



Ingegneria del Software - DISI - Anno Accademico 2023-2024

Daniele Ye - Alessandro Dalfovo. - Giovanni Panighel

Nome progetto:

Tripply

Gruppo G42 - Deliverable 4

Scopo del documento	4
User Flows	5
Utente non autenticato	5
Utente autenticato	6
Application Implementation and Document	8
Project Structure	9
- file: .gitignore	10
- directory: package.json	10
- directory: README.md	10
- directory: server/db	10
- directory: server/managers	10
- directory: server/routers	10
- directory: server/views	10
- file: server/.env	10
- file: server/.loadEnvironment.js	10
- file: server/app.js	10
- file: server/server.js	10
Project Dependencies	11
Project Database	12
- Collezione Utente	12
- Collezione Itinerario	13
- Collezione Salvati	13
Project APIs	14
Resources Extraction from Class Diagram	14
Resources Models	16
- User	16
- Saved	16
- Itinerary	16
Sviluppo API	19
.Login	19
.Logout	19

.deleteUser	20
.getUser	20
.getSavedItineraries	22
.isSavedItinerary	23
.addItineraryToSaved	24
.deleteSavedList	25
.removeItineraryFromSaved	26
Frontend Implementation	44
1. Pagina Home	44
2. Pagina Login	45
3. Pagina Registrazione	45
4. Pagina Recupero password	46
5. Pagina Home/Itinerary community	47
a. Pagina Home/Itinerari community itinerario esteso	47
b. Pagina Home/Itinerari community recensioni	48
6. Pagina i miei itinerari	49
7. Pagina itinerari salvati	49
8. Pagina crea	50
9. Pagina ricerca	50
10. Pagina Profilo	51
GitHub Repository and Deployment Info	52
Testing	53
1. Test /api/getUserItineraries	55
2. Test /api/calcTimeItinerary	55
3. Test /api/reviewItinerary	55
4. Test /api/getItineraryReview	55
5. Test /api/addDay	56
6. Test /api/containsDay	56
7. Test /api/searchItineraries	56
8. Test /api/createItinerary	57
9. Test /api/getCommunityItineraries	58
10. Test /api/deleteItinerary	60

11. Test /api/getSavedItineraries	60
12. Test /api/addItineraryToSaved	60
13. Test /api/isSavedItinerary	60
14. Test /api/removeItineraryFromSaved	60
15. Test /api/deleteSavedList	61
16. Test /api/addStop	61
17. Test /api/deleteStop	61
18. Test /api/replaceStop	61
19. Test /api/calcDistance	61
20. Test /api/calcPath	61

Scopo del documento

Il documento fornisce dettagli fondamentali per implementare una parte dell'applicazione Tripply, focalizzandosi sulla creazione dei servizi per la gestione degli itinerari e dell'accesso e delle identità degli utenti.

Inizia con la descrizione dei flussi utente per utenti autenticati e non autenticati, procedendo poi con la presentazione delle API necessarie utilizzando il Modello API e il modello delle risorse. Vengono anche illustrati lo sviluppo delle API, descritte in linguaggio naturale.

Il documento include la documentazione e i test eseguiti con Postman.

User Flows

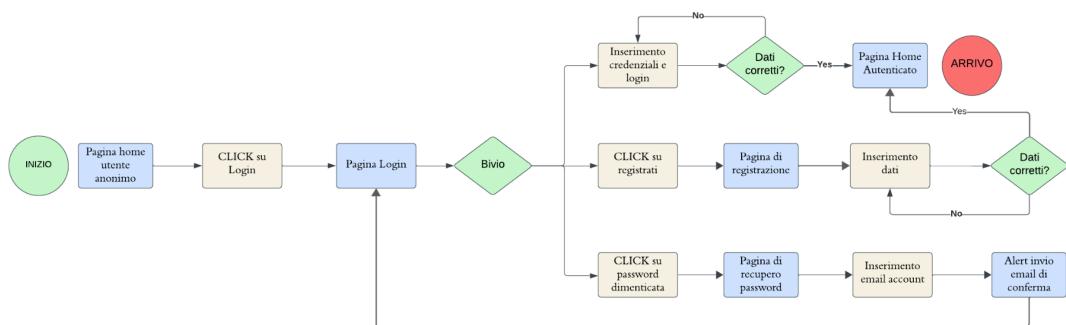
Nel seguente capitolo vengono riportati gli Users Flows che hanno lo scopo di rappresentare come un utente interagirà con il sito web.

Sono stati individuati due tipi di utenti: Utente non autenticato e utente autenticato.

Utente non autenticato

L'User Flow dell'utente anonimo è più limitato rispetto a quello dell'utente autenticato. L'utente non autenticato può visitare la pagina home e poi scegliere di effettuare il login, il sign-in o il recupero password.

Per maggiore chiarezza nello schema, non sono stati aggiunti l'accettazione dei cookies, che l'utente può fare in qualsiasi momento, nell'User Flow.



link all'immagine ad alta risoluzione [qui](#)

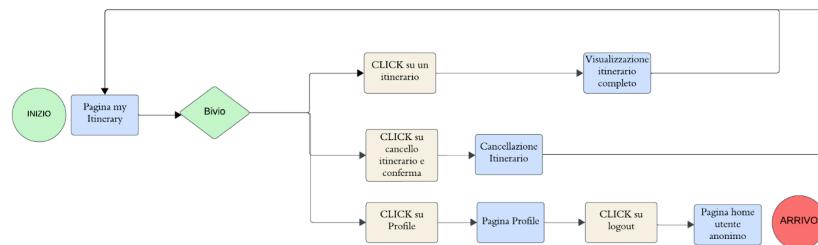
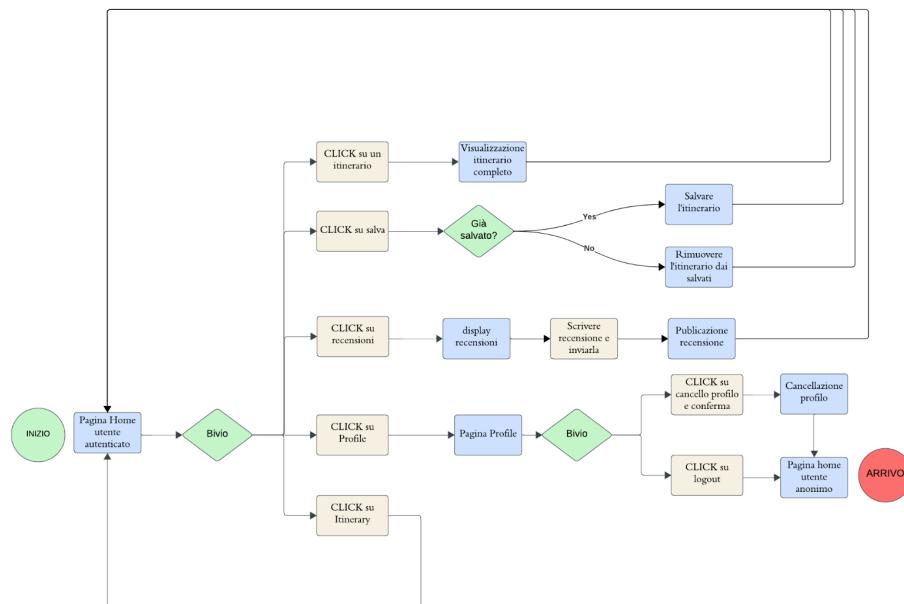
Utente autenticato

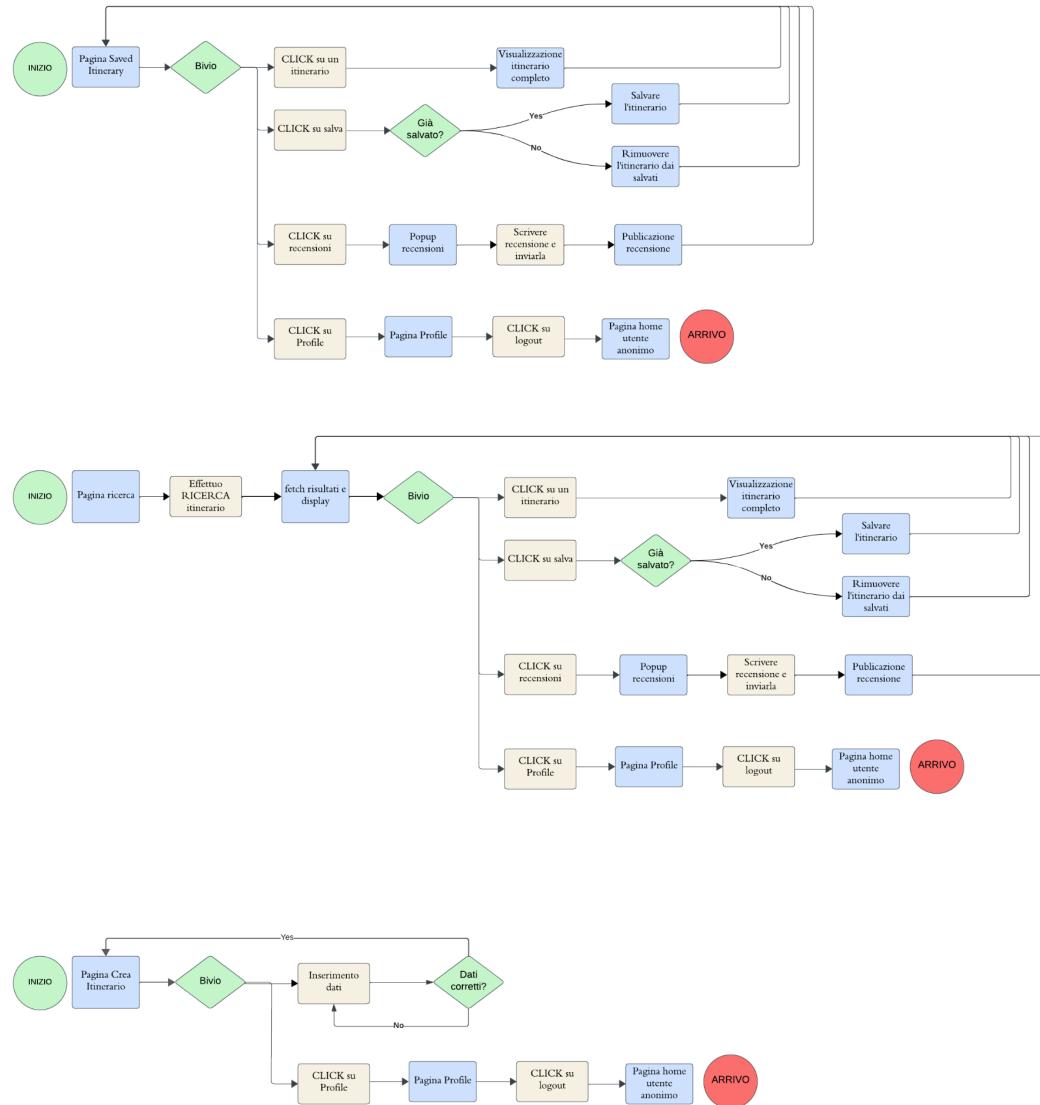
L'utente autenticato ha a disposizione in ogni pagina la possibilità di accedere alla Home, accettare i cookies, effettuare ricerche, accedere al proprio profilo e ai propri itinerari, nonché agli itinerari salvati o alla creazione di nuovi itinerari attraverso la barra di navigazione.

Per chiarezza, l'User Flow sarà diviso per le pagine Home, My Itinerary, Saved Itinerary e Ricerca, senza includere le interazioni possibili con la barra di navigazione (ad eccezione dell'accesso alla pagina del Profilo). Ogni azione che richiede una modifica o accesso a dati presenti nel Database richiede la verifica dell'account, ma questo controllo non è stato inserito nel flusso.

Ogni User Flow terminerà con il logout nella pagina del Profilo, che sarà descritta al completo solo nell'User Flow relativo alla Home.

L'utente autenticato ha la possibilità di visualizzare gli itinerari della community, salvarli e recensirli. Effettuare una ricerca oppure creare o eliminare i propri itinerari, e infine eseguire il logout o cancellare l'account.





link all'immagine ad alta risoluzione [qui](#)

Application Implementation and Document

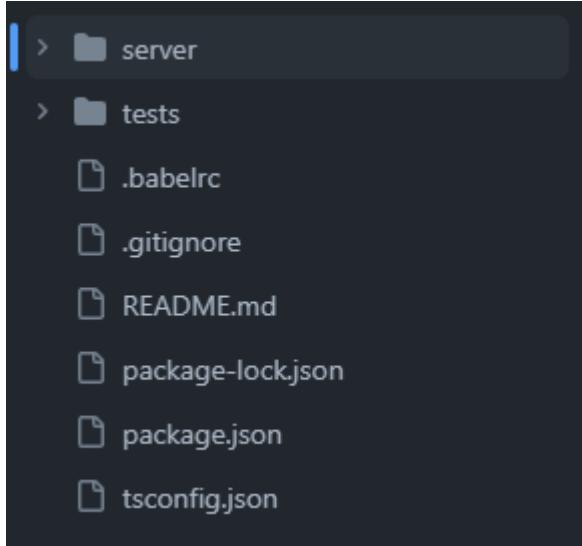
Nel seguente capitolo verranno presentate le varie funzionalità che verranno implementate nell'applicazione Tripply.

Tripply è stato sviluppato utilizzando [Express.js](#) versione 4.18.2, un framework JavaScript leggero, flessibile e open source per lo sviluppo di applicazioni web e REST API.

Project Structure

Utilizzato il software di controllo [Git](#) e come repository remota [Github](#). La repository dell'applicazione [CodeBase](#) appartiene all'organizzazione [GruppoG42](#).

Il sito si presenta nel seguente modo:



La struttura si presenta con 4 file meta:

- .gitignore
- README.md
- package-lock.json
- package.json

e 3 file autogenerati:

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

e la cartella principale dell'applicazione server e la cartella contenente i test.

- **file: .gitignore**

File usato per elencare i file da escludere dall'essere caricati sulla repository.

- **directory: package.json**

File gestito da Node.js server per elencare e salvare le dipendenze.

- **directory: README.md**

File utilizzato per introdurre al progetto fornendo una panoramica del progetto.

- **directory: server/db**

Questa cartella contiene moduli js dedicati alla gestione della connessione al database.

- **directory: server/managers**

In questa cartella risiedono i files js che esportano moduli fondamentali per il funzionamento dell'applicazione.

- **directory: server/routers**

Questa cartella contiene i file che definiscono le rotte e le API.

- **directory: server/views**

In questa cartella sono collocate tutte le pagine .ejs che costituiscono il front-end.

- **file: server/.env**

Contiene le variabili di ambiente specifiche del server e importantissime per configurare l'ambiente di esecuzione del server.

- **file: server/.loadEnvironment.js**

Carica le variabili d'ambiente

- **file: server/app.js**

Script che imposta i middleware e le route del sistema

- **file: server/server.js**

Entry point dell'applicazione che avvia il server

Project Dependencies

Il progetto sfrutta una serie di dipendenze principalmente:

- auth0, integrazione dell'autenticazione e dell'autorizzazione di Auth0
- express-openid-connect, per l'autenticazione OpenID connect
- mongodb, per la connessione al database
- swagger-ui-express per l'integrazione della documentazione Swagger generata da Swagger Core
- dotenv per il caricamento di variabili d'ambiente

Project Database

Il database scelto per memorizzare i dati a lungo termine è MongoDB gestito su Atlas. I dati presentati nei precedenti documenti sono stati organizzati in 3 collezioni: Utente, Itinerario e Salvati.

- Collezione Utente

In questa collezione vengono memorizzati i dati relativi agli utenti. Questi dati possono variare a seconda se l'utente si è registrato inserendo manualmente i propri dati o tramite un'autenticazione tramite social network. Essi includono informazioni di profilo e altre informazioni pertinenti all'utente.

Utente normale:

```

1 {-
2   "sub": "auth0|TestID",
3   "sid": "testSid",
4   "nickname": "test",
5   "name": "test@test.com",
6   "picture": "https://s.gravatar.com/avatar/test.png",
7   "updated_at": "2024-02-04T11:20:02.076Z",
8   "email": "test@test.com",
9   "email_verified": false,
10 }

```

Utente sociale:

```

1 {-
2   "sid": "testSID",
3   "given_name": "test",
4   "family_name": "test",
5   "nickname": "test.test",
6   "name": "test test",
7   "picture": "https://lh3.googleusercontent.com/a/test",
8   "locale": "it",
9   "updated_at": "2024-02-04T16:51:46.959Z",
10  "email": "test.test@studenti.unitn.it",
11  "email_verified": true,
12  "sub": "google-oauth2|test"
13 }
14

```

- Collezione Itinerario

Vengono memorizzati tutti gli itinerari dell'applicazione.

```
1 {  
2   "_id": {  
3     "$oid": "65bea7fcb410532d3c471d35"  
4   },  
5   "nome": "Itinerario di esempio",  
6   "stato": "Italia",  
7   "giorni": [  
8     {  
9       "descrizione": "Giorno 1",  
10      "tappe": [  
11        {  
12          "descrizione": "Tappa 1",  
13          "luogo": "Roma"  
14        },  
15        {  
16          "descrizione": "Tappa 2",  
17          "luogo": "Firenze"  
18        }  
19      ]  
20    },  
21  ],  
22  "recensioni": [],  
23  "descrizione": "Itinerario di viaggio in Italia",  
24  "attivo": true,  
25  "_userId": "google-oauth2|test"  
26 }  
27
```

- Collezione Salvati

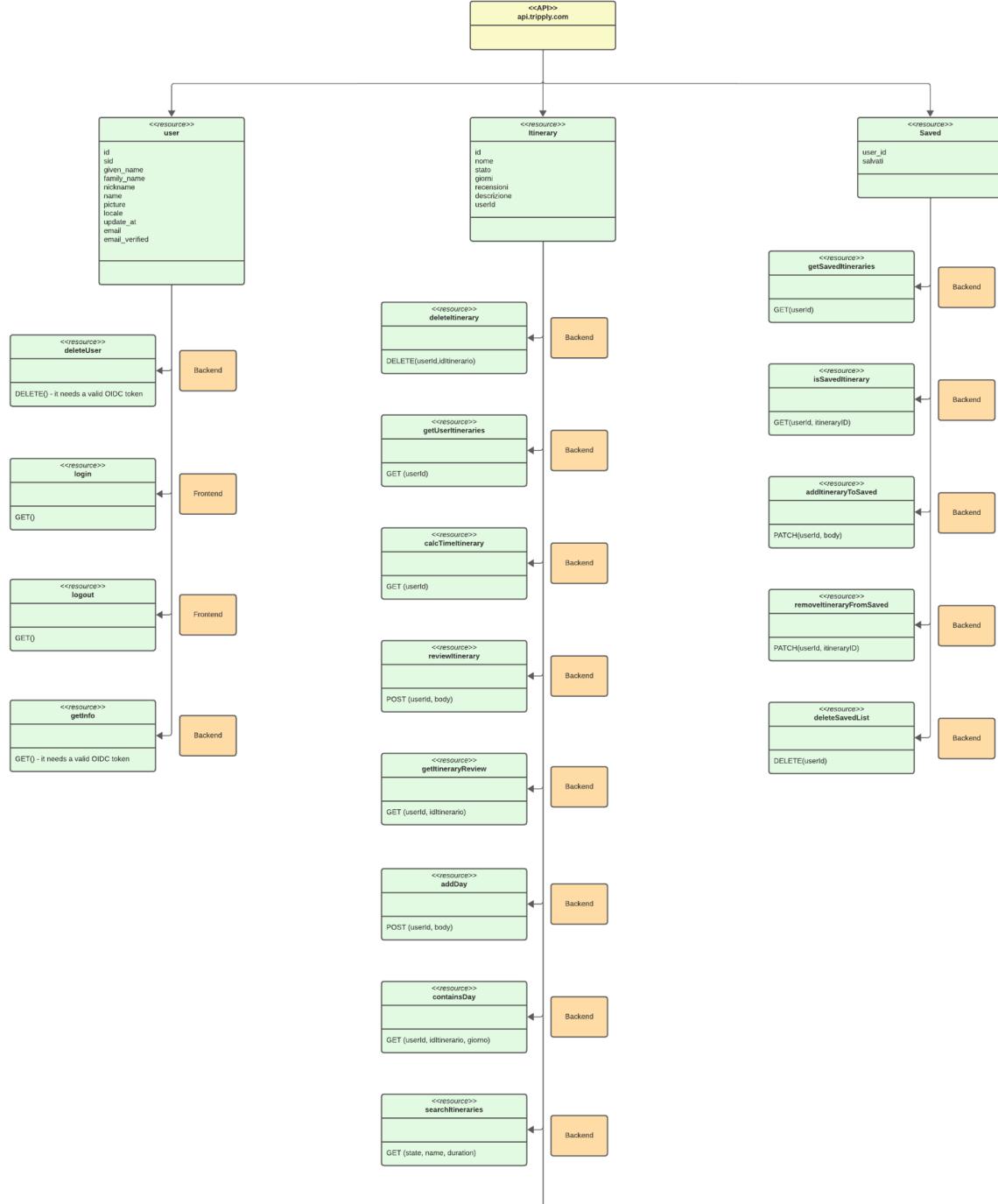
Serve per memorizzare gli itinerari salvati dall'utente.

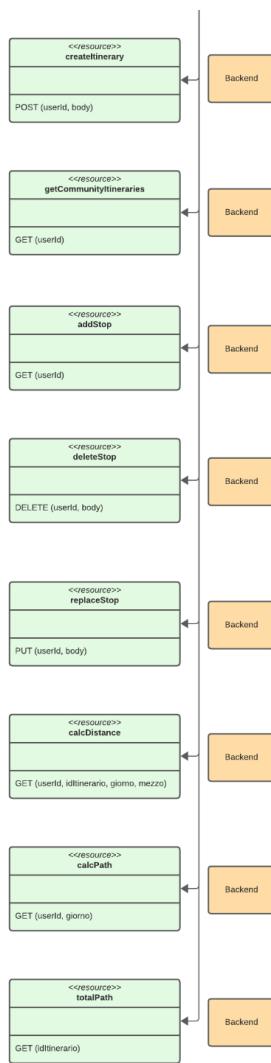
```
1 {  
2   "_id": "google-oauth2|test",  
3   "salvati": [  
4     "testItineraryId"  
5   ]  
6 }
```

Project APIs

Resources Extraction from Class Diagram

Tramite il class diagram identifichiamo le risorse per valutare quali API implementare nel sito.



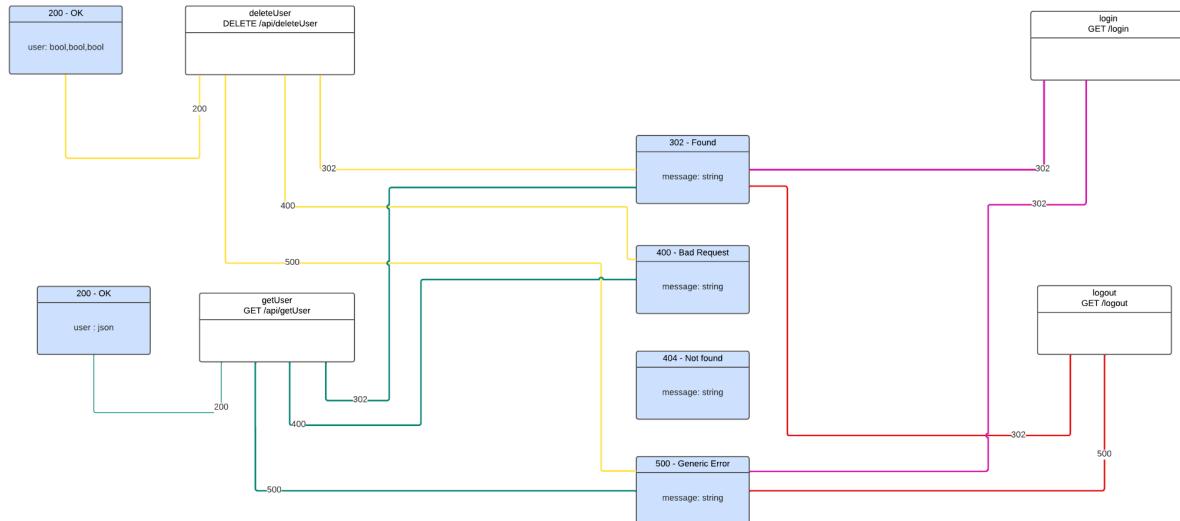


link all'immagine ad alta risoluzione [qui](#)

Resources Models

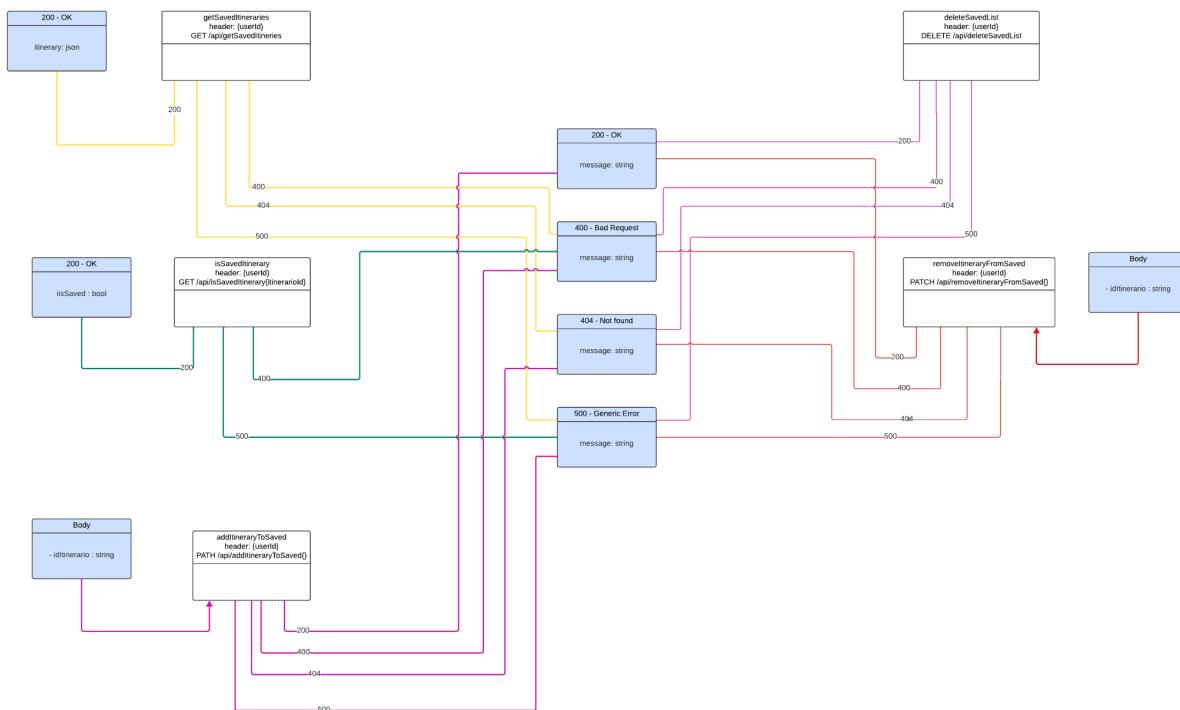
Il Resources Model astrae e rappresenta graficamente le API necessarie per il funzionamento del sito. Mostrando per ogni API, il nome, il metodo e il path.

- User



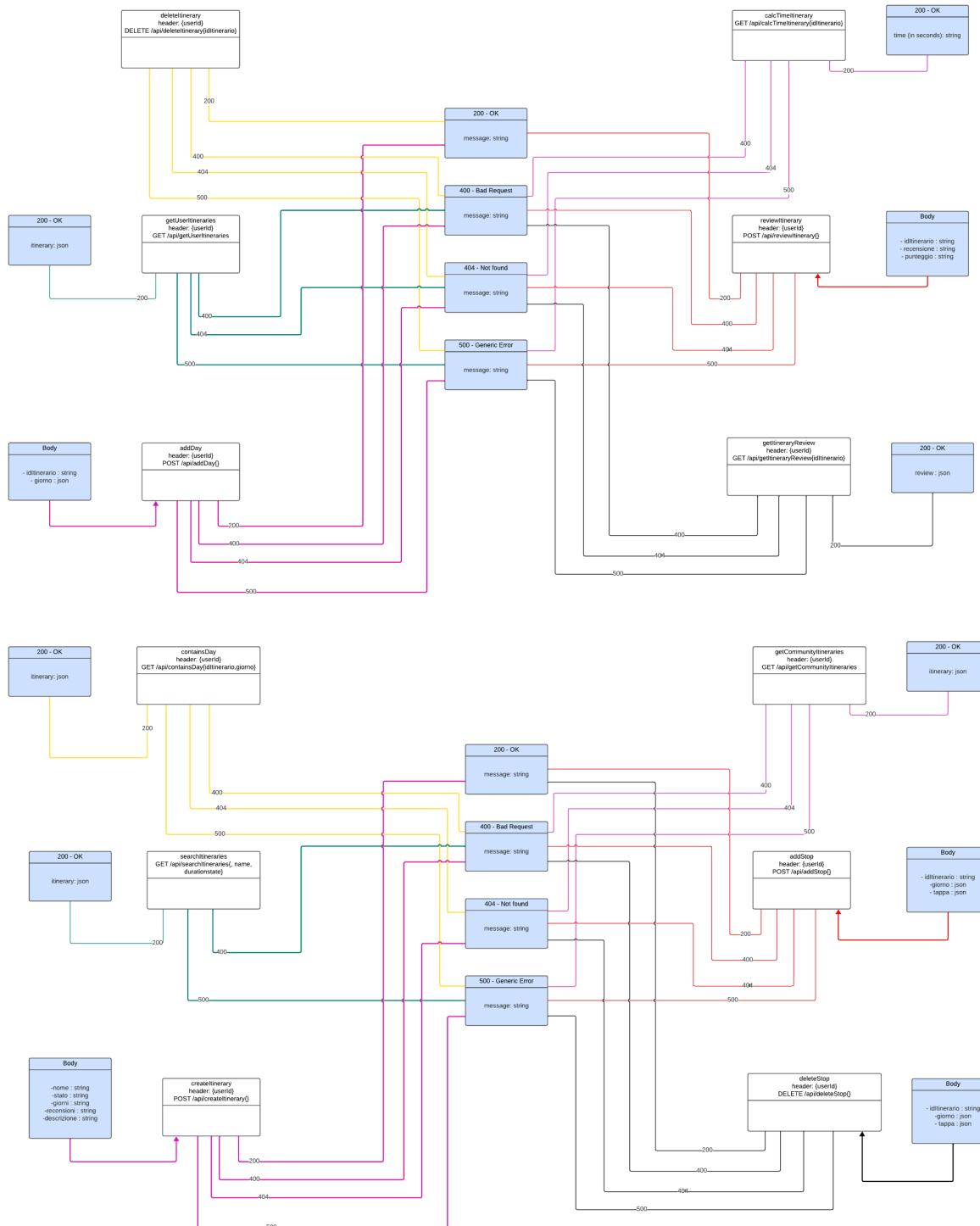
link all'immagine ad alta risoluzione [qui](#)

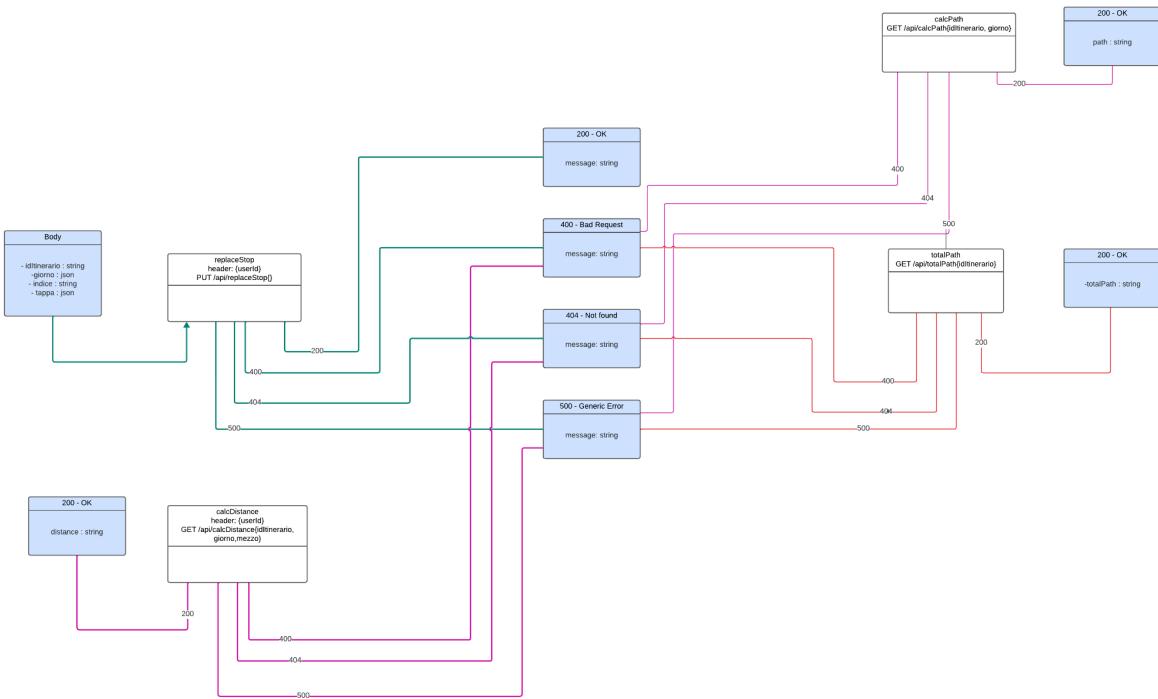
- Saved



link all'immagine ad alta risoluzione [qui](#)

- Itinerary





link all'immagine ad alta risoluzione [qui](#)

Sviluppo API

.Login

API che viene restituisce la pagina login che ridireziona verso la pagina login di auth0.

```
router.get('/login', function (req, res) {  
    res.render('login', {  
        title: 'Login'  
    });  
});
```

.Logout

req.logout() è una funzione fornita da express-openid-connect e l'API ritorna codice 302.

```
router.get('/logout', function (req, res) {  
    req.logout();  
    res.redirect('/');  
});
```

.deleteUser

Questa route gestisce una richiesta di tipo DELETE all'endpoint '/deleteUser'. Prima di procedere, richiede l'autenticazione tramite la funzione `requiresAuth()` fornita da express-openid-connect. Se l'utente non è autenticato, risponde con un codice 302, reindirizzando l'utente alla pagina di login. L'ID dell'utente viene recuperato dalla sessione dell'utente autenticato.

Successivamente, vengono eseguite tre funzioni asincrone per cancellare i dati dell'utente:

1. La funzione `userManager.deleteUser(userId)` elimina l'utente dal sistema.
2. La funzione `itinerarioManager.deleteItineraries(userId)` cancella tutti gli itinerari associati all'utente.
3. La funzione `tripplerManager.deleteSaved(userId)` elimina la cronologia degli itinerari salvati dell'utente

Se tutte queste operazioni vanno a buon fine, viene restituita una risposta JSON con lo stato delle eliminazioni eseguite. In caso di errore generico durante l'eliminazione, viene restituito uno stato 500 (Internal Server Error) con un messaggio di errore.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.delete('/deleteUser', requiresAuth(), async (req, res) => {
  try {
    const userId = req.oidc.user.sub;
    const deleteUser = await userManager.deleteUser(userId);
    const deleteItineraries = await itinerarioManager.deleteItineraries(userId);
    const deleteSaved = await tripplerManager.deleteSaved(userId);
    res.json({user: deleteUser, deleteItineraries, deleteSaved});
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

.getUser

Questa route gestisce una richiesta GET all'endpoint '/getUser'. Prima di procedere, richiede l'autenticazione tramite la funzione `requiresAuth()` fornita da `express-openid-connect`. Se l'utente non è autenticato, risponde con un codice 302, reindirizzando l'utente alla pagina di login. L'ID dell'utente viene quindi recuperato dalla sessione dell'utente autenticato.

Successivamente, viene eseguita una funzione asincrona per recuperare i dati dell'utente utilizzando `userManager.getUser(userId)`. Se l'operazione va a buon fine, viene restituita una risposta JSON contenente i dati dell'utente. In caso di errore durante il recupero dei dati dell'utente, viene restituito uno stato 500 (Internal Server Error) con un messaggio di errore.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.get('/getUser', requiresAuth(), async (req, res) => {
  try {
    const userId = req_oidc.user.sub;
    const user = await userManager.getUser(userId);
    res.json(user);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

.getSavedItineraries

Questa route gestisce una richiesta GET all'endpoint '/getSavedItineraries'. Ecco come funziona:

1. La route recupera l'ID dell'utente dalla header della richiesta.
2. Viene verificato se l'ID dell'utente è stato fornito. Se non è presente, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente esiste utilizzando una funzione checkUser(req).
4. Se l'utente non esiste, viene restituito uno stato 404 (Not Found) con un messaggio che indica che l'utente non è stato trovato.
5. Se l'utente esiste, vengono recuperati gli itinerari salvati associati a quell'utente utilizzando tripplerManager.getSavedItinerariesById(userId).
6. Gli itinerari recuperati vengono restituiti come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il recupero degli itinerari salvati, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.get('/getSavedItineraries', async (req, res) => {
  try {
    const userId = req.header('userId');
    if (!userId) {
      res.status(400).send('Bad Request: userId is required');
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    const itineraries = await tripplerManager.getSavedItinerariesById(userId);
    res.status(200).json(itineraries);
  } catch (error) {
    console.error(`Error fetching saved itineraries by id: ${error}`);
    res.status(500).send('Internal Server Error');
  }
});
```

.isSavedItinerary

Questa route gestisce una richiesta GET all'endpoint '/isSavedItinerary'.

1. Recupera l'ID dell'utente dall'header della richiesta e l'ID dell'itinerario dalla query string.
2. Viene verificato se entrambi l'ID dell'utente e l'ID dell'itinerario sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente esiste e anche l'itinerario utilizzando le funzione checkUser(req) e checkItinerary(itineraryId).
4. Se l'utente non esiste o l'itinerario, viene restituito uno stato 404 (Not Found).
5. Viene chiamata la funzione tripplerManager.isSavedItinerary(userId, itineraryId) per verificare se l'itinerario è stato salvato dall'utente.

Il risultato della verifica viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante la verifica se l'itinerario è stato salvato, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

```
● ● ●

router.get('/isSavedItinerary', async (req, res) => {
  try {
    const userId = req.header('userId');
    const itineraryId = req.query.itineraryId;
    if (!userId || !itineraryId) {
      res.status(400).send('Bad Request: userId and itineraryId are required');
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    if (!(await checkItinerary(itineraryId))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    const isSaved = await tripplerManager.isSavedItinerary(userId, itineraryId);
    res.status(200).json(isSaved);
  } catch (error) {
    console.error(`Error checking if itinerary is saved: ${error}`);
    res.status(500).send('Internal Server Error');
  }
});
```

.addItineraryToSaved

Questa route gestisce una richiesta PATCH all'endpoint '/addItineraryToSaved'.

1. Recupera l'ID dell'utente dall'header della richiesta e l'ID dell'itinerario dalla query string.
2. Viene verificato se entrambi l'ID dell'utente e l'ID dell'itinerario sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni `checkUser(req)` e `checkItinerary(itineraryId)`.
4. Se non esistono, viene restituito uno stato 404 (Not Found) con un messaggio che indica che l'utente non è stato trovato.
5. Viene chiamata la funzione `'tripplerManager.addItineraryToSaved(userId, itineraryId)'` per aggiungere l'itinerario ai salvati dell'utente.
6. Il risultato dell'operazione viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante l'aggiunta dell'itinerario ai salvati, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.patch('/addItineraryToSaved', async (req, res) => {
    try {
        const userId = req.header('userId');
        const itineraryId = req.body.itineraryId;
        if (!userId || !itineraryId) {
            res.status(400).send('Bad Request: userId and itineraryId are required');
            return;
        }
        if (!(await checkUser(req))) {
            res.status(404).json({error: 'User not found'});
            return;
        }
        if (await trippleManager.isSavedItinerary(userId, itineraryId)) {
            res.status(400).send('Itinerary already saved');
            return;
        }
        const result = await trippleManager.addItineraryToSaved(userId, itineraryId);
        res.status(200).json(result);
    } catch (error) {
        console.error(`Error adding itinerary to saved: ${error}`);
        res.status(500).send('Internal Server Error');
    }
});
```

.deleteSavedList

Questa route gestisce una richiesta DELETE all'endpoint '/deleteSavedList'.

1. Recupera l'ID dell'utente dall'header della richiesta.
2. Verifica se l'ID dell'utente è stato fornito. Se manca, viene restituito uno stato 400 (Bad Request).
3. Verifica se l'utente esiste utilizzando la funzione `checkUser(req)`. Se l'utente non esiste, viene restituito uno stato 404 (Not Found).
4. Chiama la funzione `tripplerManager.deleteSaved(userId)` per eliminare la lista di itinerari salvati dell'utente.
5. Restituisce il risultato dell'operazione come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante l'eliminazione della lista salvata, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.delete('/deleteSavedList', async (req, res) => {
  try {
    const userId = req.header('userId');
    if (!userId) {
      res.status(400).send('Bad Request: userId is required');
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    const result = await tripplerManager.deleteSaved(userId);
    res.status(200).json(result);
  } catch (error) {
    console.error(`Error deleting saved list: ${error}`);
    res.status(500).send('Internal Server Error');
  }
});
```

.removeItineraryFromSaved

Questa route gestisce una richiesta PATCH all'endpoint '/removeItineraryFromSaved'.

1. Recupera l'ID dell'utente dall'header e l'ID dell'itinerario dal corpo della richiesta
2. Viene verificato se entrambi l'ID dell'utente e l'ID dell'itinerario sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni checkUser(req) e checkItinerary(itineraryId).
4. Se non esistono, viene restituito uno stato 404 (Not Found) con un messaggio che indica che l'utente non è stato trovato.
5. Chiama la funzione `tripplerManager.removeItineraryFromSaved(userId, itineraryId)` per rimuovere l'itinerario dai salvati dell'utente.
6. Restituisce il risultato dell'operazione come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante la rimozione dell'itinerario dai salvati, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.patch('/removeItineraryFromSaved', async (req, res) => {
  try {
    const userId = req.header('userId');
    const itineraryId = req.body.itineraryId;
    if (!userId || !itineraryId) {
      res.status(400).send('Bad Request: userId and itineraryId are required');
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    if (!(await tripplerManager.isSavedItinerary(userId, itineraryId))) {
      console.log(userId, itineraryId)
      res.status(400).send('Itinerary not saved');
      return;
    }
    const result = await tripplerManager.removeItineraryFromSaved(userId, itineraryId);
    res.status(200).json(result);
  } catch (error) {
    console.error(`Error removing itinerary from saved: ${error}`);
    res.status(500).send('Internal Server Error');
  }
});
```

.deleteItinerary

Questa route gestisce una richiesta DELETE all'endpoint '/deleteItinerary'.

1. Recupera l'ID dell'utente dall'header della richiesta e l'ID dell'itinerario dalla query string.
2. Viene verificato se entrambi l'ID dell'utente e l'ID dell'itinerario sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni `checkUser(req)` e `checkItinerary(itineraryId)`.
4. Se non esistono, viene restituito uno stato 404 (Not Found) con un messaggio che indica che l'utente non è stato trovato.
5. Chiama la funzione `'tripplerManager.eliminaItinerario(idItinerario)'` per eliminare l'itinerario.
6. Restituisce il risultato dell'operazione come risposta JSON.

Se si verifica un errore durante l'eliminazione dell'itinerario, viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.delete('/deleteItinerary', async (req, res) => {
  try {
    const userId = req.header('userId');
    const idItinerario = req.query.idItinerario;
    if (!userId || !idItinerario) {
      res.status(400).send('Bad Request: userId and idItinerario are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    const eliminaItinerario = await tripplerManager.eliminaItinerario(idItinerario);
    res.json(eliminaItinerario);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

getUserItineraries

Questa route gestisce una richiesta GET all'endpoint '/getUserItineraries'.

1. Recupera l'ID dell'utente dall'header della richiesta.
2. Verifica se l'ID dell'utente è stato fornito. Se manca, viene restituito uno stato 400 (Bad Request).
3. Verifica se l'utente esiste utilizzando la funzione `checkUser(req)`. Se l'utente non esiste, viene restituito uno stato 404 (Not Found).
4. Chiama la funzione `tripplerManager.getUserItineraries(userId)` per ottenere gli itinerari dell'utente.
5. Restituisce gli itinerari dell'utente come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il recupero degli itinerari dell'utente, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.get('/getUserItineraries', async (req, res) => {
  try {
    const userId = req.header('userId');
    if (!userId) {
      res.status(400).send('Bad Request: userId are required');
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    const userItineraries = await tripplerManager.getUserItineraries(userId);

    res.status(200).json(userItineraries);
  } catch (error) {
    console.error('Error fetching user itineraries:', error);
    res.status(500).send('Internal Server Error');
  }
});
```

getUserItineraries

Questa route gestisce una richiesta POST all'endpoint '/addDay'.

1. Recupera l'ID dell'itinerario e il giorno dalla richiesta.
2. Verifica se entrambi l'ID dell'itinerario e il giorno sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Verifica se l'itinerario esiste utilizzando la funzione `checkItinerary(idItinerario)`. Se l'itinerario non esiste, viene restituito uno stato 404 (Not Found).
4. Chiama la funzione `itinerarioManager.aggiungiGiorno(idItinerario, giorno)` per aggiungere un giorno all'itinerario.
5. Restituisce il risultato dell'operazione come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante l'aggiunta del giorno, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.post('/addDay', async (req, res) => {
  try {
    const idItinerario = req.body.idItinerario;
    const giorno = req.body.giorno;
    if (!idItinerario || !giorno) {
      res.status(400).send('Bad Request: idItinerario and giorno are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }

    const aggiungiGiorno = await itinerarioManager.aggiungiGiorno(idItinerario, giorno);
    res.status(200).json(aggiungiGiorno);
  } catch (error) {
    console.error('Error adding day:', error);
    res.status(500).send('Internal Server Error: ' + error);
  }
});
```

.calcTimeItinerary

Questa route gestisce una richiesta GET all'endpoint '/calcTimeItinerary'.

1. Recupera l'ID dell'itinerario dalla query string della richiesta.
2. Verifica se l'ID dell'itinerario è stato fornito. Se manca, viene restituito uno stato 400 (Bad Request).
3. Verifica se l'itinerario esiste utilizzando la funzione `checkItinerary(idItinerario)`. Se l'itinerario non esiste, viene restituito uno stato 404 (Not Found).
4. Chiama la funzione `giornoManager.calcolaTempoPercorrenza(idItinerario)` per calcolare il tempo di percorrenza dell'itinerario.
5. Restituisce il tempo di percorrenza come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il calcolo del tempo di percorrenza, viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.get('/calcTimeItinerary', async (req, res) => {
  try {
    const idItinerario = req.query.idItinerario;
    if (!idItinerario) {
      res.status(400).send('Bad Request: idItinerario are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    const time = await giornoManager.calcolaTempoPercorrenza(idItinerario);
    res.status(200).json(time);
  } catch (error) {
    res.status(500).send('Internal Server Error ' + error);
  }
});
```

.reviewItinerary

Questa route gestisce una richiesta POST all'endpoint '/reviewItinerary'.

1. Recupera l'ID dell'utente, l'ID dell'itinerario, la recensione e il punteggio dalla richiesta.
2. Verifica se tutti i campi necessari (userId, idItinerario, recensione e punteggio) sono stati forniti. Se uno dei campi è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni checkUser(req) e checkItinerary(itineraryId).
4. Chiama la funzione `itinerarioManager.saveReview(idItinerario, userId, recensione, punteggio)` per salvare la recensione dell'utente sull'itinerario.
5. Restituisce il risultato dell'operazione come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il salvataggio della recensione, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.post('/reviewItinerary', async (req, res) => {
  try {
    const userId = req.header('userId');
    const idItinerario = req.body.idItinerario;
    const recensione = req.body.reviewText;
    const punteggio = req.body.rating;
    if (!userId || !idItinerario || !recensione || !punteggio) {
      res.status(400).send('Bad Request: userId, idItinerario, recensione and punteggio are required');
      return;
    }
    if (!(await checkUser(req))) {
      console.error('User not found')
      res.status(404).json({error: 'User not found'});
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }

    const recensisci = await itinerarioManager.saveReview(idItinerario, userId, recensione, punteggio);
    res.status(200).json(recensisci);
  } catch (error) {
    console.error('Error saving review:', error);
    res.status(500).send('Internal Server Error: ' + error);
  }
});
```

.getItineraryReview

Questa route gestisce una richiesta GET all'endpoint '/getItineraryReview'.

1. Recupera l'ID dell'utente e l'ID dell'itinerario dalla richiesta.
2. Verifica se entrambi gli ID dell'utente e dell'itinerario sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni `checkUser(req)` e `checkItinerary(itineraryId)`.
4. Chiama la funzione `'itinerarioManager.getItineraryReview(idItinerario, userId)'` per ottenere le recensioni dell'itinerario fatte dall'utente.
5. Se non ci sono recensioni per l'itinerario, viene restituito un oggetto JSON vuoto con uno stato 200 (OK).
6. Altrimenti, restituisce le recensioni come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il recupero delle recensioni, viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.get('/getItineraryReview', async (req, res) => {
  try {
    const userId = req.header('userId');
    const idItinerario = req.query.idItinerario;
    if (!userId || !idItinerario) {
      res.status(400).send('Bad Request: userId and idItinerario are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    if (!(await checkUser(req))) {
      console.error('User not found')
      res.status(404).json({error: 'User not found'});
      return;
    }

    const reviews = await itinerarioManager.getItineraryReview(idItinerario, req.header('userId'));
    if (!reviews) {
      res.json({}); // empty object
      return;
    }
    res.status(200).json(reviews);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

.containsDay

Questa route gestisce una richiesta GET all'endpoint '/containsDay'.

1. Recupera l'ID dell'itinerario e il giorno dalla query della richiesta.
2. Verifica se entrambi l'ID dell'itinerario e il giorno sono stati forniti. Se uno dei due è mancante, viene restituito uno stato 400 (Bad Request).
3. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni `checkUser(req)` e `checkItinerary(itineraryId)`.
4. Chiama la funzione `'itinerarioManager.contieneGiorno(giorno, idItinerario)'` per verificare se l'itinerario contiene il giorno specificato.
5. Il risultato della verifica viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante la verifica, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.get('/containsDay', async (req, res) => {
  try {
    const idItinerario = req.query.idItinerario;
    const giorno = req.query.giorno;

    if (!idItinerario || !giorno) {
      res.status(400).send('Bad Request: idItinerario and giorno are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }

    const contieneGiorno = await itinerarioManager.contieneGiorno(giorno, idItinerario);
    res.status(200).json(contieneGiorno);
  } catch (error) {
    console.log(error);
    res.status(500).send('Internal Server Error: ' + error);
  }
});
```

searchItineraries

Questa route gestisce una richiesta GET all'endpoint '/searchItineraries'.

1. Recupera i parametri di ricerca (state, name, duration) dalla query della richiesta.
2. Chiama la funzione `itinerarioManager.cercaItinerari(state, name, duration)` per cercare gli itinerari corrispondenti ai criteri di ricerca forniti.
3. Il risultato della ricerca viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante la ricerca, viene restituito uno stato 500 (Internal Server Error) con un messaggio che contiene la descrizione dell'errore.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.get('/searchItineraries', async (req, res) => {
  try {
    const {state, name, duration} = req.query;
    const itineraries = await itinerarioManager.cercaItinerari(state, name, duration);
    console.log(itineraries)
    res.json(itineraries);
  } catch (error) {
    res.status(500).json({error: error.toString()});
  }
});
```

.createItinerary

Questa route gestisce una richiesta POST all'endpoint '/createItinerary'.

1. Recupera l'ID dell'utente dall'header della richiesta.
2. Verifica se l'ID dell'utente è stato fornito. Se manca, viene restituito uno stato 400 (Bad Request).
3. Verifica se l'utente esiste utilizzando la funzione `checkUser(req)`. Se l'utente non esiste, viene restituito uno stato 404 (Not Found).
4. Recupera i dati dell'itinerario (nome, stato, giorni, recensioni, descrizione, attivo) dal corpo della richiesta.
5. Verifica se sono stati forniti tutti i campi obbligatori (nome, stato, giorni e descrizione). Se uno di essi manca, viene restituito uno stato 400 (Bad Request).
6. Se le recensioni non vengono fornite viene inizializzato come array vuoto
7. I dati dell'itinerario vengono salvati in un oggetto itineraryData.
8. Chiama la funzione `itinerarioManager.createItinerary(itineraryData)` per creare un nuovo itinerario associato all'utente.
9. Il nuovo itinerario creato viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante la creazione dell'itinerario, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.post('/createItinerary', async (req, res) => {
  try {
    const userId = req.header('userId')
    if (!userId) {
      res.status(400).send('Bad Request: userId is required');
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    const nome = req.body.nome;
    const stato = req.body.stato;
    const giorni = req.body.giorni;
    let recensioni = req.body.recensioni;
    const descrizione = req.body.descrizione;
    let attivo = req.body.attivo;
    if (!nome || !stato || !giorni || !descrizione) {
      res.status(400).send('Bad Request: nome, stato, giorni, and descrizione are required');
      return;
    }
    if (!recensioni) {
      recensioni = [];
    }
    if (!attivo) {
      // console.log("attivo")
      // res.status(400).send('Bad Request: attivo is required');
      // return;
      attivo = true;
    }
    const itineraryData = {
      nome,
      stato,
      giorni,
      recensioni,
      descrizione,
      attivo
    };
    const newItinerary = await itinerarioManager.createItinerary({
      ...itineraryData,
      _userId: userId
    });
    res.status(200).json(newItinerary);
  } catch (error) {
    console.error('Error creating new itinerary:', error);
    res.status(500).send('Internal Server Error');
  }
});
```

getCommunityItineraries

Questa route gestisce una richiesta GET all'endpoint '/getCommunityItineraries'.

1. Recupera l'ID dell'utente dall'header della richiesta.
2. Verifica se l'ID dell'utente è stato fornito. Se manca, viene restituito uno stato 400 (Bad Request).
3. Verifica se l'utente esiste utilizzando la funzione checkUser(req). Se l'utente non esiste, viene restituito uno stato 404 (Not Found).
4. Chiama la funzione itinerarioManager.getCommunityItineraries(userId) per ottenere tutti gli itinerari della community.
5. Gli itinerari della community vengono restituiti come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il recupero degli itinerari, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error)

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.get('/getCommunityItineraries', async (req, res) => {
    try {
        const userId = req.header('userId');
        if (!userId) {
            res.status(400).send('Bad Request: userId is required');
            return;
        }
        if (!(await checkUser(req))) {
            res.status(404).json({error: 'User not found'});
            return;
        }
        const allItineraries = await itinerarioManager.getCommunityItineraries(userId);
        res.status(200).json(allItineraries);
    } catch (error) {
        console.error('Error fetching all itineraries:', error);
        res.status(500).send('Internal Server Error');
    }
});
```

.addStop

Questa route gestisce una richiesta POST all'endpoint '/addStop'.

1. Recupera l'ID dell'itinerario, il giorno e la tappa dalla richiesta.
2. Recupera anche l'ID dell'utente dall'header della richiesta.
3. Verifica se tutti i parametri necessari sono stati forniti. Se uno di essi manca, viene restituito uno stato 400 (Bad Request).
4. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni checkUser(req) e checkItinerary(itineraryId).
5. Chiama la funzione `giornoManager.aggiungiTappa(idItinerario, giorno, tappa)` per aggiungere la tappa all'itinerario.
6. Il risultato dell'operazione viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante l'aggiunta della tappa, viene restituito uno stato 500 (Internal Server Error).

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.post('/addStop', async (req, res) => {
  try {
    const idItinerario = req.body.idItinerario;
    const giorno = req.body.giorno;
    const tappa = req.body.tappa;
    const userId = req.header('userId');
    if (!idItinerario || !giorno || !tappa || !userId) {
      res.status(400).send('Bad Request: idItinerario, giorno, tappa and userId are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }

    const aggiungiTappa = await giornoManager.aggiungiTappa(idItinerario, giorno, tappa);
    res.status(200).json(aggiungiTappa);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

.deleteStop

Questa route gestisce una richiesta DELETE all'endpoint '/deleteStop'.

1. Recupera l'ID dell'itinerario, il giorno e la tappa dalla richiesta.
2. Recupera anche l'ID dell'utente dall'header della richiesta.
3. Verifica se tutti i parametri necessari sono stati forniti. Se uno di essi manca, viene restituito uno stato 400 (Bad Request).
4. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni checkUser(req) e checkItinerary(itineraryId).
5. Chiama la funzione `giornoManager.eliminaTappa(idItinerario, giorno, tappa)` per eliminare la tappa dell'itinerario.
6. Il risultato dell'operazione viene restituito come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante l'eliminazione della tappa, viene registrato un messaggio di errore nella console e viene restituito uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●

router.delete('/deleteStop', async (req, res) => {
  try {
    const idItinerario = req.body.idItinerario;
    const giorno = req.body.giorno;
    const tappa = req.body.tappa;
    const userId = req.header('userId');
    if (!idItinerario || !giorno || !tappa || !userId) {
      res.status(400).send('Bad Request: idItinerario, giorno, tappa and userId are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }
    const eliminaTappa = await giornoManager.eliminaTappa(idItinerario, giorno, tappa);
    res.status(200).json(eliminaTappa);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

replaceStop

Questa route gestisce una richiesta PUT all'endpoint '/replaceStop'.

1. Recupera l'ID dell'itinerario, il giorno, l'indice e la tappa dal corpo della richiesta e l'ID dell'utente dall'header della richiesta.
2. Verifica se l'ID dell'itinerario, il giorno e la tappa sono stati forniti. Se uno di essi è mancante, registra un messaggio di errore nella console e restituisce uno stato 400 (Bad Request) con un messaggio che indica quali parametri mancano.
3. Viene verificato se l'itinerario esiste utilizzando checkItinerary(itineraryId).
4. Chiama la funzione `giornoManager.ripiazzaTappa(idItinerario, giorno, indice, tappa)` per sostituire la tappa nell'itinerario.
5. Restituisce il risultato dell'operazione come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il processo, registra un messaggio di errore nella console e restituisce uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
● ● ●
router.put('/replaceStop', async (req, res) => {
  try {
    const idItinerario = req.body.idItinerario;
    const giorno = req.body.giorno;
    const indice = req.body.indice;
    const tappa = req.body.tappa;
    const userId = req.header('userId');
    if (!idItinerario || !giorno || !tappa) {
      console.error('Bad Request: idItinerario: ' + idItinerario + ' giorno: ' + giorno + ' indice: ' + indice + ' tappa: ' + tappa + ' userId: ' + userId);
      res.status(400).send('Bad Request: idItinerario, giorno, indice, tappa and userId are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      console.error('Itinerary not found')
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }

    const ripiazzaTappa = await giornoManager.ripiazzaTappa(idItinerario, giorno, indice, tappa);
    res.status(200).json(ripiizzaTappa);
  } catch (error) {
    console.error('Error replacing stop:', error);
    res.status(500).send('Internal Server Error: ' + error);
  }
});
```

.calcDistance

Questa route gestisce una richiesta GET all'endpoint '/calcDistance'. Ecco come funziona:

1. Verifica se l'ID dell'itinerario, il giorno, il mezzo di trasporto e l'ID dell'utente sono stati forniti. Se uno di essi è mancante, restituisce uno stato 400 (Bad Request) con un messaggio che indica quali parametri mancano.
2. Viene verificato se l'utente e l'itinerario esistono utilizzando le funzioni checkUser(req) e checkItinerary(itineraryId).
3. Chiama la funzione `giornoManager.calcolaDistanza(idItinerario, giorno, mezzo)` per calcolare la distanza percorsa nel giorno specificato dell'itinerario utilizzando il mezzo di trasporto specificato.
4. Restituisce il risultato del calcolo della distanza come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il processo, restituisce uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
router.get('/calcDistance', async (req, res) => {
  try {
    const idItinerario = req.query.idItinerario;
    const giorno = req.query.giorno;
    const mezzo = req.query.mezzo;
    const userId = req.header('userId');
    if (!idItinerario || !giorno || !mezzo || !userId) {
      res.status(400).send('Bad Request: idItinerario, giorno, mezzo and userId are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    if (!(await checkUser(req))) {
      res.status(404).json({error: 'User not found'});
      return;
    }

    const calcolaDistanza = await giornoManager.calcolaDistanza(idItinerario, giorno, mezzo);
    res.status(200).json(calcolaDistanza);
  } catch (error) {
    res.status(500).send('Internal Server Error ' + error);
  }
});
```

.calcPath

Questa route gestisce una richiesta GET all'endpoint '/calcPath'. Ecco come funziona:

1. Verifica se l'ID dell'itinerario e il giorno sono stati forniti. Se uno dei due è mancante, restituisce uno stato 400 (Bad Request) con un messaggio che indica quali parametri mancano.
2. Viene verificato se l'itinerario esiste utilizzando la funzione `checkItinerary(itineraryId)`.
3. Chiama la funzione `'giornoManager.calcolaPercorso(idItinerario, giorno)'` per calcolare il percorso ottimale per il giorno specificato dell'itinerario.
4. Restituisce il percorso calcolato come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il processo, restituisce uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

Le implementazioni delle funzioni li si trovano nella CodeBase.

```
  ● ● ●

router.get('/calcPath', async (req, res) => {
  try {
    const idItinerario = req.query.idItinerario;
    const giorno = req.query.giorno;
    if (!idItinerario || !giorno) {
      res.status(400).send('Bad Request: idItinerario and giorno are required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }

    const calcolaPercorso = await giornoManager.calcolaPercorso(idItinerario, giorno);
    res.json(calcolaPercorso);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

.totalPath

Questa route gestisce una richiesta GET all'endpoint '/totalPath'. Ecco come funziona:

1. Verifica se l'ID dell'itinerario è stato fornito. Se manca, restituisce uno stato 400 (Bad Request) con un messaggio che indica che l'ID dell'itinerario è richiesto.
2. Viene verificato se l'itinerario esiste utilizzando la funzione `checkItinerary(itineraryId)`.
3. Chiama la funzione `'giornoManager.calcolaPercorsoTotale(idItinerario)'` per calcolare il percorso totale ottimale per l'itinerario specificato.
4. Restituisce il percorso totale calcolato come risposta JSON con uno stato 200 (OK).

Se si verifica un errore durante il processo, restituisce uno stato 500 (Internal Server Error) con un messaggio che indica che si è verificato un errore interno del server.

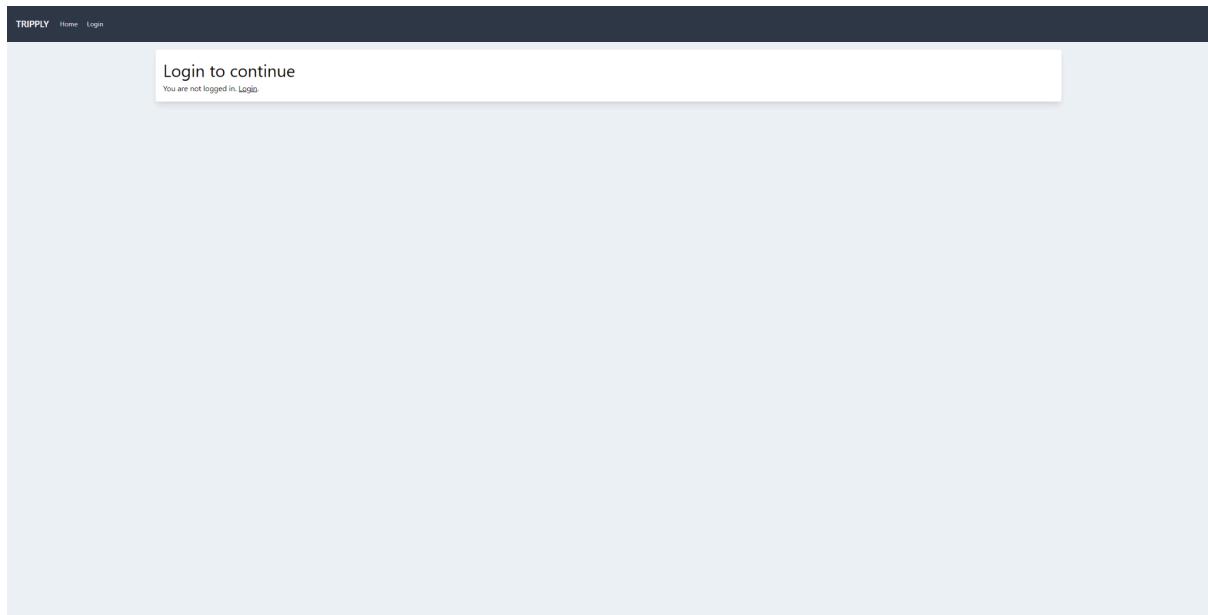
Il risultato della verifica viene restituito come risposta JSON con uno stato 200 (OK).

```
router.get('/totalPath', async (req, res) => {
  try {
    const idItinerario = req.query.idItinerario;
    if (!idItinerario) {
      res.status(400).send('Bad Request: idItinerario is required');
      return;
    }
    if (!(await checkItinerary(idItinerario))) {
      res.status(404).json({error: 'Itinerary not found'});
      return;
    }
    const totalPath = await giornoManager.calcolaPercorsoTotale(idItinerario);
    res.status(200).json(totalPath);
  } catch (error) {
    res.status(500).send('Internal Server Error');
  }
});
```

Frontend Implementation

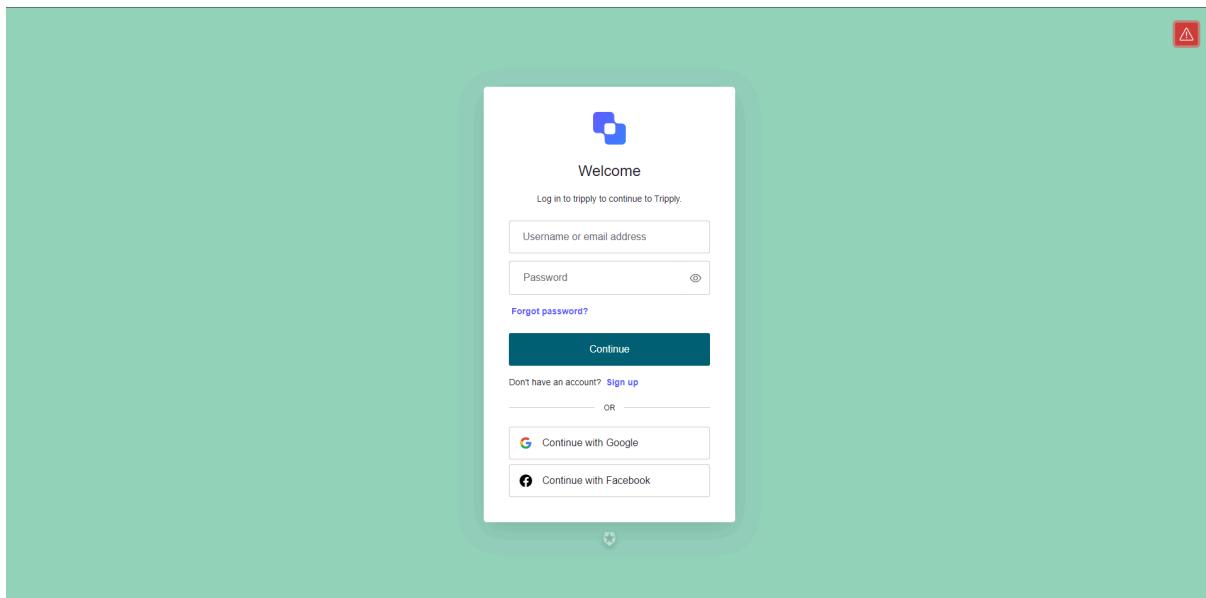
1. Pagina Home

La prima pagina a cui si accede se non si è autenticati da essa si può effettuare il login



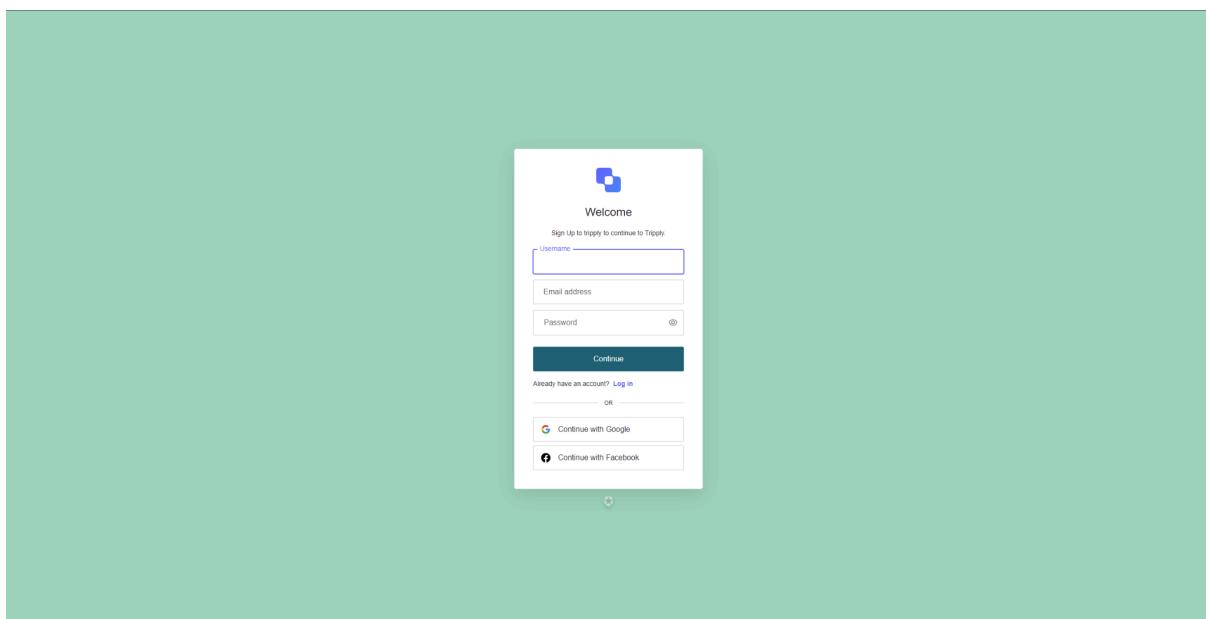
2. Pagina Login

Questa è la schermata di accesso, dove gli utenti possono inserire le proprie credenziali per accedere all'applicazione. Inoltre, è possibile registrarsi come nuovo utente se non si dispone ancora di un account oppure recuperare la password in caso di smarrimento.



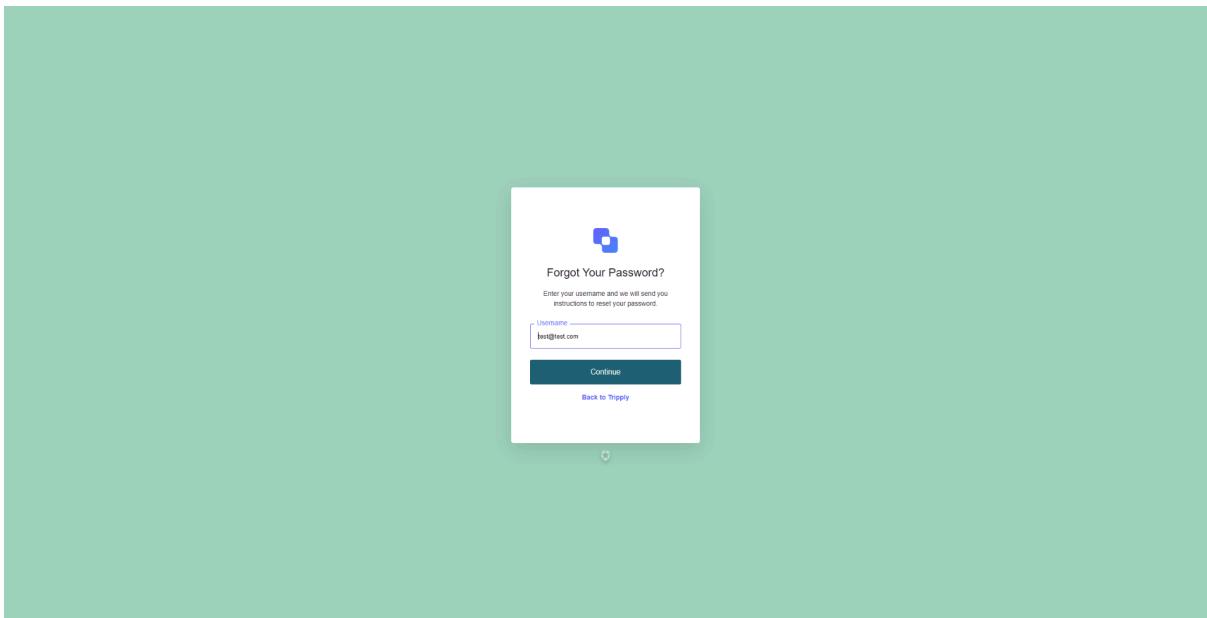
3. Pagina Registrazione

Viene chiesto all'utente di inserire l'username, la mail ed una password



4. Pagina Recupero password

Viene chiesto la mail di recupero password



5. Pagina Home/Itinerary community

Questa è la prima pagina dell'utente autenticato, dove vengono visualizzati gli itinerari della community. Gli utenti possono esplorare gli itinerari proposti dagli altri utenti e hanno la possibilità di salvarli o recensirli.

The screenshot shows the TRIPPLY platform's user interface for managing itineraries. At the top, there is a navigation bar with links for 'Home/Itinerari community', 'I miei itinerari', 'Salvati', 'Crea Itinerario', and 'Cerca Itinerari'. On the right side of the header is a 'Profilo' button. The main content area is titled 'Community Itineraries' and displays two separate itinerary cards.

Card 1: Pisa Itinerary

- Pisa Itinerary 1706993662679
- Test Descrizione 1706993662679
- Creatore: 65a9e460b1c0d00bc09c34b
- Numero di giorni: 3
- Stato: Attivo
- Kilometri totali: 6410.413 km
- Tempo totale: 69.2 ore

Card 2: Florence Itinerary

- Florence Itinerary 1707068310845
- Test Descrizione 1707068310845
- Creatore: google-oauth2|2111261636165234106646
- Numero di giorni: 3
- Stato: Attivo
- Kilometri totali: 2390.973 km
- Tempo totale: 30.8 ore

Both cards feature three buttons at the bottom: 'Toggle Days' (blue), 'Unsave' (red), and 'Recensioni' (blue).

a. Pagina Home/Itinerari community itinerario esteso

This screenshot shows the same TRIPPLY interface, but the second itinerary card is expanded to show its detailed components. The card is titled 'Days: Test Day Descrizione 1706993662679'.

Day 1: Test Day Descrizione 1706993662679

- Tappa 1
 - Descrizione: Test Tappa Descrizione 1706993662679
 - Location: Roma
 - Ristori: Test Tappa Ristori BjQhQzWk7
 - Alloggi: Test Tappa Alloggi 1160875sqJ
- Tappa 2
 - Descrizione: Test Tappa Descrizione 1706993662679
 - Location: Messina
 - Ristori: Test Tappa Ristori Vcd09taFaw
 - Alloggi: Test Tappa Alloggi NpBannG6V
- Tappa 3
 - Descrizione: Test Tappa Descrizione 1706993662679
 - Location: Palermo
 - Ristori: Test Tappa Ristori Id6wwjDPHu
 - Alloggi: Test Tappa Alloggi UCE99mBHU

Day 2: Test Day Descrizione 1706993662679

- Tappa 1
 - Descrizione: Test Tappa Descrizione 1706993662679
 - Location: Modena
 - Ristori: Test Tappa Ristori 18UK9KXdq
 - Alloggi: Test Tappa Alloggi a3mransb8P
- Tappa 2
 - Descrizione: Test Tappa Descrizione 1706993662679
 - Location: Ferrara
 - Ristori: Test Tappa Ristori yqg3708TE9
 - Alloggi: Test Tappa Alloggi JAOX9AEKd

Day 3: Test Day Descrizione 1706993662679

- Tappa 1
 - Descrizione: Test Tappa Descrizione 1706993662679
 - Location: Roma
 - Ristori: Test Tappa Ristori 18UK9KXdq
 - Alloggi: Test Tappa Alloggi a3mransb8P

b. Pagina Home/Itinerari community recensioni

The screenshot displays the TRIPPLY platform's 'Community Itineraries' section. Two itinerary cards are shown side-by-side.

Top Itinerary Card (Pisa Itinerary):

- Pisa Itinerary 1706993663679
- Test Description 170699366279
- Creatore 65ae9ad0b1cd00bd09c34b
- Numero di giorni: 3
- Stato: Italy
- Kilometri totali: 6410.413 km
- Tempo totale: 69.2 ore

Buttons: **Toggle Days**, **Unsave**, **Recensisci**.

Form fields:
Scriv la tua recensione...
Valutazione: 1
Save recensione

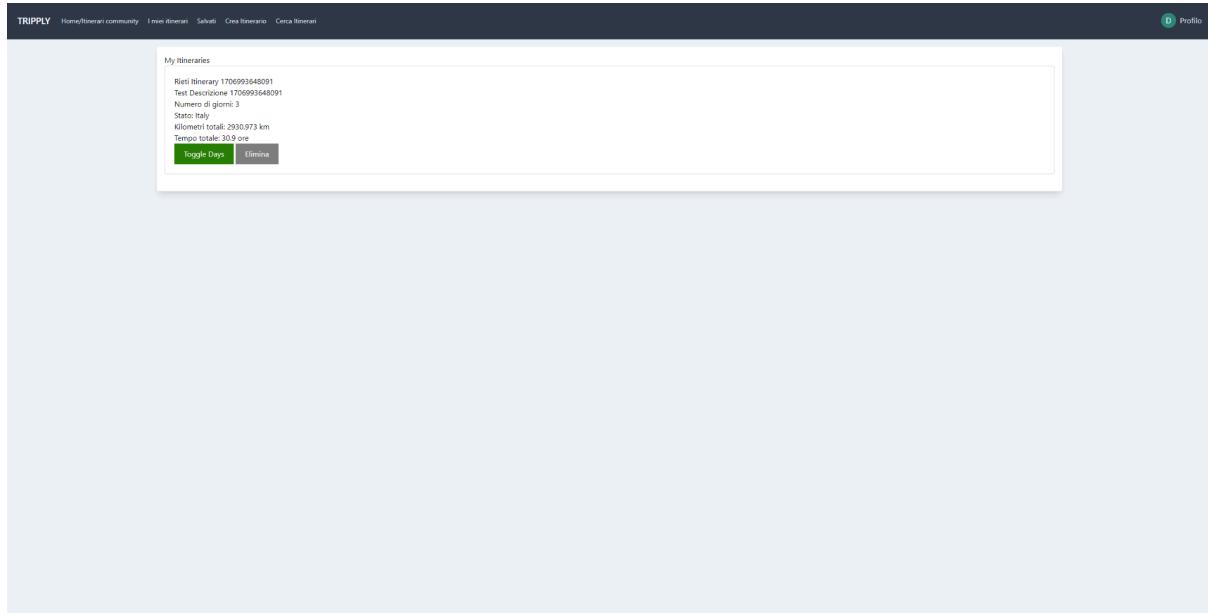
Bottom Itinerary Card (Florence Itinerary):

- Florence Itinerary 1707068310845
- Test Description 1707068310845
- Creatore google-oauth2|111261636165234106646
- Numero di giorni: 3
- Stato: Italy
- Kilometri totali: 2930.973 km
- Tempo totale: 30.9 ore

Buttons: **Toggle Days**, **Save**, **Recensisci**.

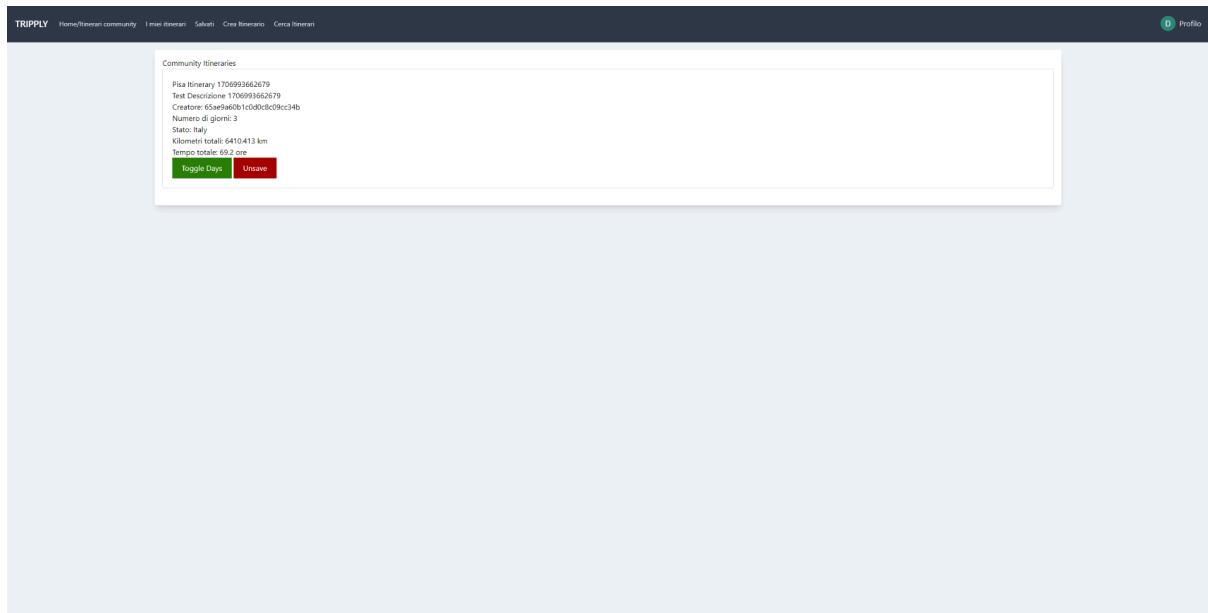
6. Pagina i miei itinerari

Questa è la dove l'utente trova gli itinerari da lui creati



7. Pagina itinerari salvati

Questa è la dove l'utente trova gli itinerari che ha salvato



8. Pagina crea

Questa è la dove l'utente crea gli itinerari

9. Pagina ricerca

Questa è la dove l'utente effettua una ricerca sugli itinerari tramite stato, durata e nome

10.Pagina Profilo

In questa pagina si ritrovano le informazioni dell'utente. Qui si può effettuare il logout e il cancella profilo.

The screenshot shows the TRIPPLY profile page. At the top, there is a dark header bar with the TRIPPLY logo and navigation links: Home/Itinerari.community, I miei itinerari, Saluto, Crea Itinerario, and Cerca Itinerari. On the right side of the header is a pink circular icon with a white profile picture and the word "Profilo". The main content area has a light gray background. At the top left, it says "Welcome test". Below this is a large pink square placeholder for a profile picture, with the letters "TE" in white. Underneath the placeholder, a note states: "This is the content of req.user. Note: __raw and __json properties have been omitted." Below this note is a JSON object representing the user profile:

```
{ "id": "77e0c16d14db0qpxXwv54qptYnx", "nickname": "test", "name": "test@test.com", "picture": "https://www.proavatar.com/avatar/8642b4217b34012b8d0bd019Fc05c449271a488d~pg8d~https%3A%2F%2Fcdn.auth0.com%2Favatars%2Fte.png", "updated_at": "2024-02-07T08:13:48.611Z", "email": "test@test.com", "email_verified": false, "uid": "auth0|68f72aefaf522fcfd7805a5c" }
```

At the bottom of the profile section, there are two buttons: a blue "Logout" button and a red "Delete Account" button.

GitHub Repository and Deployment Info

Nel seguente Link si ritrovano la repository dove si trovano i Deliverables e gli allegati
<https://github.com/GruppoG42/Deliverables>.

In ogni cartella si trova le cartelle contenenti le versioni oppure il documento direttamente e una cartella UML contenente i schemi utilizzati.

Il deployment del progetto è stato effettuato con la piattaforma [render.com](#) e di seguito il link per visualizzare la web app:

<https://trippy.onrender.com/>

Nota:

- Per quanto riguarda il pacchetto gratuito della piattaforma e le caratteristiche di Express.js, alla prima richiesta al sito, sarà necessario attendere 30 secondi. Durante questo tempo, Render.com utilizzerà un'immagine di Docker per creare il container necessario.
- Account Test:
 - email: test@test.com
 - pwd: TestTest1.
- Per comodità abbiamo lasciato un pulsante che carica casualmente valori nel form di creazione

Il deployment del progetto in locale:

Configurazione:

- clone repository CodeBase
- npm install
- npm run start

API DOCUMENTATION

Le API locali fornite dall'applicazione Tripply sono state documentate utilizzando il modulo swagger-ui-express, seguendo lo standard OpenAPI. Sono stati descritti i vari endpoint utilizzati dall'applicazione.

Puoi consultare la documentazione all'indirizzo seguente: [\[link per la documentazione\]](#).

Durante lo sviluppo delle API vengono utilizzate le funzioni GET, POST, PATCH, PUT e DELETE per consentire a diverse applicazioni e servizi di interagire tra loro fornendo strumenti flessibili.

Nota: Si consiglia di fare attenzione quando si utilizzano API come /api/createItinerary, che richiedono molti parametri, attraverso la GUI di Swagger.

Itinerario Itinerario API

^

<code>GET</code>	/api/getUserItineraries	Get all user itineraries		
<code>GET</code>	/api/calcTimeItinerary	Calculate the time for an itinerary		
<code>POST</code>	/api/reviewItinerary	Add a review to an itinerary		
<code>GET</code>	/api/getItineraryReview	Get reviews for an itinerary		
<code>GET</code>	/api/searchItineraries	Search for itineraries		
<code>POST</code>	/api/createItinerary	Create a new itinerary		
<code>GET</code>	/api/getCommunityItineraries	Get all community itineraries		
<code>GET</code>	/api/totalPath	Calculate the total path for an itinerary		

Trippler Trippler API

^

<code>DELETE</code>	/api/deleteItinerary	Delete an itinerary		
<code>GET</code>	/api/getUser	Get user details		
<code>DELETE</code>	/api/deleteUser	Delete a user		
<code>GET</code>	/api/getSavedItineraries	Get saved itineraries for a user		
<code>GET</code>	/api/isSavedItinerary	Check if an itinerary is saved by a user		
<code>PATCH</code>	/api/addItineraryToSaved	Add an itinerary to the saved list of a user		
<code>PATCH</code>	/api/removeItineraryFromSaved	Remove an itinerary from the saved list of a user		
<code>DELETE</code>	/api/deleteSavedList	Delete the saved list of a user		

Giorno Giorno API

^

<code>POST</code>	/api/addDay	Add a day to an itinerary		
<code>GET</code>	/api/containsDay	Check if a day is in an itinerary		
<code>GET</code>	/api/calcDistance	Calculate the distance for a day in an itinerary		
<code>GET</code>	/api/calcPath	Calculate the path for a day in an itinerary		

Tappa Tappa API

^

<code>POST</code>	/api/addStop	Add a stop to a day in an itinerary		
<code>DELETE</code>	/api/deleteStop	Delete a stop from a day in an itinerary		
<code>PUT</code>	/api/replaceStop	Replace a stop in a day in an itinerary		

Testing

Per eseguire i test, abbiamo utilizzato Mocha, il quale è stato configurato nello script `test/ItineraryTest.js`.

Per effettuare i test:

- npm run start
 - npm run test

1. Test /api/getUserItineraries

2. Test /api/calcTimeItinerary

Tempo percorrenza: 111179 ✓ should calculate time (423ms)

3. Test /api/reviewItinerary

```
{ review: 'Great itinerary!', rating: 5 }  
    ✓ should POST a review for an itinerary
```

4. Test /api/getItineraryReview

```
{  
  _userId: 'google-oauth2|111261636165234106646',  
  recensione: 'Great itinerary!',  
  punteggio: 5  
}  
  
✓ should GET a review for an itinerary
```

5. Test /api/addDay

```
{
  descrizione: 'Test Day Descrizione 1706993659036',
  tappe: [
    {
      descrizione: 'Test Tappa Descrizione 1706993659036',
      luogo: 'Bolzano',
      ristori: 'Test Tappa Ristori KqqSwE1ZBd',
      alloggi: 'Test Tappa Alloggi MUpKRNhi8t'
    },
    {
      descrizione: 'Test Tappa Descrizione 1706993659036',
      luogo: 'Grosseto',
      ristori: 'Test Tappa Ristori q9Z95SjjTz',
      alloggi: 'Test Tappa Alloggi I4c6X9AVT2'
    },
    {
      descrizione: 'Test Tappa Descrizione 1706993659036',
      luogo: 'Padova',
      ristori: 'Test Tappa Ristori Z1KnV7gMB7',
      alloggi: 'Test Tappa Alloggi G3VPhFQ6VD'
    }
  ]
}
  ✓ should add a day to an itinerary
```

6. Test /api/containsDay

```
true
  ✓ should check if an itinerary contains a day
```

7. Test /api/searchItineraries

```
[
  {
    _id: '65bea7f2b410532d3c471d34',
    nome: 'Rieti Itinerary 1706993648091',
    stato: 'Italy',
    giorni: [ [Object], [Object], [Object] ],
    recensioni: [ [Object] ],
    descrizione: 'Test Descrizione 1706993648091',
    attivo: true,
    _userId: 'google-oauth2|109681722197817748077'
  }
]
  ✓ should search itineraries
```

8. Test /api/createItinerary

```
{ acknowledged: true, insertedId: '65c34e7187ec49f52cb19b7d' }  
  ✓ should create an itinerary
```

9. Test /api/getCommunityItineraries

```
[  
  {  
    _id: '65bea7f2b410532d3c471d34',  
    nome: 'Rieti Itinerary 1706993648091',  
    stato: 'Italy',  
    giorni: [ [Object], [Object], [Object] ],  
    recensioni: [ [Object] ],  
    descrizione: 'Test Descrizione 1706993648091',  
    attivo: true,  
    _userId: 'google-oauth2|109681722197817748077'  
  },  
  {  
    _id: '65bea7fcbb410532d3c471d35',  
    nome: 'Campobasso Itinerary 1706993659036',  
    stato: 'Italy',  
    giorni: [  
      [Object],  
      [Object],  
      [Object],  
      '{\n' +  
        '      "descrizione": "Test Day Descrizione 1706993659036",\n' +  
        '      "tappe": [\n' +  
        '        {\n' +  
        '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
        '          "luogo": "Bolzano",\n' +  
        '          "ristori": "Test Tappa Ristori KqqSwE1ZBd",\n' +  
        '          "alloggi": "Test Tappa Alloggi MUUpKRNh18t"\n' +  
        '        },\n' +  
        '        {\n' +  
        '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
        '          "luogo": "Grosseto",\n' +  
        '          "ristori": "Test Tappa Ristori q9Z95SjjTz",\n' +  
        '          "alloggi": "Test Tappa Alloggi I4c6X9AVT2"\n' +  
        '        },\n' +  
        '        {\n' +  
        '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
        '          "luogo": "Padova",\n' +  
        '          "ristori": "Test Tappa Ristori Z1KnV7gMB7",\n' +  
        '          "alloggi": "Test Tappa Alloggi G3VPhFQ6VD"\n' +  
        '        }\n' +  
      ]\n' +  
    },  
    '{\n' +  
      '      "descrizione": "Test Day Descrizione 1706993659036",\n' +  
      '      "tappe": [\n' +  
      '        {\n' +  
      '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
      '          "luogo": "Bolzano",\n' +  
      '          "ristori": "Test Tappa Ristori KqqSwE1ZBd",\n' +  
      '          "alloggi": "Test Tappa Alloggi MUUpKRNh18t"\n' +  
      '        },\n' +  
      '        {\n' +  
      '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
      '          "luogo": "Grosseto",\n' +  
      '          "ristori": "Test Tappa Ristori q9Z95SjjTz",\n' +  
      '          "alloggi": "Test Tappa Alloggi I4c6X9AVT2"\n' +  
      '        },\n' +  
      '        {\n' +  
      '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
      '          "luogo": "Padova",\n' +  
      '          "ristori": "Test Tappa Ristori Z1KnV7gMB7",\n' +  
      '          "alloggi": "Test Tappa Alloggi G3VPhFQ6VD"\n' +  
      '        }\n' +  
    ]\n' +  
  },  
  '{\n' +  
    '      "descrizione": "Test Day Descrizione 1706993659036",\n' +  
    '      "tappe": [\n' +  
    '        {\n' +  
    '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
    '          "luogo": "Bolzano",\n' +  
    '          "ristori": "Test Tappa Ristori KqqSwE1ZBd",\n' +  
    '          "alloggi": "Test Tappa Alloggi MUUpKRNh18t"\n' +  
    '        },  
    '        {\n' +  
    '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
    '          "luogo": "Grosseto",\n' +  
    '          "ristori": "Test Tappa Ristori q9Z95SjjTz",\n' +  
    '          "alloggi": "Test Tappa Alloggi I4c6X9AVT2"\n' +  
    '        },  
    '        {\n' +  
    '          "descrizione": "Test Tappa Descrizione 1706993659036",\n' +  
    '          "luogo": "Padova",\n' +  
    '          "ristori": "Test Tappa Ristori Z1KnV7gMB7",\n' +  
    '          "alloggi": "Test Tappa Alloggi G3VPhFQ6VD"\n' +  
    '        }\n' +  
  ]\n' +  
}
```

```
[Object],  
[Object],  
[Object],  
[Object]  
],  
recensioni: [ [Object] ],  
descrizione: 'Test Descrizione 1706993659036'  
attivo: true,  
_userId: 'google-oauth2|109681722197817748077'  
},  
{  
_id: '65bea7ffb410532d3c471d36',  
nome: 'Pisa Itinerary 1706993662679',  
stato: 'Italy',  
giorni: [ [Object], [Object], [Object] ],  
recensioni: [ [Object] ],  
descrizione: 'Test Descrizione 1706993662679'  
attivo: true,  
_userId: '65ae9a60b1c0d0c8c09cc34b'  
},  
{  
_id: '65bfcc8be5153f7439cc9351',  
nome: 'Viterbo Itinerary 1707068553601',  
stato: 'Italy',  
giorni: [ [Object], [Object], [Object] ],  
recensioni: [],  
descrizione: 'Test Descrizione 1707068553601'  
attivo: true,  
_userId: 'google-oauth2|109681722197817748077'  
},  
{  
_id: '65c15edf6bbd7e358c4630c5',  
nome: 'Pescara Itinerary 1707171549673',  
stato: 'Italy',  
giorni: [ [Object], [Object], [Object] ],  
recensioni: [],  
descrizione: 'Test Descrizione 1707171549673'  
attivo: true,  
_userId: 'google-oauth2|109681722197817748077'  
},  
{  
_id: '65c34b7f87ec49f52cb19b7b',  
nome: 'Pescara Itinerary 1707297662503',  
stato: 'Italy',  
giorni: [ [Object], [Object], [Object] ],  
recensioni: [],  
descrizione: 'Test Descrizione 1707297662503'  
attivo: true,  
_userId: 'auth0|65bf72e2daf22cfcd7865a3c'  
}  
]  
✓ should get community itineraries (91ms)
```

10. Test /api/deleteItinerary

```
{
  acknowledged: true, deletedCount: 1
}
  ✓ should delete an itinerary
```

11. Test /api/getSavedItineraries

```
[
  {
    _id: '65bea7f2b410532d3c471d34',
    nome: 'Rieti Itinerary 1706993648091',
    stato: 'Italy',
    giorni: [ [Object], [Object], [Object] ],
    recensioni: [ [Object] ],
    descrizione: 'Test Descrizione 1706993648091',
    attivo: true,
    _userId: 'google-oauth2|109681722197817748077'
  }
]
  ✓ should get saved itineraries
```

12. Test /api/addItineraryToSaved

```
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
  ✓ should add an itinerary to saved (64ms)
```

13. Test /api/isSavedItinerary

```
true
  ✓ should check if an itinerary is saved
```

14. Test /api/removeItineraryFromSaved

```
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
  ✓ should remove an itinerary from saved (71ms)
```

15. Test /api/deleteSavedList

```
{ acknowledged: true, deletedCount: 0 }
  ✓ should delete a saved list
```

16. Test /api/addStop

```
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
  ✓ should add a stop to an itinerary (42ms)
```

17. Test /api/deleteStop

```
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
  ✓ should delete a stop from an itinerary (39ms)
```

18. Test /api/replaceStop

```
{
  acknowledged: true,
  modifiedCount: 1,
  upsertedId: null,
  upsertedCount: 0,
  matchedCount: 1
}
  ✓ should replace a stop in an itinerary (78ms)
```

19. Test /api/calcDistance

```
{ text: '57.8 km', value: 57763 }
  ✓ should calculate distance (293ms)
```

20. Test /api/calcPath

```
1053915
  ✓ should calculate path (355ms)
```