa ad un calendario, ovvero ai suoi eventi) sia uguale all' "userId" inserito in input. Nel caso non lo fosse, si riceve il messaggio di errore: "409 - You do not own the calendar".
Infine, invece che usare la funzione "create()", usata per salvare un calendario nel database in "creaCalendario", viene usata la funzione "updataMany()" che va ad aggiornare gli attributi che costituiscono il calendario che si vuole modificare secondo i parametri ricevuti in input.



```
export async function modificaCalendario(req, res) {
  await dbConnect();
  try {
    const {
      IDCalendario,
      nome,
      fusoOrario,
      colore,
      partecipanti,
      principale,
      impostazioniPredefiniteEventi,
    } = req.query;
    const { userId } = req.query;

    let tempFusoOrario

    if (fusoOrario != null) {
      try{
        tempFusoOrario = JSON.parse(fusoOrario)
      }catch{
        tempFusoOrario = fusoOrario
      }

    if (tempFusoOrario.GMTOffset == null ||
        tempFusoOrario.localita == null ||
        tempFusoOrario.GMTOffset > 12 ||
        tempFusoOrario.GMTOffset < -12) {
      res.status(400).json({ error: "Wrong format for time zone" });
      return;
    }
  }
  let tempImpostazioniPredefiniteEventi

    if (impostazioniPredefiniteEventi != null) {
      try{
        tempImpostazioniPredefiniteEventi = JSON.parse(impostazioniPredefiniteEventi)
      }catch{
        tempImpostazioniPredefiniteEventi = impostazioniPredefiniteEventi
      }
    if (
      tempImpostazioniPredefiniteEventi.titolo == null ||
      tempImpostazioniPredefiniteEventi.descrizione == null ||
      tempImpostazioniPredefiniteEventi.durata == null ||
      tempImpostazioniPredefiniteEventi.tempAnticNotifica == null ||
      tempImpostazioniPredefiniteEventi.luogo == null ||
      tempImpostazioniPredefiniteEventi.luogo.latitudine == null ||
      tempImpostazioniPredefiniteEventi.luogo.longitudine == null ||
      !/^(\+|-)?(?:90(?:(?:\.0{1,6})?)|(?:[0-9]|[1-8][0-9])(?:(?:\.[0-9]{1,6})?))$/.test(
        tempImpostazioniPredefiniteEventi.luogo.latitudine,
      ) ||
      !/^(\+|-)?(?:180(?:(?:\.0{1,6})?)|(?:[0-9]|[1-9][0-9]|1[0-7][0-9])(?:(?:\.[0-9]{1,6})?))$/.
      test(
        tempImpostazioniPredefiniteEventi.luogo.longitudine,
      ) ||
      tempImpostazioniPredefiniteEventi.priorita == null ||
      tempImpostazioniPredefiniteEventi.priorita <= 0 ||
      tempImpostazioniPredefiniteEventi.priorita > 10 ||
      tempImpostazioniPredefiniteEventi.difficolta == null ||
      tempImpostazioniPredefiniteEventi.difficolta <= 0 ||
      tempImpostazioniPredefiniteEventi.difficolta > 10 ||
      tempImpostazioniPredefiniteEventi.durata <= 0 ||
      tempImpostazioniPredefiniteEventi.tempAnticNotifica < 0
    ) {
      res.status(400).json({ error: "Wrong format impostazioni predefinite" });
      return;
    }
  }

    if (
      IDCalendario == null ||
      userId == null ||
      nome == null ||
      fusoOrario == null ||
      tempFusoOrario.GMTOffset == null ||
      tempFusoOrario.localita == null ||
      colore == null ||
      partecipanti == null ||
      impostazioniPredefiniteEventi == null ||
      tempImpostazioniPredefiniteEventi.titolo == null ||
      tempImpostazioniPredefiniteEventi.descrizione == null ||
      tempImpostazioniPredefiniteEventi.durata == null ||
      tempImpostazioniPredefiniteEventi.tempAnticNotifica == null ||
      tempImpostazioniPredefiniteEventi.luogo == null ||
      tempImpostazioniPredefiniteEventi.luogo.latitudine == null ||
      tempImpostazioniPredefiniteEventi.luogo.longitudine == null ||
      tempImpostazioniPredefiniteEventi.priorita == null ||
      tempImpostazioniPredefiniteEventi.difficolta == null
    ) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    if (!/^#([0-9a-f]{3}){1,2}$/i.test(colore)) {
      res.status(400).json({ error: "Wrong format for color" });
      return;
    }
```

API - modificaCalendario, 1a parte

Immagine PNG API - modificaCalendario

```
    if (!/^#([0-9a-f]{3}){1,2}$/i.test(colore)) {
      res.status(400).json({ error: "Wrong format for color" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }
    var ObjectId = require("mongoose").Types.ObjectId;

    const calendariPosseduti = await Calendario.find({
      $and: [{ partecipanti: userId }, { _id: new ObjectId(IDCalendario) }],
    });
    if (
      Object.keys(calendariPosseduti).length == 0 ||
      calendariPosseduti[0].partecipanti[0] != userId
    ) {
      res.status(409).json({
        error: "You do not own the calendar",
      });
      return;
    }

    Calendario.updateMany(
      { _id: new ObjectId(IDCalendario) },
      {
        nome: nome,
        fusoOrario: tempFusoOrario,
        colore: colore,
        partecipanti: partecipanti,
        impostazioniPredefiniteEventi: tempImpostazioniPredefiniteEventi,
      },
      function (err, calendar) {
        if (err) {
          res.status(500).json({ error: "Not edited" });
          return;
        }
      },
    );

    res.status(200).json({ success: "Calendar updated correctly" });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - modificaCalendario, 2a parte

Immagine PNG API - modificaCalendario, 2a parte

5.3 : **Elimina Calendario**

Lo scopo di questa API, come già descritto in APD4.2.1, è quella di eliminare un calendario, dato il suo "IDCalendario" e l' "userId", per questo motivo questi due parametri non devono essere vuoti, sennò si riceve il messaggio di errore "400 - Parameter missing". Anche in questa API, come in "modificaCalendario", viene controllato che l' "userId" inviato non sia un duplicato e che esista, che il calendario che si vuole eliminare appartenga alla lista di calendari dell'utente autenticato identificato dall' "userId" e vengono fatti anche tutti gli altri controlli già sopra descritti.
Infine, per eliminare il calendario dal database viene usata la funzione "deleteMany()" che sfrutta solo l' "IDCalendario" inviato in input per identificare il calendario da eliminare. Nel caso in cui non fosse eliminato nessun calendario per qualche motivo, c'è l'invio dell'errore "500 - Calendar not deleted"; un motivo che porta questo errore è la sconnessione improvvisa dal database MongoDB.

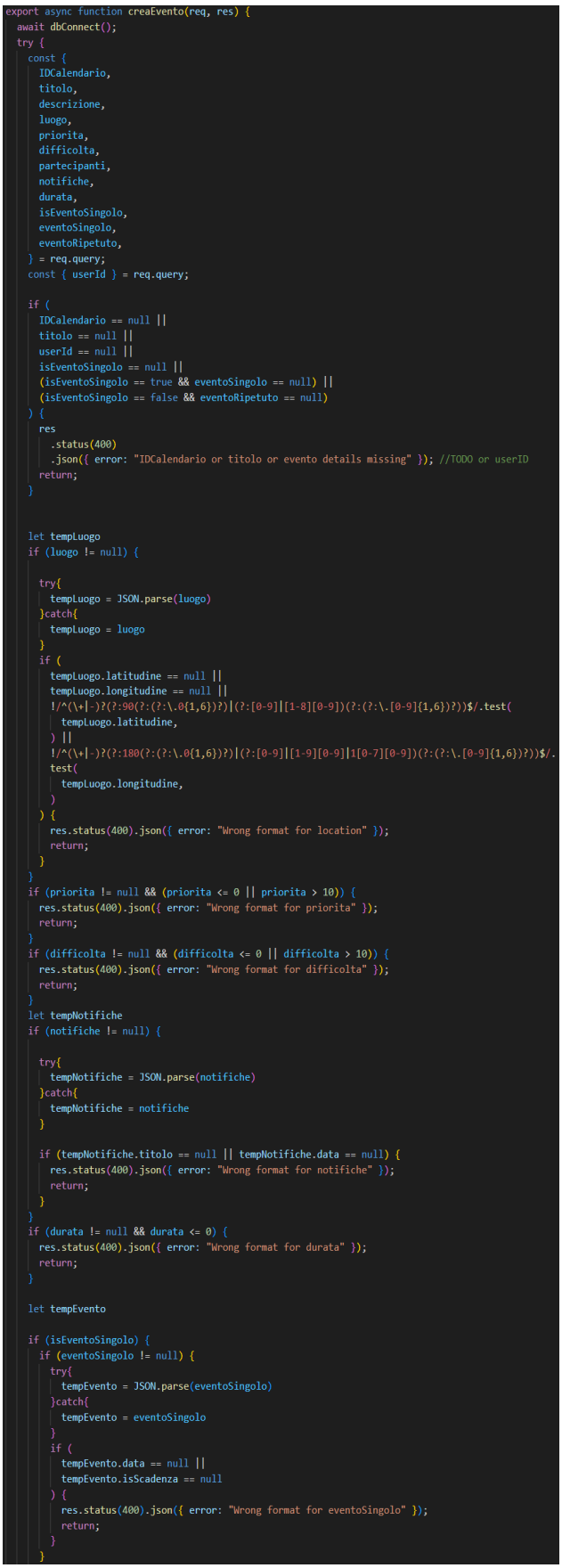
```
export async function eliminaCalendario(req, res) {
  await dbConnect();
  try {
    const { IDCalendario } = req.query;
    const { userId } = req.query;

    if (IDCalendario == null || userId == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }
    var ObjectId = require("mongoose").Types.ObjectId;

    const calendariPosseduti = await Calendario.find({
      $and: [{ partecipanti: userId }, { _id: new ObjectId(IDCalendario) }],
    });
    if (
      Object.keys(calendariPosseduti).length == 0 ||
      calendariPosseduti[0].partecipanti[0] != userId
    ) {
      res.status(409).json({
        error: "You do not own the calendar",
      });
      return;
    }

    const deleteCalendar = await Calendario.deleteMany({
      _id: new ObjectId(IDCalendario),
    });

    if (deleteCalendar.deletedCount >= 1) {
      res.status(200).json({ success: "Calendar deleted correctly" });
      return;
    } else {
      res.status(500).json({ error: "Calendar not deleted" });
      return;
    }
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - eliminaCalendario

Immagine PNG API - eliminaCalendario

5.4 : Crea Evento

Mediante questa API, l'applicativo PlanIt riesce a salvare nel database MongoDB un evento con gli attributi uguali ai parametri inviati in input, per l'utente autenticato identificato dall' "userId" indicato. Già in APD4.2.2, nella descrizione della risorsa "Crea Evento", siamo andati a presentare quali sono i casi in cui questa API invia come messaggio di errore "400 - IDCalendario or titolo or evento details missing", per questo motivo non andremo a ripeterli nuovamente.
Come si può notare per gli attributi "luogo", "prorita", "difficolta", "notifiche" e "durata" vengono fatti gli stessi controlli già presenti in "modificaCalendario" per l'attributo "impostazioniPredefiniteEventi", in quanto quest'ultimo oggetto contiene gli attributi sopra citati, presenti anche durante la crazione e modifica di un evento. Nel caso in cui uno di questi attributi non passasse un controllo, riceviamo un errore del tipo "400 - Wrong format (something, name of the attribute)".
I controlli non presenti nelle altre API, già descritte, riguardano gli attributi "eventoSingolo" ed "eventoRipetuto", oggetti che contengono informazioni fondamentali per creare un evento singolo, evento che poniamo solo in un giorno in fase di creazione, e un evento ripetuto, evento poniamo su più giornate in fase di creazione. Per la struttura dati "eventoSingolo" andiamo a controllare che gli attributi "data" e "isScadenza" (questo attributo indica se l'evento che si sta mettendo con tale data sia una deadline di qualcosa oppure no) siano diversi da "null": se lo fossero riceviamo il messaggio di errore "400 - Wrong format for eventoSingolo". Invece, per la struttura dati "eventoRipetuto" andiamo ad controllare che "numeroRipetizione", attributo che indica quante volte si ripete tale evento, sia diverso da "null" e maggiore e uguale di 1, che "data", attributo che indica il giorno, mese, anno e orario dell'evento, sia diverso da "null" e che, infine, "giornidellaSettimana", attributo che indica in quali giorni si ripete tale evento, sia diverso da "null" che non sia un array vuoto. Se questi attributi non rispettassero queste restrizioni, riceviamo l'errore "400 - Wrong format for eventoRipetuto".
Dopo sono presenti i soliti controlli già citati, ma evidenziamo la presenza della funzione "find()" che viene utilizzata per andare a vedere se il calendario, dove stiamo aggiungendo l'evento da che si sta creando, esista e appartenga all'utente autenticato identificato dall' "userId"; se così non fosse, otteniamo il messaggio di errore "409 - There is no calendar with that ID or you do not own the calendar". Come per "creaCalendario", viene usata la funzione "create()" per andare a creare nel database MongoDB un oggetto "Evento" con le caratteristiche dei parametri ricevuti in input.



```
export async function creaEvento(req, res) {
  await dbConnect();
  try {
    const {
      IDCalendario,
      titolo,
      descrizione,
      luogo,
      priorita,
      difficolta,
      partecipanti,
      notifiche,
      durata,
      isEventoSingolo,
      eventoSingolo,
      eventoRipetuto,
    } = req.query;
    const { userId } = req.query;

    if (
      IDCalendario == null ||
      titolo == null ||
      userId == null ||
      isEventoSingolo == null ||
      (isEventoSingolo == true && eventoSingolo == null) ||
      (isEventoSingolo == false && eventoRipetuto == null)
    ) {
      res
        .status(400)
        .json({ error: "IDCalendario or titolo or evento details missing" }); //TODO or userID
      return;
    }


    let tempLuogo
    if (luogo != null) {

      try{
        tempLuogo = JSON.parse(luogo)
      }catch{
        tempLuogo = luogo
      }
      if (
        tempLuogo.latitudine == null ||
        tempLuogo.longitudine == null ||
        !/^(\+|-)?(?:90(?:(?:\.0{1,6})?)|(?:[0-9]|[1-8][0-9])(?:(?:\.[0-9]{1,6})?))$/.test(
          tempLuogo.latitudine,
        ) ||
        !/^(\+|-)?(?:180(?:(?:\.0{1,6})?)|(?:[0-9]|[1-9][0-9]|1[0-7][0-9])(?:(?:\.[0-9]{1,6})?))$/.
        test(
          tempLuogo.longitudine,
        )
      ) {
        res.status(400).json({ error: "Wrong format for location" });
        return;
      }
    }
    if (priorita != null && (priorita <= 0 || priorita > 10)) {
      res.status(400).json({ error: "Wrong format for priorita" });
      return;
    }
    if (difficolta != null && (difficolta <= 0 || difficolta > 10)) {
      res.status(400).json({ error: "Wrong format for difficolta" });
      return;
    }
    let tempNotifiche
    if (notifiche != null) {

      try{
        tempNotifiche = JSON.parse(notifiche)
      }catch{
        tempNotifiche = notifiche
      }

      if (tempNotifiche.titolo == null || tempNotifiche.data == null) {
        res.status(400).json({ error: "Wrong format for notifiche" });
        return;
      }
    }
    if (durata != null && durata <= 0) {
      res.status(400).json({ error: "Wrong format for durata" });
      return;
    }

    let tempEvento

    if (isEventoSingolo) {
      if (eventoSingolo != null) {
        try{
          tempEvento = JSON.parse(eventoSingolo)
        }catch{
          tempEvento = eventoSingolo
        }
        if (
          tempEvento.data == null ||
          tempEvento.isScadenza == null
        ) {
          res.status(400).json({ error: "Wrong format for eventoSingolo" });
          return;
        }
      }
```

API - creaEvento, 1a parte

Immagine PNG API - creaEvento, 1a parte

```
if (isEventoSingolo) {
  if (eventoSingolo != null) {
    try{
      tempEvento = JSON.parse(eventoSingolo)
    }catch{
      tempEvento = eventoSingolo
    }
    if (
      tempEvento.data == null ||
      tempEvento.isScadenza == null
    ) {
      res.status(400).json({ error: "Wrong format for eventoSingolo" });
      return;
    }
  }
} else {
  if (eventoRipetuto != null) {
    try{
      tempEvento = JSON.parse(eventoRipetuto)
    }catch{
      tempEvento = eventoRipetuto
    }
    if (
      tempEvento.numeroRipetizioni == null ||
      tempEvento.impostazioniAvanzate == null ||
      tempEvento.impostazioniAvanzate.giorniSettimana == null ||
      tempEvento.impostazioniAvanzate.data == null ||
      tempEvento.numeroRipetizioni < 1 ||
      tempEvento.impostazioniAvanzate.giorniSettimana == []
    ) {
      res.status(400).json({ error: "Wrong format for eventoRipetuto" });
      return;
    }
  }
}

const users = await UtenteAutenticato.find({
  userId: userId,
});
if (Object.keys(users).length == 0) {
  res.status(409).json({ error: "There is no user with that userId" });
  return;
} else if (Object.keys(users).length > 1) {
  res
    .status(409)
    .json({ error: "There are too many users with that userId" });
  return;
}

var ObjectId = require("mongoose").Types.ObjectId;

const calendariPosseduti = await Calendario.find({
  $and: [{ partecipanti: userId }, { _id: new ObjectId(IDCalendario) }],
});
if (
  Object.keys(calendariPosseduti).length == 0 ||
  calendariPosseduti[0].partecipanti[0] != userId
) {
  res.status(409).json({
    error:
      "There is no calendar with that ID or you do not own the calendar",
  });
  return;
}


if (isEventoSingolo) {
  Evento.create(
    {
      IDCalendario: IDCalendario,
      titolo: titolo,
      descrizione: descrizione == null ? undefined : descrizione,
      luogo: luogo == null ? undefined : tempLuogo,
      priorita: priorita == null ? undefined : priorita,
      difficolta: difficolta == null ? undefined : difficolta,
      partecipanti:
        partecipanti == null
          ? calendariPosseduti[0].partecipanti
          : partecipanti,
      notifiche: notifiche == null ? undefined : tempNotifiche,
      durata: durata == null ? undefined : durata,
      isEventoSingolo: true,
      eventoSingolo: eventoSingolo == null ? undefined : tempEvento,
    },
    function (err, calendar) {
      if (err) {
        res.status(500).json({ error: "Not inserted" });
        return;
      }
    },
  );
```

API - creaEvento, 2a parte

Immagine PNG API - creaEvento, 2a parte

```
    } else {
      Evento.create(
        {
          IDCalendario: IDCalendario,
          titolo: titolo,
          descrizione: descrizione == null ? undefined : descrizione,
          luogo: luogo == null ? undefined : tempLuogo,
          priorita: priorita == null ? undefined : priorita,
          difficolta: difficolta == null ? undefined : difficolta,
          partecipanti:
            partecipanti == null
              ? calendariPosseduti[0].partecipanti
              : partecipanti,
          notifiche: notifiche == null ? undefined : tempNotifiche,
          durata: durata == null ? undefined : durata,
          isEventoSingolo: false,
          eventoRipetuto: eventoRipetuto == null ? undefined : tempEvento,
        },
        function (err, calendar) {
          if (err) {
            res.status(500).json({ error: "Not inserted" });
            return;
          }
        },
      );
    }

    res.status(200).json({ success: "Event inserted correctly" });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - creaEvento, 3a parte

Immagine PNG API - creaEvento, 3a parte

5.5 : **Modifica Evento**

Grazie a questa API, è possibile modificare un "Evento" già salvato nel database MongoDB, con i vari parametri ricevuti in input, che non sono altro che gli attributi che formano la struttura dati "Evento", struttura dati già più volte mostrata, e l' "userId" dell'utente autenticato che vuole modificare tale evento. I controlli, presenti in questa API, sono gli stessi già descritti per "creaEvento" (APD5.4) e "modificaCalendario" (APD5.2); l'unica differenza è che per non ricevere il messaggio di errore "400 - Parameter missing", tutti gli attributi che formano l'oggetto "Evento" e l' "userId" devono essere diversi da "null".

Alla fine della funzione, grazie alla funzione "updateMany()", viene aggiornato l'oggetto "Evento" presente nel database secondo i parametri ricevuti, come per "modificaCalendario" (APD5.2). Specifichiamo che a seconda se stiamo andando a modificare un "eventoSingolo" o "eventoRipetuto" abbiamo una diversa procedura di aggiornamento.

```
export async function modificaEvento(req, res) {
  await dbConnect();
  try {
    const {
      IDEvento,
      IDCalendario,
      titolo,
      descrizione,
      luogo,
      priorita,
      difficolta,
      partecipanti,
      notifiche,
      durata,
      isEventoSingolo,
      eventoSingolo,
      eventoRipetuto,
    } = req.query;
    const { userId } = req.query;



    let tempLuogo
    if (luogo != null) {

      try{
        tempLuogo = JSON.parse(luogo)
      }catch{
        tempLuogo = luogo
      }
      if (
        tempLuogo.latitudine == null ||
        tempLuogo.longitudine == null ||
        !/^(\+|-)?(?:90(?:(?:\.0{1,6})?)|(?:[0-9]|[1-8][0-9])(?:(?:\.[0-9]{1,6})?))$/.test(
          tempLuogo.latitudine,
        ) ||
        !/^(\+|-)?(?:180(?:(?:\.0{1,6})?)|(?:[0-9]|[1-9][0-9]|1[0-7][0-9])(?:(?:\.[0-9]{1,6})?))$/.
        test(
          tempLuogo.longitudine,
        )
      ) {
        res.status(400).json({ error: "Wrong format for location" });
        return;
      }
    }
    if (priorita <= 0 || priorita > 10) {
      res.status(400).json({ error: "Wrong format for priorita" });
      return;
    }
    if (difficolta <= 0 || difficolta > 10) {
      res.status(400).json({ error: "Wrong format for difficolta" });
      return;
    }
    let tempNotifiche
    if (notifiche != null) {

      try{
        tempNotifiche = JSON.parse(notifiche)
      }catch{
        tempNotifiche = notifiche
      }

      if (tempNotifiche.titolo == null || tempNotifiche.data == null) {
        res.status(400).json({ error: "Wrong format for notifiche" });
        return;
      }
    }
    if (durata <= 0) {
      res.status(400).json({ error: "Wrong format for durata" });
      return;
    }
    let tempEvento
    if (isEventoSingolo) {
      if (eventoSingolo != null) {
        try{
          tempEvento = JSON.parse(eventoSingolo)
        }catch{
          tempEvento = eventoSingolo
        }
        if (
          tempEvento.data == null ||
          tempEvento.isScadenza == null
        ) {
          res.status(400).json({ error: "Wrong format for eventoSingolo" });
          return;
        }
      }
    } else {
      if (eventoRipetuto != null) {
        try{
          tempEvento = JSON.parse(eventoRipetuto)
        }catch{
          tempEvento = eventoRipetuto
        }
        if (
          tempEvento.numeroRipetizioni == null ||
          tempEvento.impostazioniAvanzate == null ||
          tempEvento.impostazioniAvanzate.giorniSettimana == null ||
          tempEvento.impostazioniAvanzate.data == null ||
          tempEvento.numeroRipetizioni < 1 ||
          tempEvento.impostazioniAvanzate.giorniSettimana == []
        ) {
          res.status(400).json({ error: "Wrong format for eventoRipetuto" });
          return;
```

API - modificaEvento, 1a parte

Immagine PNG API - modificaEvento, 1a parte

```
      res.status(400).json({ error: "Wrong format for eventoRipetuto" });
      return;
    }
  }
}


if (
  IDEvento == null ||
  IDCalendario == null ||
  titolo == null ||
  userId == null ||
  isEventoSingolo == null ||
  (isEventoSingolo == true && eventoSingolo == null) ||
  (isEventoSingolo == false && eventoRipetuto == null) ||
  descrizione == null ||
  luogo == null ||
  partecipanti == null ||
  priorita == null ||
  difficolta == null ||
  notifiche == null ||
  durata == null
) {
  res.status(400).json({ error: "Parameter missing" });
  return;
}

const users = await UtenteAutenticato.find({
  userId: userId,
});
if (Object.keys(users).length == 0) {
  res.status(409).json({ error: "There is no user with that userId" });
  return;
} else if (Object.keys(users).length > 1) {
  res
    .status(409)
    .json({ error: "There are too many users with that userId" });
  return;
}
var ObjectId = require("mongoose").Types.ObjectId;

const evento = await Evento.find({
  _id: new ObjectId(IDEvento),
  partecipanti: userId,
});
if (
  Object.keys(evento).length == 0 ||
  evento[0].partecipanti[0] != userId
) {
  res.status(409).json({
    error: "You do not own the event",
  });
  return;
}

const calendario = await Calendario.find({
  _id: new ObjectId(IDCalendario),
  partecipanti: userId,
});
if (
  Object.keys(calendario).length == 0 ||
  calendario[0].partecipanti[0] != userId
) {
  res.status(409).json({
    error:
      "There is no calendar with that ID or you do not own the calendar",
  });
  return;
}


if (isEventoSingolo) {
  Evento.updateMany(
    { _id: new ObjectId(IDEvento) },
    {
      IDCalendario: IDCalendario,
      titolo: titolo,
      descrizione: descrizione == null ? undefined : descrizione,
      luogo: luogo == null ? undefined : tempLuogo,
      priorita: priorita == null ? undefined : priorita,
      difficolta: difficolta == null ? undefined : difficolta,
      partecipanti:
        partecipanti == null ? calendario[0].partecipanti : partecipanti,
      notifiche: notifiche == null ? undefined : tempNotifiche,
      durata: durata == null ? undefined : durata,
      isEventoSingolo: true,
      eventoSingolo: eventoSingolo == null ? undefined : tempEvento,
    },
    function (err, calendar) {
      if (err) {
        res.status(500).json({ error: "Not inserted" });
        return;
      }
    },
  );
```

API - modificaEvento, 2a parte

Immagine PNG API - modificaEvento, 2a parte

```
    } else {
      Evento.updateMany(
        { _id: new ObjectId(IDEvento) },
        {
          IDCalendario: IDCalendario,
          titolo: titolo,
          descrizione: descrizione == null ? undefined : descrizione,
          luogo: luogo == null ? undefined : tempLuogo,
          priorita: priorita == null ? undefined : priorita,
          difficolta: difficolta == null ? undefined : difficolta,
          partecipanti:
            partecipanti == null ? calendario[0].partecipanti : partecipanti,
          notifiche: notifiche == null ? undefined : tempNotifiche,
          durata: durata == null ? undefined : durata,
          isEventoSingolo: false,
          eventoRipetuto: eventoRipetuto == null ? undefined : tempEvento,
        },
        function (err, calendar) {
          if (err) {
            res.status(500).json({ error: "Not modified" });
            return;
          }
        },
      );
    }

    res.status(200).json({ success: "Event edited correctly" });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - modificaEvento, 3a parte

5.6 : Elimina Evento

Questa API ha lo scopo di andare ad eliminare un "Evento", dato il suo "IDEvento" e l' "userId" dell'utente autenticato che ha questo evento che vuole eliminare. Ovviamente, nel caso in cui non avessimo uno di questi due parametri, come già detto in APD4.2.2, otteniamo un messaggio di errore del tipo "400 - Parameter missing".
In seguito, vengono fatti sempre gli stessi controlli già citati in APD5.3, ma stavolta per gli eventi. Infatti si controlla che l'evento che si vuole eliminare faccia parte della lista degli eventi appartenenti all'utente autenticato indicato dall' "userId". Se così non fosse, l'API invia l'errore "409 - You do not own the event".
Infine, anche per questa funzione vengono usate le funzioni "find()" e "deleteMany()" per ottenere rispettivamente la lista di eventi di un utente autenticato e per eliminare l'evento identificato dall' "IDEvento".

Immagine PNG API - modificaEvento, 3a parte

```
export async function eliminaEvento(req, res) {
  await dbConnect();
  try {
    const { IDEvento } = req.query;
    const { userId } = req.query;

    if (IDEvento == null || userId == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }
    var ObjectId = require("mongoose").Types.ObjectId;

    const eventiPosseduti = await Evento.find({
      $and: [{ partecipanti: userId }, { _id: new ObjectId(IDEvento) }],
    });
    if (
      Object.keys(eventiPosseduti).length == 0 ||
      eventiPosseduti[0].partecipanti[0] != userId
    ) {
      res.status(409).json({
        error: "You do not own the event",
      });
      return;
    }

    const deleteEvent = await Evento.deleteMany({
      _id: new ObjectId(IDEvento),
    });

    if (deleteEvent.deletedCount >= 1) {
      res.status(200).json({ success: "Event deleted correctly" });
      return;
    } else {
      res.status(500).json({ error: "Event not deleted" });
      return;
    }
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - eliminaEvento

Immagine PNG API - eliminaEvento

5.7 : Crea Account

Questa API viene utilizzata per creare un "account", ovvero un oggetto di tipo "UtenteAutenticato", struttura dati già descritta in precendenza, all'interno del database MongoDB, la prima volta che un utente accede autenticandosi nel sito PlanIt. I parametri, che devono essere presenti per non ottenere il messaggio di errore "400 - Parameter missing", sono: "userId", "email" e "username" dell'utente autenticato che si vuole creare.
I controlli che vengono effettuati sono gli stessi già descritti nelle altre API, si evidenzia la presenza della funzione "find()" che ha lo scopo di trovare se esistono nel database già utenti con l' "userId" o l' "email" inseriti in input; se ne esistessero otteniamo come errore "409 - There is already one user with that id or email".
Alla fine della funzione, viene usata la funzione "create()" per andare a creare nel database un oggetto "UtenteAutenticato" con gli attributi ricevuti come parametri.

```
export async function creaUser(req, res) {
  await dbConnect();
  try {
    const { userId, email, username } = req.query;

    if (userId == null || email == null || username == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      $or: [{ userId: userId }, { email: email }],
    });
    if (Object.keys(users).length >= 1) {
      res
        .status(409)
        .json({ error: "There is alrady one user with that id or email" });
      return;
    }

    UtenteAutenticato.create(
      {
        userId: userId,
        email: email,
        username: username,
      },
      function (err, user) {
        if (err) {
          res.status(500).json({ error: "Not inserted" });
          return;
        }
      },
    );
    res.status(200).json({ success: "User inserted correctly" });
    return;
  } catch (e) {
    res.status(501).json({ error: "Generic error" });
    return;
  }
}
```

API - creaAccount

Immagine PNG API - creaAccount

5.8 : Modifica Account

L'applicativo PlanIt sfrutta questa API per andare a modificare l'account di un oggetto "UtenteAutenticato" già presente nel database. Per andare a modificare l'oggetto "UtenteAutenticato" vengono utilizzati i parametri che sono ricevuti in input, che non sono altro che gli attributi che formano un oggetto di tipo "UtenteAutenticato". Per andare ad identificare l'utente da modificare viene usato l' "userId" ricevuto, invece l' "username" è l'attributo che viene modificato nell'oggetto "UtenteAutenticato" già presente nel database. I controlli effettuati, all'interno di questa API, sono gli stessi già descritti; infatti si controlla che l' "userId" inserito corrisponda ad un utente esistente e non duplicato e che la funzione "updateMany()" esegua con nessun tipo di errore.

```
export async function modificaUser(req, res) {
  await dbConnect();
  try {
    const { userId, username } = req.query;

    if (userId == null || username == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }

    const put = await UtenteAutenticato.updateMany(
      { userId: userId },
      { $set: { username: username } },
    );

    if (put.modifiedCount == 1) {
      res.status(200).json({ success: "User updated correctly" });
    } else {
      res.status(500).json({ error: "Not edited" });
    }
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - modificaAccount

Immagine PNG API - modificaAccount

5.9 : Get Dati Account - userId

Questa API viene utilizzata per ottenere in output un oggetto di tipo "UtenteAutenticato" che abbia l' "userId" uguale al parametro "userId" inserito in input. Nel caso in cui l' "userId" non fosse ricevuto in input o non corrispondesse a nessun "UtenteAutenticato" salvato nel database o corrispondesse a più "UtenteAutenticato", riceviamo un messaggio di errore diverso per ciascuno dei casi, già descritti più volte precedentemente.

```
export async function getUser(req, res) {
  await dbConnect();
  try {
    const { userId } = req.query;

    if (userId == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }

    res.status(200).json({
      userId: users[0].userId,
      email: users[0].email,
      username: users[0].username,
    });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - Get Dati Account_userId

Immagine PNG API - Get Dati Account_userId

5.10 : Get Dati Account - email

Questa API viene utilizzata per ottenere in output un oggetto di tipo "UtenteAutenticato" che abbia l' "email" uguale al parametro "email" inserito in input. Questa API fa la stessa procedura e ha lo stesso scopo dell'API descritta in APD5.9, l'unica cosa che cambia è che viene utilizzata l'email per identificare univocamente un "UtenteAutenticato". Con entrambe le APIs, ad ogni modo, si dovrebbe ottenere lo stesso risultato in quanto sia l' "userId" che l' "email" dovrebbero identificare univocamente un "UtenteAutenticato".

```
export async function getUser(req, res) {
  await dbConnect();
  try {
    const { email } = req.query;

    if (
      email == null ||
      !/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/.test(email)
    ) {
      res.status(400).json({ error: "Parameter missing or malformed" });
      return;
    }

    const emailDB = await UtenteAutenticato.find({
      email: email,
    });

    if (Object.keys(emailDB).length == 0) {
      res.status(409).json({ error: "There is no user with that email" });
      return;
    } else if (Object.keys(emailDB).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that email" });
      return;
    }

    res.status(200).json({
      userId: emailDB[0].userId,
      email: emailDB[0].email,
      username: emailDB[0].username,
    });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - Get Dati Account_email

Immagine PNG API - Get Dati Account_email

5.11 : getCalendari

L'applicativo PlanIt usa questa API per ottenere tutti i calendari dell'utente autenticato identificato dall' "userId" ricevuto in input. Dunque l'unico parametro che deve essere presente, per non riceve l'errore "400 - Parameter missing", è l' "userId", su cui si fanno sempre i soliti controlli. Alla fine della procedura, se questa va a buon fine, si ottiene l'array di strutture dati "Calendario" che appartengono all' utente autenticato che ha tale "userId".

```
export async function getCalendari(req, res) {
  await dbConnect();
  try {
    const { userId } = req.query;

    if (userId == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }

    const calendari = await Calendario.find({
      partecipanti: userId,
    });
    if (Object.keys(calendari).length == 0) {
      res
        .status(409)
        .json({ error: "There are no calendars with that userId" });
      return;
    }
    res.status(200).json({
      calendari,
    });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - getCalendari

Immagine PNG API - getCalendari

5.12 : getEventi

Questa API viene usata per ottenere tutti gli eventi che appartengono ad un calendario di un utente autenticato. Questo calendario viene identificato dall' "IDCalendario" e l' "userId" ricevuti in input, i quali, per far portare a buon fine la procedura dell'API, non devono essere vuoti. Sul parametro "userId" vengono fatti i soliti controlli già presenti nelle altre API. Invece, dato l' "IDCalendario" e l' "userId", grazie alla funzione "find()", "getEventi" controlla che tale calendario esista e che appartenga alla lista dei calendari dell'utente autenticato; se così non si fosse, si riceve il messaggio di errore "409 - There is no calendar with that ID or you are not part of it".
Infine, si controlla anche che questo calendario non sia vuoto, ovvero che abbia almeno un evento, in caso lo fosse si ottiene il messaggio di errore "409 - There are no events with that userId and IDCalendario".

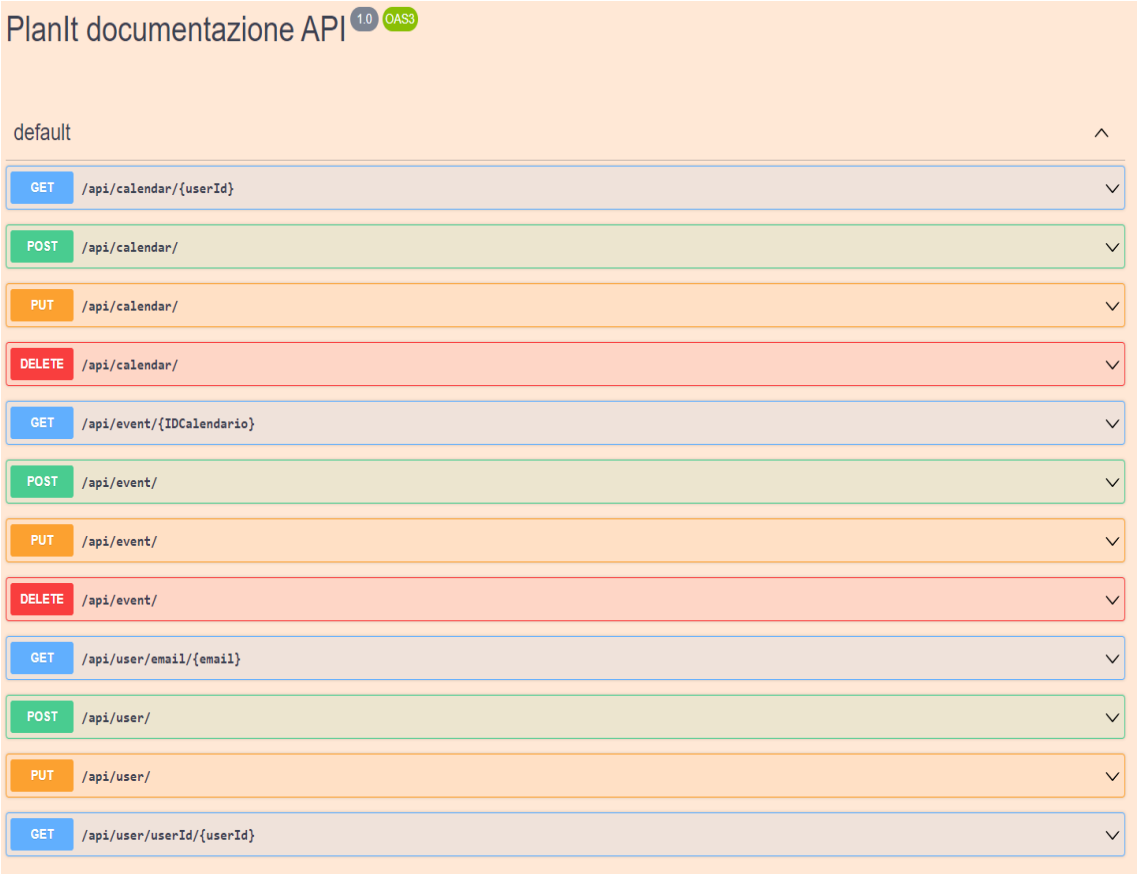
```
export async function getEventi(req, res) {
  await dbConnect();
  try {
    const { IDCalendario } = req.query;
    const { userId } = req.query;

    if (IDCalendario == null || userId == null) {
      res.status(400).json({ error: "Parameter missing" });
      return;
    }

    const users = await UtenteAutenticato.find({
      userId: userId,
    });
    if (Object.keys(users).length == 0) {
      res.status(409).json({ error: "There is no user with that userId" });
      return;
    } else if (Object.keys(users).length > 1) {
      res
        .status(409)
        .json({ error: "There are too many users with that userId" });
      return;
    }
    var ObjectId = require("mongoose").Types.ObjectId;

    const calendariPosseduti = await Calendario.find({
      $and: [{ partecipanti: userId }, { _id: new ObjectId(IDCalendario) }],
    });
    if (Object.keys(calendariPosseduti).length == 0) {
      res.status(409).json({
        error: "There is no calendar with that ID or you are not part of it",
      });
      return;
    }

    const eventi = await Evento.find({
      $and: [
        { partecipanti: userId },
        { IDCalendario: new ObjectId(IDCalendario) },
      ],
    });
    if (Object.keys(eventi).length == 0) {
      res.status(409).json({
        error: "There are no events with that userId and IDCalendario",
      });
      return;
    }
    res.status(200).json({
      eventi,
    });
    return;
  } catch (e) {
    console.error(e);
    res.status(501).json({ error: "Generic error" });
    throw new Error(e).message;
  }
}
```

API - getEventi

Immagine PNG API - getEventi

3 API documentation

Le API fornite dall'applicazione PlanIt e già descritte nella sezione precedente APD5 sono state documentate utilizzando il modulo di NextJS chiamato "next-swagger-doc". In questo modo la documentazione relativa alle API è direttamente disponibile a chiunque veda il codice sorgente.
Per poter generare l'endpoint dedicato alla presentazione delle API abbiamo utilizzato swagger-ui-react in quanto crea una pagina web secondo la documentazione scritta. In particolare, di seguito mostriamo la pagina web relativa alla documentazione che presenta le 12 API, ce ne sono sia di tipo GET che POST che DELETE, essenziali per la gestione del prototipo del sito PlanIt che è stato sviluppato.
Nella pagina della documentazione fornita, abbiamo reso disponibile anche la possibilità di utilizzare degli esempi preparati "ad hoc" per poter vedere come funzionano tutte le API da noi implementate.
L'endpoint da invocare per raggiungere la seguente documentazione e': http://localhost:3000/ApiDoc oppure http://127.0.0.1:3000/ApiDoc Inoltre, la documentazione è possibile visionarla non solo, grazie al codice sorgente da noi sviluppato, ma anche direttamente al link: https://api.plan-it.it/apidoc. Infatti abbiamo usato il sito "netlify" per l'hosting di ciò che abbiamo implementato per questo deliverable D4, in modo tale che la documentazione e anche il FrontEnd, il cui link verrà passato successivamente, possano essere sempre raggiungibili anche senza dover avere per forza il codice sorgente da eseguire.



Documentazione

Immagine PNG Documentazione

4 FrontEnd Implementation

5 GitHub Repository and Deployment Info

Riguardo alla descrizione del repo Git, che nel nostro caso si chiama "Codice" (repo dove è presente tutto il codice da noi sviluppato per l'implementazione del sito PlanIt, sia BackEnd che FrontEnd), molto è già descritto in APD1. Ma in questa capitolo andiamo a descrivere altre cartelle presenti nel nostro git "Codice" (link: https://github.com/Life-planner/Codice). Infatti in questo cartella git è presente anche la cartella "_tests_" dove sono presenti tutti i test effettuati per ciascuna API che è stata implementata, divisi a seconda di quale "resource" (guardare Extract Diagram APD4.1) riguardano. Inoltre, in ciascuno file della cartella api, è presente la documentazione relativa alle funzioni presenti in tale file. E' stato fatto in questo modo, ovvero la presenza della documentazione nel relativo file, in quando abbiamo usato una libreria apposita di Swagger per NextJS che richiede tale procedura per andare a formare la documentazione. Infatti, grazie "next-swagger-doc", basta fornire la documentazione in ciascun file e automaticamente viene compilato un file "swagger.json" dove è scritta tutta la documentazione in formato ".json". Questo file è contenuto nella cartella "public", dove sono presenti anche tutti gli "asset" del progetto.
In questo capitolo presentiamo anche le informazioni riguardo al deployment e al link per eseguire il prototipo PlanIt da noi implementato. Per l'hosting del nostro sito abbiamo deciso di utilizzare il sito di hosting netlify.
Questi sono i link da noi fatti da cui si può usufruire di ciò che abbiamo implementato:

- main branch: https://plan-it.it;
- api branch: https://api.plan-it.it;
- frontend branch: https://frontend.plan-it.it
- documentazione: https://api.plan-it.it/apidoc;

6 Testing

7 Legenda Riferimenti

riferimenti

Diagramma delle classi	DCL
Object Constraint Language	OCL