UNIVERSITÀ
DEGLI STUDI
DI UDINE

Documentazione finale di Air Connect

Corso: Cybersecurity

Nome Studente: Stefano Panico

Matricola: 169091

Indice generale

1. Introduzione

Air Connect è una piattaforma web progettata per semplificare e ottimizzare l'intero processo di gestione dei **voli**, rendendo l'esperienza dell'utente **intuitiva** e **sicura**.

Il sistema offre funzionalità avanzate per gli **utenti finali**, consentendo loro di effettuare il **check-in**, **acquistare** e **cancellare biglietti** con pochi passaggi.

Grazie a una solida architettura **backend** e a un'interfaccia **frontend** user-friendly, gli utenti possono accedere facilmente ai servizi di **prenotazione voli.**

In aggiunta, **Air Connect** include un modulo di **gestione** dedicato agli **amministratori**, che ha il compito di monitorare e gestire le operazioni quotidiane legate ai **voli**. Gli **amministratori** possono **creare**, **modificare** e **cancellare voli**, assicurando che l'offerta di itinerari sia sempre aggiornata e allineata alle esigenze degli utenti. Il sistema offre inoltre la possibilità di **assegnare ruoli** di **amministratore** ad altri utenti, permettendo una gestione **flessibile** e distribuita delle risorse.

L'**autenticazione** degli utenti e l'accesso alle funzionalità amministrative sono gestiti tramite un sistema di **controllo dei ruoli**, che garantisce che solo gli utenti **autorizzati** possano eseguire operazioni sensibili, come la gestione dei **voli** e dei **permessi**. Inoltre, il sistema è **scalabile** e può facilmente adattarsi a futuri miglioramenti, come l'integrazione con altri sistemi di **viaggio** o l'aggiunta di nuove funzionalità per gli **utenti finali.**

2. Obiettivi del progetto

L'obiettivo principale di questo progetto è sviluppare un'applicazione web sicura e affidabile, in grado di proteggere i dati degli utenti da potenziali minacce informatiche.

Sono state implementate diverse misure di sicurezza per garantire la riservatezza, l'integrità e la disponibilità delle informazioni.

In particolare, il progetto si propone di raggiungere i seguenti obiettivi di sicurezza:

- **Implementazione di un database:** per l'archiviazione delle informazioni degli utenti.
- **Gestione delle sessioni utente**: per prevenire accessi non autorizzati e furti di identità.
- **Implementazione del protocollo HTTPS**: per garantire la cifratura dei dati durante la trasmissione tra il browser dell'utente e il server.
- **Gestione dei cookie**: utilizzo di cookie sicuri e conformi alle normative sulla privacy per l'autenticazione e la gestione delle preferenze degli utenti.
- **Password criptate**: implementazione di un sistema di criptazione delle password, al fine di non memorizzare le password in chiaro.
- **Prevenzione di attacchi comuni**: Implementazione di misure di sicurezza per prevenire attacchi comuni, come cross-site request forgery (CSRF).

Attraverso il raggiungimento di questi obiettivi, il progetto mira a creare un ambiente online sicuro e affidabile per gli utenti

3. Architettura di Air Connect

Il progetto *Air Connect* è strutturato secondo un'**architettura a microservizi**, suddividendo il sistema in componenti indipendenti che comunicano tra loro tramite API REST. Questo approccio migliora la scalabilità, la manutenzione e l'affidabilità dell'applicazione.

3.1. Frontend (React.js)

Il frontend è sviluppato in **React.js** e si occupa di gestire l'interfaccia utente, interagendo con il backend per ottenere e inviare dati.

La struttura del frontend è suddivisa nei seguenti moduli:

- `assets/` → Contiene risorse multimediali (immagini, icone, ecc.).
- `components/` → Racchiude componenti riutilizzabili, come il layout principale (`Layout.js`).
- `pages/` → Contiene le pagine principali dell'applicazione, tra cui:
  - `HomePage.js` (pagina principale)
  - `LoginPage.js` e `RegisterPage.js` (gestione autenticazione)
  - `FlightsPage.js` (visualizzazione voli disponibili)
  - `AddFlightPage.js` e `ManagementFlightPage.js` (gestione amministrativa dei voli)
  - `HistoryPage.js` (storico delle prenotazioni)
  - `ProfilePage.js` (profilo utente)
  - `UserManagementPage.js` (gestione utenti con ruoli di admin)
- `services/` → Gestisce la comunicazione con il backend attraverso API REST. Ogni file si occupa di una specifica funzionalità:
  - `authService.js` (autenticazione)
  - `flightService.js` e `flightAdminService.js` (gestione voli)
  - `ticketService.js` (gestione biglietti)
  - `historyService.js` (storico operazioni)
  - `userService.js` (gestione utenti e ruoli)
- `App.js` → Punto di ingresso principale del frontend, che gestisce il routing e l'inizializzazione dell'applicazione.

3.2. Backend (Node.js + Express)

Il backend è sviluppato in **Node.js** con il framework **Express** e gestisce la logica di business dell'applicazione. Si interfaccia con un database per gestire utenti, voli, biglietti e storico delle operazioni.

La struttura del backend è suddivisa nei seguenti moduli:

- `config/` → Contiene la configurazione del database (`index.js`).
- `middleware/` → Contiene il middleware per la gestione dei permessi utente, ovvero `authMiddleware.js`.
- `models/` → Definisce la struttura delle tabelle nel database:
  - `flightModel.js` (dati relativi ai voli)
  - `historyModel.js` (storico operazioni)
  - `ticketModel.js` (gestione biglietti)
  - `userModel.js` (gestione utenti e ruoli)
- `routes/` → Contiene le API REST che gestiscono le operazioni principali:
  - `authRoute.js` (autenticazione e gestione login/register)
  - `flightRoute.js` (visualizzazione voli per gli utenti e gestione voli per admin)
  - `historyRoute.js` (storico prenotazioni e azioni)
  - `ticketRoute.js` (gestione acquisto, cancellazione e check-in dei biglietti)
  - `userRoleRoute.js` (gestione ruoli amministrativi)
- `app.js` → Avvia il server Express e configura tutte le rotte e i middleware.

3.3. Comunicazione tra Frontend e Backend

Il frontend comunica con il backend tramite **API RESTful**, utilizzando richieste HTTPS per inviare e ricevere dati in formato **JSON**. Le chiamate sono le seguenti:

authRoute.js

- **POST** `/api/auth/register` - Registrazione di un nuovo utente.
- **POST** `/api/auth/login` - Login di un utente.
- **GET** `/api/auth/profile` - Visualizzazione del profilo utente.
- **PUT** `/api/auth/profile/edit` - Modifica del profilo utente.
- **POST** `/api/auth/logout` - Logout dell'utente.

ticketRoute.js

- **POST** `/api/ticket/purchase` - Acquisto di un biglietto.
- **POST** `/api/ticket/cancel/:ticketId` - Cancellazione di un biglietto.
- **POST** `/api/ticket/checkin/:ticketId` - Check-in di un biglietto.

flightRoute.js

- **POST** `/api/flight/flights` - Aggiunta di un nuovo volo (solo per admin).
- **GET** `/api/flight/flights/:flightNumber` - Lettura di un volo specifico.
- **PUT** `/api/flight/flights/:id` - Aggiornamento di un volo (solo per admin).
- **DELETE** `/api/flight/flights/:id` - Cancellazione di un volo (solo per admin).
- **GET** `/api/flight/search` - Ricerca dei voli con validazione, paginazione e filtri.
- **GET** `/api/flight/flights` - Lettura di tutti i voli con paginazione e filtri.

historyRoute.js

- **GET** `/api/history/read` - Lettura dello storico delle operazioni dell'utente autenticato.

userRoleRoute.js

- **GET** `/api/user/users` - Recupero di tutti gli utenti.
- **PUT** `/api/user/users/:id/role` - Aggiornamento del ruolo di un utente.

3.4. Sicurezza e gestione dei ruoli

L’applicazione include un sistema di **autenticazione e autorizzazione** basato su **JWT (JSON Web Token)** per proteggere le API. Gli utenti sono classificati in due ruoli:

- **Utente normale (User)**: può prenotare e cancellare biglietti, visualizzare lo storico.
- **Admin**: può gestire voli e assegnare ruoli.

L'accesso a determinate API è regolato tramite **middleware di autorizzazione** (`authMiddleware.js`).

4. Panoramica codice sorgente della parte backend

4.1. app.js

Questo script è il cuore del backend del progetto *Air Connect* e ha il compito di gestire tutte le operazioni legate agli utenti, ai voli e ai biglietti, garantendo al tempo stesso sicurezza e affidabilità.

Il codice inizia importando i moduli necessari:

```
const express = require('express');
const helmet = require('helmet');
const session = require('express-session');
const https = require('https');
const fs = require('fs');
const path = require('path');
const cors = require('cors');
const sequelize = require('./config/index');
const authRoutes = require('./routes/authRoute');
const ticketRoutes = require('./routes/ticketRoute');
const flightRoutes = require('./routes/flightRoute');
const historyRoutes = require('./routes/historyRoute');
const userRoleRoutes = require('./routes/userRoleRoute');
const app = express();
```

Express viene utilizzato per creare il server, Helmet per proteggerlo da attacchi informatici, mentre express-session gestisce le sessioni utente. Inoltre, i moduli fs e path permettono di caricare il certificato SSL per abilitare la connessione sicura HTTPS. Infine, vengono importati i file che definiscono le varie API del sistema, suddivise in base alla loro funzione, come l'autenticazione e la gestione dei voli.

Dopo l'importazione, il codice configura alcune misure di sicurezza. Il middleware Helmet viene attivato con questa linea di codice:

```
app.use(helmet());
```

Subito dopo, viene definita la gestione delle richieste da altre origini con CORS. Il server accetta connessioni solo da alcuni indirizzi localhost specifici, evitando così richieste non autorizzate da altri domini:

```
const allowedOrigins = ['https://localhost:8081',
                        'https://localhost:8082',
                        'https://localhost:8083',
                        'https://localhost:8084',
                        'https://localhost:8085',
                        'https://localhost:8086',
                        'https://localhost:8087',
                        'https://localhost:8088',
```

```
                    'https://localhost:8089',
                    'https://localhost:8090' ];

app.use(cors({
    origin: (origin, callback) => {
        if (!origin || allowedOrigins.includes(origin)) {
            callback(null, true);
        } else {
            callback(new Error('Non consentito da CORS'));
        }
    },
    credentials: true
}));
```

Successivamente, viene configurata la gestione delle sessioni. Il server utilizza i cookie per identificare gli utenti e mantenere la loro sessione attiva. Viene specificato un **segreto di sessione**, e si impostano alcune regole per garantire maggiore sicurezza:

```
app.use(session({
    secret: 'il_tuo_segreto_di_sessione',
    resave: false,
    saveUninitialized: false,
    cookie: {
        httpOnly: false,
        secure: false,
        sameSite: 'lax'
    }
}));
```

Il server è ora pronto a ricevere e interpretare le richieste degli utenti, grazie a questi due middleware:

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

A questo punto, per verificare che tutto funzioni correttamente, viene definito un semplice endpoint di test, che risponde con un messaggio quando qualcuno accede alla root del server:

```
app.get('/', (req, res) => {
    res.send('Server Express HTTPS è in esecuzione!');
});
```

Dopo la configurazione di base, il server deve connettersi al database. Utilizza Sequelize per verificare che la connessione sia attiva:

```
sequelize.authenticate()
    .then(() => {
```

```
        console.log('Connessione al database avvenuta con successo.');
    })
    .catch(err => {
        console.error('Errore di connessione al database:', err);
    });
```

Una volta stabilita la connessione, il codice assicura che le tabelle del database siano correttamente sincronizzate con il backend:

```
const syncModels = async () => {
    try {
        await sequelize.sync();
        console.log('Modelli sincronizzati con successo.');
    } catch (err) {
        console.error('Errore nella sincronizzazione del database:',
err);
    }
};

syncModels();
```

Ora che il database è pronto, il backend attiva le API che gestiscono le varie operazioni dell'applicazione. Ogni gruppo di funzionalità è organizzato in un file separato e viene collegato al server in questo modo:

```
app.use('/api/auth', authRoutes);
app.use('/api/ticket', ticketRoutes);
app.use('/api/flight', flightRoutes);
app.use('/api/history', historyRoutes);
app.use('/api/user', userRoleRoutes);
```

Infine, il server viene avviato in modalità HTTPS, utilizzando un certificato SSL per garantire che tutte le comunicazioni tra client e server siano crittografate. Per farlo, vengono caricati il file della chiave privata e il certificato, e il server viene avviato sulla porta 3000:

```
const options = {
    key: fs.readFileSync(path.join(__dirname, 'localhost.key')),
    cert: fs.readFileSync(path.join(__dirname, 'localhost.crt'))
};

https.createServer(options, app).listen(3000, () => {
    console.log('Server HTTPS in ascolto sulla porta 3000');
});
```

Con questa configurazione, il backend è pronto a ricevere richieste dai client e a fornire le risposte necessarie per gestire il check-in, l'acquisto e la cancellazione dei biglietti, oltre alla gestione amministrativa dei voli e dei ruoli utente.

4.2. index.js

Questo script serve per configurare e inizializzare una connessione al database utilizzando Sequelize con SQLite come database. Sequelize è un ORM (Object-Relational Mapping) che facilita l'interazione con il database tramite oggetti JavaScript.

Lo script inizia con l'importazione dei moduli necessari:

```
const { Sequelize } = require('sequelize');
const path = require('path');
```

Il modulo `Sequelize` viene estratto dalla libreria `sequelize` per gestire la connessione e le operazioni sul database. Inoltre, il modulo `path` è utilizzato per risolvere correttamente i percorsi dei file nel filesystem, in particolare per trovare il file del database SQLite.

Successivamente, viene creata una nuova istanza di Sequelize, configurandola per usare il database SQLite:

```
const sequelize = new Sequelize({
  dialect: 'sqlite',
  storage: path.join(__dirname, '../database.sqlite'),
  logging: false
});
```

La configurazione include il tipo di database, specificando che si utilizza SQLite con la proprietà `dialect: 'sqlite'`. Il percorso del file del database viene risolto utilizzando `path.join(__dirname, '../database.sqlite')`, dove `__dirname` rappresenta la cartella corrente dello script, e `../database.sqlite` indica che il file del database si trova nella directory superiore rispetto a quella dello script. La proprietà `logging: false` è impostata per disabilitare la stampa dei log SQL durante le operazioni di Sequelize.

Infine, l'istanza di Sequelize viene esportata per poter essere utilizzata altrove nel progetto:

```
module.exports = sequelize;
```

In questo modo, il modulo `sequelize` è disponibile in altre parti dell'applicazione per interagire con il database SQLite configurato.

4.3. authMiddleware.js

Questo script definisce un middleware per controllare i ruoli degli utenti e verificare se hanno i permessi necessari per accedere a determinate risorse.

Lo script inizia importando il modello `User`, che rappresenta la tabella degli utenti nel database:

```
const User = require('../models/userModel');
```

Successivamente, viene dichiarata la funzione `checkRole`, che accetta un parametro `requiredRole`, ovvero il ruolo richiesto per accedere alla risorsa:

```
const checkRole = (requiredRole) => {
    return (req, res, next) => {
```

Questa funzione restituisce un middleware che viene eseguito per ogni richiesta HTTPS. La prima cosa che fa è verificare se l'utente è autenticato controllando se nella sessione esiste un `userId`:

```
if (!req.session || !req.session.userId) {
    return res.status(401).json({ message: 'Utente non autenticato.' });
}
```

Se l'utente non è autenticato, viene restituito un errore con codice 401 e il middleware si interrompe.

A questo punto, viene effettuata una ricerca nel database per trovare l'utente corrispondente all'`userId` memorizzato nella sessione:

```
User.findOne({ where: { id: req.session.userId } })
    .then(user => {
```

Se l'utente non esiste, viene restituito un errore con codice 404:

```
if (!user) {
  return res.status(404).json({ message: 'Utente non trovato.' });
}
```

Se invece l'utente esiste, viene controllato il suo ruolo confrontandolo con `requiredRole`:

```
if (user.role !== requiredRole) {
  return res.status(403).json({
  message: 'Accesso negato.
  Non hai i permessi necessari.' });
}
```

Se il ruolo non corrisponde a quello richiesto, viene restituito un errore 403, impedendo l'accesso alla risorsa.

Se tutti i controlli vengono superati, la funzione `next()` viene chiamata per consentire alla richiesta di proseguire

Infine, se durante l'esecuzione si verifica un errore del server, questo viene gestito e restituito con codice 500.

Alla fine dello script, la funzione `checkRole` viene esportata per poter essere utilizzata in altre parti del progetto:

```
module.exports = {
    checkRole,
};
```

Questo middleware è utile per proteggere le API, assicurandosi che solo gli utenti con determinati ruoli possano accedere a specifiche funzionalità del sistema.

4.4. flightModel.js, historyModel.js, ticketModel.js, userModel.js

Questi modelli definiscono la struttura del database per il progetto *Air Connect*, organizzando le informazioni essenziali per la gestione degli utenti, dei voli, dei biglietti e dello storico delle operazioni.

- **User**: rappresenta gli utenti registrati nel sistema e memorizza dati come:
    - `email`: indirizzo email dell'utente, univoco per ogni account.
    - `password`: la password dell'utente, memorizzata in modo sicuro.
    - `name` e `surname`: nome e cognome dell'utente.
    - `role`: specifica se l'utente è un normale cliente (`user`) o un amministratore (`admin`), determinando i suoi permessi all'interno dell'applicazione.

- **Flight**: descrive i voli disponibili e contiene le seguenti informazioni:
    - `flightNumber`: numero identificativo univoco del volo.
    - `departureTime` e `arrivalTime`: orari di partenza e arrivo.
    - `destination`: destinazione finale del volo.
    - `availableSeats`: numero di posti disponibili per il volo, aggiornato dinamicamente.

- **Ticket**: rappresenta i biglietti acquistati dagli utenti e include:
    - `flightNumber`: il numero del volo a cui il biglietto è associato.
    - `destination`: la destinazione del volo.
    - `date`: la data del volo per cui il biglietto è valido.
    - `status`: lo stato del biglietto, che di default è impostato su "active".
    - `checkinDone`: indica se l'utente ha effettuato il check-in per il volo.

- **History**: tiene traccia di tutte le operazioni effettuate dagli utenti e registra:
    - `userId`: identificativo dell'utente che ha effettuato l'operazione.
    - `operation`: il tipo di operazione eseguita (acquisto, cancellazione o check-in).
    - `ticketId`: il biglietto a cui si riferisce l'operazione.
    - `flightNumber`: il numero del volo coinvolto.
    - `departureTime` e `destination`: informazioni sul volo per contestualizzare l'operazione.
    - `timestamp`: la data e l'ora in cui l'operazione è stata eseguita.
    - `flightStatus`: lo stato attuale del volo (attivo, cancellato o modificato).
    - `seatNumber`: il numero del posto assegnato, se disponibile.

4.5. authRoute.js

Questo script definisce le rotte per la gestione dell'autenticazione e del profilo utente nel backend di *Air Connect*, utilizzando `express` per creare le API, `bcryptjs` per la gestione delle password crittografate e `express-validator` per la validazione degli input.

All'inizio, vengono importati i moduli necessari e il modello `User`, che rappresenta gli utenti del sistema:

```
const express = require('express');
const bcrypt = require('bcryptjs');
const { check, validationResult } = require('express-validator');
const User = require('../models/userModel'); // Assicurati che il percorso sia
corretto
const router = express.Router();
```

La prima rotta gestisce la registrazione degli utenti. Utilizza `express-validator` per verificare che l'email sia valida, che la password abbia almeno 6 caratteri e che nome e cognome siano presenti:

```
router.post('/register', [
    check('email').isEmail().withMessage('Inserisci un indirizzo email
valido.'),
    check('password').isLength({ min: 6 }).withMessage('La password deve
                                        contenere almeno 6 caratteri.'),
    check('name').notEmpty().withMessage('Il nome è obbligatorio.'),
    check('surname').notEmpty().withMessage('Il cognome è obbligatorio.'),
], async (req, res) => {
```

Se ci sono errori di validazione, la richiesta viene bloccata e viene restituito un messaggio di errore:

```
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
```

Successivamente, si controlla se l'email è già registrata nel database. Se esiste già un utente con la stessa email, viene restituito un errore:

```
    const existingUser = await User.findOne({ where: { email } });
    if (existingUser) {
        return res.status(400).json({ errors: [{ msg: 'Email già in uso.' }] });
    }
```

Se l'email è disponibile, la password viene crittografata con `bcryptjs` prima di essere salvata nel database:

```
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = await User.create({ email, password: hashedPassword,  name,
surname, role });
```

Alla fine della registrazione, l'utente viene creato con successo e viene restituito un messaggio di conferma:

```
    res.status(201).json({ message: 'Utente registrato con successo.', userId: newUser.id });
```

La rotta di login verifica l'esistenza dell'utente con l'email fornita:

```
router.post('/login', async (req, res) => {
    const { email, password } = req.body;

    try {
        const user = await User.findOne({ where: { email } });

        if (!user) {
            return res.status(400).json({ message: 'Email o password non corretti.' });
        }
```

Se l'utente esiste, viene confrontata la password inserita con quella salvata nel database. Se non corrispondono, il login fallisce:

```
    const isMatch = await bcrypt.compare(password, user.password);

    if (!isMatch) {
      return res.status(400).json({ message: 'Email o password non corretti.'});
    }
```

Se il login è corretto, i dati essenziali dell'utente vengono salvati nella sessione e viene restituito un messaggio di conferma:

```
        req.session.userId = user.id;
        req.session.userEmail = user.email;
        req.session.userPassword = user.password;
        res.status(200).json({ message: 'Login riuscito.' });
```

La rotta per la visualizzazione del profilo controlla se l'utente è autenticato. Se non è presente una sessione attiva, viene restituito un errore:

```
router.get('/profile', async (req, res) => {
    if (!req.session.userId) {
        return res.status(401).json({ message: 'Utente non autenticato.' });
    }
```

Se l'utente è autenticato, i suoi dati vengono recuperati dal database e inviati come risposta:

```
    try {
        const user = await User.findByPk(req.session.userId, {
            attributes: ['email', 'name', 'surname', 'role']
        });
        if (!user) {
```

```
            return res.status(404).json({ message: 'Utente non trovato.' });
        }
        res.status(200).json({ user });
```

La rotta per la modifica del profilo permette agli utenti di aggiornare le proprie informazioni. Se non esiste una sessione attiva, viene restituito un errore di autenticazione:

```
router.put('/profile/edit', async (req, res) => {
    const { name, surname, email, password, newPassword } = req.body;

    if (!req.session.userId) {
        return res.status(401).json({ message: 'Utente non autenticato.' });
    }
```

Se l'utente vuole cambiare l'email, il sistema verifica che la nuova email non sia già utilizzata da un altro utente:

```
        if (email && email !== user.email) {
            const existingEmail = await User.findOne({ where: { email } });
            if (existingEmail) {
                return res.status(400).json({ message: 'Email già in uso.' });
            }
            user.email = email;
        }
```

Se invece vuole cambiare la password, viene prima verificata la password attuale, e solo se corrisponde la nuova password viene crittografata e aggiornata:

```
        if (password && newPassword) {
            const isMatch = await bcrypt.compare(password, user.password);
            if (!isMatch) {
                return res.status(400).json({ message: 'La password attuale non
                                                              è corretta.' });
            }
            const hashedPassword = await bcrypt.hash(newPassword, 10);
            user.password = hashedPassword;
        }
```

Dopo aver aggiornato le informazioni dell'utente, i dati vengono salvati nel database e restituito un messaggio di conferma:

```
         await user.save();

       res.status(200).json({ message: 'Profilo aggiornato con successo.', user });
```

Infine, la rotta per il logout distrugge la sessione dell'utente. Se la sessione esiste, viene eliminata e il sistema conferma l'operazione:

```
router.post('/logout', (req, res) => {
    if (req.session) {
        req.session.destroy((err) => {
            if (err) {
```

```
            return res.status(500).json({ message: 'Errore durante il logout.' });
              } else {
                  return res.status(200).json({ message: 'Logout riuscito.' });
              }
          });
```

Se invece non esiste alcuna sessione attiva, il sistema restituisce un messaggio di errore:

```
    } else {
        return res.status(400).json({ message: 'Nessuna sessione attiva.' });
    }
```

4.6. flightRoute.js

Questo script gestisce le operazioni relative ai voli nel backend di *Air Connect*, utilizzando `express` per definire le API, `sequelize` per la gestione del database e `express-validator` per la validazione dei dati in input.

Si parte importando i moduli necessari, tra cui `Flight` e `History`, che rappresentano rispettivamente i voli e lo storico delle operazioni. Inoltre, viene importata la funzione `checkRole` per verificare se l'utente ha il ruolo di amministratore prima di eseguire determinate operazioni:

```
const express = require('express');
const router = express.Router();
const Flight = require('../models/flightModel');
const History = require('../models/historyModel');
const { check, validationResult } = require('express-validator');
const { checkRole } = require('../middlewares/authMiddleware');
const { Op } = require('sequelize');
```

La prima rotta consente agli amministratori di creare un nuovo volo. Prima di procedere, `checkRole('admin')` verifica se l'utente ha i permessi richiesti. Se l'utente è autorizzato, i dati del volo vengono estratti dal `body` della richiesta e salvati nel database:

```
router.post('/flights', checkRole('admin'), async (req, res) => {
    const { flightNumber, departureTime, arrivalTime, destination,
availableSeats } = req.body;

    try {
      const newFlight = await Flight.create({ flightNumber, departureTime,
arrivalTime, destination, availableSeats });
      res.status(201).json({ message: 'Volo creato con successo.', flight:
newFlight });
    } catch (error) {
      res.status(500).json({ message: 'Errore durante la creazione del volo.',
error });
    }
});
```

La seconda rotta permette di recuperare le informazioni di un volo specifico utilizzando il numero di volo come parametro. Se il volo non viene trovato, viene restituito un errore con codice 404:

```
router.get('/flights/:flightNumber', async (req, res) => {
    const { flightNumber } = req.params;

    try {
      const flight = await Flight.findOne({
        where: { flightNumber }
      });
      if (!flight) {
        return res.status(404).json({ message: 'Volo non trovato.' });
```

```
      }
      res.status(200).json({ flight });
    } catch (error) {
      res.status(500).json({ message: 'Errore durante il recupero del volo.',
error });
    }
});
```

La terza rotta consente di aggiornare le informazioni di un volo, ma solo agli amministratori. Dopo aver recuperato il volo dal database tramite il suo ID, i nuovi dati vengono assegnati ai campi corrispondenti e salvati:

```
router.put('/flights/:id', checkRole('admin'), async (req, res) => {
    const { id } = req.params;
    const { flightNumber, departureTime, arrivalTime, destination,
availableSeats } = req.body;

    try {
      const flight = await Flight.findByPk(id);
      if (!flight) {
        return res.status(404).json({ message: 'Volo non trovato.' });
      }
      flight.flightNumber = flightNumber;
      flight.departureTime = departureTime;
      flight.arrivalTime = arrivalTime;
      flight.destination = destination;
      flight.availableSeats = availableSeats;

      await flight.save();
      res.status(200).json({ message: 'Volo aggiornato con successo.',
                                                              flight });
    } catch (error) {
      res.status(500).json({ message: 'Errore durante l\'aggiornamento del
                                                            volo.', error });
    }
});
```

La quarta rotta gestisce l'eliminazione di un volo. Prima di cancellare il volo dal database, aggiorna il modello `History` per segnalare che lo stato del volo è stato cambiato in "cancellato". Infine, il volo viene eliminato dal database:

```
router.delete('/flights/:id', checkRole('admin'), async (req, res) => {
  const { id } = req.params;

  try {
    const flight = await Flight.findByPk(id);
    if (!flight) {
      return res.status(404).json({ message: 'Volo non trovato.' });
    }

    await History.update(
      { flightStatus: 'cancellato' },
```

```
      { where: { flightNumber: flight.flightNumber } }
    );

    await flight.destroy();

    res.status(200).json({ message: 'Volo cancellato con successo.' });
  } catch (error) {
    res.status(500).json({ message: 'Errore durante la cancellazione del volo.',
error });
  }
});
```

La quinta rotta permette di cercare i voli con filtri avanzati, paginazione e validazione dei parametri di input. Viene usato `express-validator` per garantire che i dati siano nel formato corretto:

```
router.get('/search', [
  check('flightNumber').optional().isString().withMessage('Il numero del volo
                                                    deve essere una stringa.'),
  check('destination').optional().isString().withMessage('La destinazione deve
                                                        essere una stringa.'),
  check('departureTime').optional().isISO8601().toDate().withMessage('La data di
             partenza deve essere in formato ISO 8601 (YYYY-MM-DDTHH:mm:ss).'),
  check('availableSeats').optional().isInt({ min: 0 }).withMessage('Il numero di
                    posti disponibili deve essere un numero intero positivo.'),
  check('page').optional().isInt({ min: 1 }).withMessage('La pagina deve essere
                                                    un numero intero positivo.'),
  check('limit').optional().isInt({ min: 1 }).withMessage('Il limite deve essere
                                                     un numero intero positivo.')
], async (req, res) => {
```

Se la validazione fallisce, viene restituito un errore con codice 400. Altrimenti, vengono applicati i filtri sui campi specificati:

```
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
  }

  const { flightNumber, destination, departureTime, availableSeats, page = 1,
limit = 10 } = req.query;
  try {
      let searchCriteria = {};

      if (flightNumber) {
          searchCriteria.flightNumber = flightNumber;
      }

      if (destination) {
          searchCriteria.destination = {
              [Op.like]: `${destination}%`
          };
```

```
    }

    if (departureTime) {
        searchCriteria.departureTime = {
            [Op.gte]: new Date(departureTime),
        };
    }

    if (availableSeats) {
        searchCriteria.availableSeats = {
            [Op.gte]: availableSeats,
        };
    }
```

La ricerca viene eseguita con `findAndCountAll()`, che permette di ottenere sia il numero totale di voli trovati che i dettagli di quelli che rientrano nei criteri specificati:

```
     const offset = (page - 1) * limit;

    const { count, rows: flights } = await Flight.findAndCountAll({
        where: searchCriteria,
        offset: offset,
        limit: parseInt(limit),
    });

    if (flights.length === 0) {
        return res.status(404).json({ message: 'Nessun volo trovato con i
                                    criteri di ricerca specificati.' });
    }

    res.status(200).json({
        flights,
        currentPage: parseInt(page),
        totalPages: Math.ceil(count / limit),
        totalFlights: count,
    });
  } catch (error) {
    res.status(500).json({ message: 'Errore durante la ricerca dei voli.',
                                                                error });
  }
});
```

L'ultima rotta elenca tutti i voli disponibili con paginazione e filtri, simile alla ricerca avanzata, ma senza validazione dei parametri:

```
router.get('/flights', async (req, res) => {
  const { destination, departureTime, availableSeats, page = 1, limit = 10 } =
req.query;
```

4.7. historyRoute.js

Questo script definisce un'API per recuperare lo storico delle operazioni effettuate dagli utenti nel sistema *Air Connect*. Utilizza `express` per gestire le richieste HTTPS e `sequelize` per interagire con il database, in particolare con il modello `History`, che registra eventi come acquisti, cancellazioni e check-in.

Si parte importando i moduli necessari e creando un router di Express per gestire le richieste:

```
const express = require('express');
const router = express.Router();
const { Op } = require('sequelize');
const History = require('../models/historyModel');
```

La rotta definita gestisce la lettura dello storico delle operazioni di un utente. Per prima cosa, verifica se l'utente è autenticato controllando la presenza del suo `userId` nella sessione. Se l'utente non è autenticato, viene restituito un errore con codice 401:

```
router.get('/read', async (req, res) => {
    if (!req.session.userId) {
        return res.status(401).json({ message: 'Utente non autenticato.' });
    }
```

Successivamente, vengono estratti i parametri dalla query string della richiesta, che servono per applicare eventuali filtri sui risultati. I parametri includono `operation` (tipo di operazione, es. acquisto o cancellazione), `departureTime` (data di partenza), `destination` (destinazione del volo), `page` e `limit` per gestire la paginazione:

```
    try {
        const { operation, departureTime, destination, page = 1, limit = 10 } =
req.query;
```

Viene quindi costruito un oggetto `filters` per specificare le condizioni di ricerca. Il primo filtro assicura che vengano mostrate solo le operazioni dell'utente attualmente autenticato:

```
        const filters = { userId: req.session.userId };
```

Se l'utente ha specificato un tipo di operazione, questo viene aggiunto ai filtri:

```
        if (operation) {
            filters.operation = operation;
        }
```

Se è stata fornita una data di partenza, il sistema cerca solo le operazioni con una `departureTime` uguale o successiva a quella specificata, utilizzando l'operatore `Op.gte` (greater than or equal):

```
if (departureTime) {
    filters.departureTime = { [Op.gte]: departureTime };
}
```

Se l'utente ha specificato una destinazione, il sistema effettua una ricerca che include anche destinazioni parzialmente corrispondenti grazie all'operatore `Op.like`, che permette di trovare stringhe che contengono il valore dato:

```
if (destination) {
    filters.destination = {
        [Op.like]: `%${destination}%`,
    };
}
```

Dopo aver costruito i filtri, viene calcolato l'`offset`, che serve per implementare la paginazione. L'offset indica quanti record devono essere saltati prima di iniziare a recuperare i risultati:

```
const offset = (page - 1) * limit;
```

A questo punto, viene eseguita la query per recuperare lo storico, utilizzando `findAndCountAll()`, che permette di ottenere sia il numero totale di record trovati che un array contenente i risultati della ricerca. I risultati vengono ordinati in ordine decrescente rispetto al timestamp per mostrare prima le operazioni più recenti:

```
const { count, rows: history } = await History.findAndCountAll({
    where: filters,
    attributes: ['ticketId', 'operation', 'flightNumber',
    'departureTime', 'destination', 'timestamp', 'flightStatus',
    'seatNumber'],
    order: [['timestamp', 'DESC']],
    offset,
    limit: parseInt(limit),
});
```

Se non vengono trovate operazioni, il server restituisce un errore con codice 400 e un messaggio appropriato:

```
if (history.length === 0) {
    return res.status(400).json({ message: 'Non è stata effettuata
    ancora nessuna operazione.' });
}
```

Se invece vengono trovati dei risultati, questi vengono restituiti nel formato JSON, includendo le informazioni sulle operazioni effettuate, la pagina corrente, il numero totale di pagine e il numero totale di record trovati:

```
         res.status(200).json({
            history,
            currentPage: parseInt(page),
            totalPages: Math.ceil(count / limit),
            totalRecords: count,
        });
```

Infine, se si verifica un errore durante l'elaborazione della richiesta, viene restituito un errore con codice 500 e il messaggio di errore corrispondente:

```
    } catch (error) {
        res.status(500).json({ message: 'Errore durante il recupero dello
        storico.', error });
    }
});
```

Il router viene esportato alla fine del file, in modo che possa essere utilizzato in altre parti dell'applicazione:

```
module.exports = router;
```

4.8. ticketRoute.js

Questo script gestisce le operazioni relative ai biglietti nel sistema *Air Connect*, consentendo agli utenti di acquistare, cancellare e effettuare il check-in per un volo.

Si parte importando i moduli necessari, tra cui `Ticket`, `Flight` e `History`, che rappresentano rispettivamente i biglietti, i voli e lo storico delle operazioni. Viene anche importato `checkRole` per garantire che solo gli amministratori possano accedere a determinate funzionalità:

```
const express = require('express');
const { Op } = require('sequelize');
const router = express.Router();
const Ticket = require('../models/ticketModel');
const Flight = require('../models/flightModel');
const History = require('../models/historyModel');
const { checkRole } = require('../middlewares/authMiddleware');
const { query, validationResult } = require('express-validator');
```

La prima rotta gestisce l'acquisto di un biglietto. L'utente deve essere autenticato, quindi viene controllata la presenza del suo `userId` nella sessione. Se l'utente non è autenticato, viene restituito un errore:

```
router.post('/purchase', async (req, res) => {
    const { flightNumber } = req.body;
    const userId = req.session.userId;

    if (!userId) {
        return res.status(401).json({ message: 'Utente non autenticato.' });
    }
```

Dopo aver verificato che il volo esista e abbia posti disponibili, viene creato un nuovo biglietto e il numero di posti disponibili del volo viene ridotto di uno:

```
        const flight = await Flight.findOne({
            where: { flightNumber }
        });

        if (!flight) {
            return res.status(404).json({ message: 'Volo non trovato.' });
        }

        if (flight.availableSeats <= 0) {
            return res.status(400).json({ message: 'Non ci sono posti
            disponibili per questo volo.' });
        }

        const newTicket = await Ticket.create({
            flightNumber,
            destination: flight.destination,
            date: flight.departureTime,
            UserId: userId
```

```
});

flight.availableSeats -= 1;
await flight.save();
```

L'operazione viene registrata nello storico con il tipo `acquisto`:

```
await History.create({
    userId: req.session.userId,
    operation: 'acquisto',
    ticketId: newTicket.id,
    flightNumber: flight.flightNumber,
    departureTime: flight.departureTime,
    destination: flight.destination,
    timestamp: new Date(),
});
```

La rotta successiva gestisce la cancellazione di un biglietto acquistato. Anche qui si verifica prima se l'utente è autenticato:

```
router.post('/cancel/:ticketId', async (req, res) => {
    const { ticketId } = req.params;
    const userId = req.session.userId;

    if (!userId) {
        return res.status(401).json({ message: 'Utente non autenticato.' });
    }
```

Si cerca il biglietto nel database e si verificano alcune condizioni: se il biglietto è già stato cancellato o se il check-in è stato effettuato, l'operazione viene bloccata:

```
const ticket = await Ticket.findOne({ where: { id: ticketId, UserId:
userId } });

if (!ticket) {
    return res.status(404).json({ message: 'Biglietto non trovato.' });
}

if (ticket.status === 'cancelled') {
    return res.status(400).json({ message: 'Il biglietto è già stato
    cancellato.' });
}

if (ticket.checkinDone) {
    return res.status(400).json({ message: 'Non è possibile cancellare
    un biglietto dopo aver effettuato il check-in.' });
}
```

Il biglietto viene quindi aggiornato come `cancelled` e il posto liberato nel volo corrispondente:

```
ticket.status = 'cancelled';
ticket.timestamp = new Date();
await ticket.save();

const flight = await Flight.findOne({
    where: { flightNumber: ticket.flightNumber }
});
flight.availableSeats += 1;
await flight.save();
```

Anche questa operazione viene registrata nello storico:

```
const historyRecord = await History.findOne({
    where: { ticketId: ticket.id, userId: req.session.userId, operation:
             'acquisto' }
});

if (historyRecord) {
    historyRecord.operation = 'cancellazione';
    historyRecord.timestamp = new Date();
    await historyRecord.save();
}
```

L'ultima rotta permette agli utenti di effettuare il check-in per un biglietto acquistato. Dopo aver verificato che l'utente sia autenticato e che il biglietto esista, si controlla che non sia stato cancellato o già utilizzato per il check-in:

```
router.post('/checkin/:ticketId', async (req, res) => {
    const { ticketId } = req.params;
    const userId = req.session.userId;

    if (!userId) {
        return res.status(401).json({ message: 'Utente non autenticato.' });
    }

    try {
        const ticket = await Ticket.findOne({ where: { id: ticketId, UserId:
        userId } });

        if (!ticket) {
            return res.status(404).json({ message: 'Biglietto non trovato.' });
        }

        if (ticket.status === 'cancelled') {
            return res.status(400).json({ message: 'Il biglietto è stato
            cancellato e non può essere utilizzato per il check-in.' });
        }
```

```
if (ticket.checkinDone) {
    return res.status(400).json({ message: 'Il check-in è già stato
    effettuato.' });
}
```

Il biglietto viene aggiornato come `checkinDone = true` e si assegna un numero di posto in base al numero di check-in già registrati per quel volo:

```
 ticket.checkinDone = true;
ticket.timestamp = new Date();
await ticket.save();

const historyRecord = await History.findOne({
    where: { ticketId: ticket.id, userId: req.session.userId, operation:
             'acquisto' }
});

if (historyRecord) {
    historyRecord.operation = 'check-in';
    historyRecord.timestamp = new Date();
    historyRecord.seatNumber = await History.count({ where:
    { flightNumber: ticket.flightNumber, operation: 'check-in' } }) + 1;
    await historyRecord.save();
}
```

4.9. userRoleRoute.js

Questo script definisce le API per la gestione degli utenti nel sistema *Air Connect*, consentendo di recuperare la lista degli utenti con filtri e di aggiornare il ruolo di un utente.

Si parte importando i moduli necessari, tra cui `User`, che rappresenta il modello degli utenti, ed `Op` di Sequelize, che permette di usare operatori avanzati nelle query:

```
const express = require('express');
const router = express.Router();
const User = require('../models/userModel');
const { Op } = require('sequelize');
```

La prima rotta permette di ottenere un elenco di utenti con filtri opzionali e paginazione. I parametri della richiesta possono includere email, nome, cognome e ruolo, oltre ai parametri `page` e `limit` per la paginazione. Viene quindi inizializzato un oggetto `filters`, che conterrà le condizioni di ricerca:

```
router.get('/users', async (req, res) => {
  const { email, name, surname, role, page = 1, limit = 10 } = req.query;

  try {
    const filters = {};
```

Se l'utente specifica un filtro per l'email, il sistema cerca gli utenti il cui indirizzo email contiene la stringa fornita, grazie all'operatore `Op.like`, che permette di effettuare ricerche parziali:

```
    if (email) filters.email = { [Op.like]: `%${email}%` };
```

Lo stesso metodo viene applicato per il nome e il cognome, mentre per il ruolo viene fatto un controllo diretto, dato che può avere solo valori predefiniti (`user` o `admin`):

```
    if (name) filters.name = { [Op.like]: `%${name}%` };
    if (surname) filters.surname = { [Op.like]: `%${surname}%` };
    if (role) filters.role = role;
```

Per gestire la paginazione, viene calcolato l'`offset`, che rappresenta il numero di record da saltare prima di iniziare a restituire i risultati:

```
    const offset = (page - 1) * limit;
```

Viene quindi eseguita una query con `findAndCountAll()`, che permette di recuperare sia la lista degli utenti che il numero totale di risultati trovati. La paginazione viene applicata tramite `offset` e `limit`:

```
    const { count, rows: users } = await User.findAndCountAll({
      where: filters,
      offset,
      limit: parseInt(limit),
    });
```

I risultati vengono restituiti come un oggetto JSON contenente l'elenco degli utenti trovati, il numero totale di pagine, la pagina corrente e il numero complessivo di utenti:

```
    res.json({
      users,
      totalPages: Math.ceil(count / limit),
      currentPage: parseInt(page),
      totalUsers: count,
    });
```

Se si verifica un errore durante l'esecuzione della query, viene restituito un messaggio di errore con codice 500:

```
  } catch (error) {
    res.status(500).json({ message: 'Errore durante il recupero degli
utenti.' });
  }
});
```

La seconda rotta permette di aggiornare il ruolo di un utente. Il sistema riceve l'`id` dell'utente come parametro nell'URL e il nuovo ruolo nel corpo della richiesta:

```
router.put('/users/:id/role', async (req, res) => {
  const { id } = req.params;
  const { role } = req.body;
```

Viene cercato l'utente nel database in base al suo ID con `findByPk()`. Se l'utente esiste, il suo ruolo viene aggiornato e i dati vengono salvati nel database:

```
  try {
    const user = await User.findByPk(id);
    if (user) {
      user.role = role;
      await user.save();
      res.json({ message: 'Ruolo aggiornato con successo.' });
```

Se l'utente non viene trovato, viene restituito un errore con codice 404:

```
    } else {
      res.status(404).json({ message: 'Utente non trovato.' });
    }
```

Se si verifica un errore durante l'aggiornamento del ruolo, il server restituisce un errore 500:

```
  } catch (error) {
    res.status(500).json({ message: 'Errore durante l\'aggiornamento del ruolo.'
});
  }
});
```

Infine, il router viene esportato per essere utilizzato in altre parti dell'applicazione:

```
module.exports = router;
```

5. Panoramica codice sorgente della parte frontend

Nela parte frontend, i servizi gestiscono la comunicazione con il backend tramite richieste HTTPS, utilizzando la libreria **Axios** per inviare dati e processare le risposte. Ogni servizio è dedicato a un'area specifica dell'applicazione, come l'autenticazione, la gestione dei voli, degli utenti e dei biglietti.

Le pagine (componenti page) in un'applicazione React rappresentano le diverse viste o schermate che l'utente può vedere e interagire. Ogni pagina è un componente React che gestisce una specifica funzionalità o insieme di funzionalità. Le pagine sono spesso collegate tramite un router (come `react-router-dom`), che permette la navigazione tra le diverse viste dell'applicazione.

Le pagine possono contenere moduli, tabelle, filtri e altre interfacce utente che permettono agli utenti di interagire con l'applicazione. Ogni pagina può fare uso di servizi per comunicare con il backend e gestire lo stato locale tramite gli hook di React come `useState` e `useEffect`. Le pagine sono essenzialmente viste senza logica di business, focalizzandosi sulla presentazione grafica per l'utente finale, mentre la gestione della logica è delegata ai componenti e ai servizi, per cui non verrà fornita una descrizione dettagliata delle singole pagine in questa documentazione.

5.1. userService.js

Il servizio `authService` gestisce tutte le operazioni relative all'autenticazione e al profilo utente.

Il servizio inizia importando `axios` e definendo una costante `API_URL` che contiene l'URL di base per le richieste di autenticazione:

```
import axios from 'axios';

const API_URL = 'https://localhost:3000/api/auth/';
```

La funzione `login` invia una richiesta POST all'endpoint `/login` con le credenziali dell'utente. Se la risposta contiene un token, questo viene salvato nel `localStorage`. La funzione restituisce i dati della risposta:

```
const login = async (email, password) => {
  const response = await axios.post(API_URL + 'login', { email, password },
{ withCredentials: true });
  if (response.data.token) {
    localStorage.setItem('auth_token', response.data.token);
  }
  return response.data;
};
```

La funzione `register` invia una richiesta POST all'endpoint `/register` con i dati di registrazione dell'utente e restituisce i dati della risposta:

```
const register = async (email, password, name, surname) => {
  const response = await axios.post(API_URL + 'register', { email, password,
name, surname }, { withCredentials: true });
  return response.data;
};
```

La funzione `logout` invia una richiesta POST all'endpoint `/logout` per disconnettere l'utente e rimuove il token di autenticazione dal `localStorage`:

```
const logout = async () => {
  await axios.post(API_URL + 'logout', {}, { withCredentials: true });
  localStorage.removeItem('auth_token');
};
```

La funzione `getProfile` invia una richiesta GET all'endpoint `/profile` per ottenere i dati del profilo dell'utente autenticato e restituisce i dati della risposta:

```
const getProfile = async () => {
  const response = await axios.get(API_URL + 'profile', { withCredentials:
true });
  return response.data;
};
```

La funzione `updateProfile` invia una richiesta PUT all'endpoint `/profile/edit` con i dati aggiornati del profilo dell'utente e restituisce i dati della risposta:

```
const updateProfile = async (profileData) => {
  const response = await axios.put(API_URL + 'profile/edit', profileData,
{ withCredentials: true });
  return response.data;
};
```

Infine, tutte le funzioni vengono esportate come parte di un oggetto `authService`:

```
const authService = {
  login,
  register,
  logout,
  getProfile,
  updateProfile,
};

export default authService;
```

5.2. userService.js

Il servizio `userService` gestisce le operazioni relative agli utenti, come ottenere la lista degli utenti e aggiornare il ruolo di un utente.

Il servizio inizia importando `axios` e definendo una costante `API_URL` che contiene l'URL di base per le richieste relative agli utenti:

```
import axios from 'axios';

const API_URL = 'https://localhost:3000/api/user';
```

La funzione `getUsers` invia una richiesta GET all'endpoint users con eventuali filtri, il numero di pagina e il limite di risultati per pagina. I parametri vengono passati come query string. La funzione restituisce i dati della risposta, che contengono la lista degli utenti:

```
const getUsers = async (filters = {}, page = 1, limit = 10) => {
  const params = { ...filters, page, limit };
  const response = await axios.get(`${API_URL}/users`, { params });
  return response.data;
};
```

La funzione `updateUserRole` invia una richiesta PUT all'endpoint `/users/{userId}/role` con il nuovo ruolo dell'utente. La funzione non restituisce alcun dato:

```
const updateUserRole = async (userId, newRole) => {
  await axios.put(`${API_URL}/users/${userId}/role`, { role: newRole });
};
```

Infine, le funzioni `getUsers` e `updateUserRole` vengono esportate come parte di un oggetto:

```
export default {
  getUsers,
  updateUserRole,
};
```

5.3. flightAdminService.js

Il servizio `flightAdminService` gestisce le operazioni amministrative relative ai voli, come ottenere la lista dei voli, aggiungere un nuovo volo, aggiornare un volo esistente e cancellare un volo.

Il servizio inizia importando `axios` e definendo una costante `API_URL` che contiene l'URL di base per le richieste relative ai voli:

```
import axios from 'axios';

const API_URL = 'https://localhost:3000/api/flight/flights';
```

La funzione `getFlights` invia una richiesta GET all'endpoint `/flights` con eventuali filtri, il numero di pagina e il limite di risultati per pagina. I parametri vengono passati come query string. La funzione restituisce i dati della risposta, che contengono la lista dei voli:

```
const getFlights = async (filters = {}, page = 1, limit = 10) => {
  const params = { ...filters, page, limit };
  const response = await axios.get(API_URL, { params, withCredentials: true });
  return response.data;
};
```

La funzione `addFlight` invia una richiesta POST all'endpoint `/flights` con i dati del nuovo volo. La funzione restituisce i dati della risposta, che contengono le informazioni del volo aggiunto:

```
const addFlight = async (flightData) => {
  const response = await axios.post(API_URL, flightData, { withCredentials: true
});
  return response.data;
};
```

La funzione `updateFlight` invia una richiesta PUT all'endpoint `/flights/{flightId}` con i dati aggiornati del volo. La funzione restituisce i dati della risposta, che contengono le informazioni del volo aggiornato:

```
const updateFlight = async (flightId, flightData) => {
  const response = await axios.put(`${API_URL}/${flightId}`, flightData,
{ withCredentials: true });
  return response.data;
};
```

La funzione `deleteFlight` invia una richiesta DELETE all'endpoint `/flights/{flightId}` per cancellare un volo. La funzione restituisce i dati della risposta, che confermano la cancellazione del volo:

```
const deleteFlight = async (flightId) => {
  const response = await axios.delete(`${API_URL}/${flightId}`,
{ withCredentials: true });
  return response.data;
};
```

Infine, tutte le funzioni vengono esportate come parte di un oggetto `flightAdminService`:

```
const flightAdminService = {
  getFlights,
  addFlight,
  updateFlight,
  deleteFlight,
};

export default flightAdminService;
```

5.4. flightService.js

Il servizio `flightService` gestisce le operazioni relative alla ricerca dei voli.

Il servizio inizia importando `axios` e definendo una costante `API_URL` che contiene l'URL di base per le richieste relative alla ricerca dei voli:

```
import axios from 'axios';

const API_URL = 'https://localhost:3000/api/flight/search';
```

La funzione `getFlights` invia una richiesta GET all'endpoint `/search` con eventuali filtri, il numero di pagina e il limite di risultati per pagina. I parametri vengono passati come query string. La funzione restituisce i dati della risposta, che contengono la lista dei voli trovati:

```
const getFlights = async (filters = {}, page = 1, limit = 10) => {
  const params = { ...filters, page, limit };
  const response = await axios.get(API_URL, { params });
  return response.data;
};
```

Infine, la funzione `getFlights` viene esportata come parte di un oggetto `flightService`:

```
const flightService = {
  getFlights,
};

export default flightService;
```

5.5. historyService.js

Il servizio `historyService` gestisce le operazioni relative alla cronologia delle operazioni e alla gestione dei biglietti.

La funzione `fetchHistory` invia una richiesta GET all'endpoint `/history/read` con il numero di pagina e i filtri specificati. I parametri vengono passati come query string e l'autenticazione viene gestita tramite un token passato nell'header della richiesta. La funzione restituisce i dati della risposta, che contengono la cronologia delle operazioni:

```
import axios from 'axios';

const fetchHistory = async (token, page, filters) => {
  try {
    const response = await axios.get('https://localhost:3000/api/history/read',
{
      params: {
        page,
        operation: filters.operation,
        arrivalTime: filters.arrivalTime,
        destination: filters.destination
      },
      headers: {
        Authorization: `Bearer ${token}`,
      },
      withCredentials: true,
    });
    return response.data;
  } catch (error) {
    console.error('Errore durante il recupero dello storico:', error);
    throw error;
  }
};
```

La funzione `cancelTicket` invia una richiesta POST all'endpoint `/ticket/cancel/{ticketId}` per cancellare un biglietto. Se la richiesta ha successo, viene mostrato un messaggio di conferma. In caso di errore, viene mostrato un messaggio di errore e l'errore viene rilanciato:

```
const cancelTicket = async (ticketId) => {
  try {
    await axios.post(`https://localhost:3000/api/ticket/cancel/${ticketId}`, {},
{
      withCredentials: true,
    });
    alert('Biglietto cancellato con successo!');
  } catch (error) {
    console.error('Errore durante la cancellazione del biglietto:', error);
    alert('Errore durante la cancellazione del biglietto');
    throw error;
```

```
  }
};
```

La funzione `checkInTicket` invia una richiesta POST all'endpoint `/ticket/checkin/{ticketId}` per effettuare il check-in di un biglietto. Se la richiesta ha successo, viene mostrato un messaggio di conferma. In caso di errore, viene mostrato un messaggio di errore e l'errore viene rilanciato:

```
const checkInTicket = async (ticketId) => {
  try {
    await axios.post(`https://localhost:3000/api/ticket/checkin/${ticketId}`,
{}, {
      withCredentials: true,
    });
    alert('Check-in effettuato con successo!');
  } catch (error) {
    console.error('Errore durante il check-in del biglietto:', error);
    alert('Errore durante il check-in del biglietto');
    throw error;
  }
};
```

Infine, le funzioni `fetchHistory`, `cancelTicket` e `checkInTicket` vengono esportate:

```
export { fetchHistory, cancelTicket, checkInTicket };
```