A yellow letter with an arrow

AI-generated content may be incorrect.

SoTourist

Salvatore Oteri – Matteo Filizzola – Gabriele Faraone – Annalisa Curto

**Progetto Programmazione Web e Mobile (SoTourist)**

Il progetto **SoTourist** è stato sviluppato con l'obiettivo di realizzare un'applicazione multipiattaforma innovativa per la gestione personalizzata di itinerari turistici, rispondendo a una necessità concreta di semplificare l'organizzazione dei viaggi e valorizzare l'esperienza degli utenti.

In particolare, è stata posta grande attenzione all'**analisi dei requisiti**, che ha guidato la scelta delle tecnologie e dell'architettura complessiva. Il progetto prevede un **frontend mobile-first** sviluppato con **Angular, Ionic e Capacitor**, ottimizzato per dispositivi Android, e un **backend in Node.js con Express**, che gestisce le operazioni lato server e si basa su un database locale SQLite. L'applicazione integra, inoltre, i servizi **Google Maps** per la visualizzazione delle mappe e la ricerca dei luoghi, garantendo così un'interazione semplice e intuitiva.

I risultati ottenuti dimostrano la coerenza con gli obiettivi iniziali, offrendo agli utenti un'applicazione funzionale, intuitiva e flessibile per la creazione e la gestione dei propri itinerari di viaggio personalizzati.

**Indice:**

**Consigli d’uso** pag. 4

**Front-End** “ 5

Struttura cartelle “ 5

**Funzionalità offerte** “ 5

Login “ 5

Registrazione “ 7

Home “ 9

Navigazione tramite Tabs “ 11

Viaggi “ 12

Crea un itinerario “ 14

Storico Viaggi “ 17

Itinerario “ 19

Personalizzazione Itinerario “ 21

Modifica Date “ 22

Panoramica “ 25

Mappa “ 27

Settings “ 29

Profilo “ 31

Cambio credenziali “ 32

Impostazioni avanzate “ 34

Upgrade Piano Abbonamento “ 35

**Componenti** “ 37

generation-overlay “ 37

google-autocomplete “ 38

app-header “ 39

app-luogo-card “ 40

app-navigation-bar “ 41

app-profile-icon “ 42

app-range-calendar-lite “ 44

app-toggle “ 45

app-trip-card “ 46

app-unfinished-card “ 48

app-user-hero “ 50

**Guards** “ 51

GoldGuard “ 51

PremiumGuard “ 52

NoGuestGuard “ 53

**Back-end** “ 54-83

**Consigli per l’uso:**

**Browser consigliati:**

* Microsoft Edge
* Safari

Per il funzionamento delle API Google, il front-end deve girare sull’indirizzo IP di rete;  
per facilitare l’avvio del front-end e del back-end è presente, nella directory di GitHub, il file “*Avvio server.bat*” (avvia entrambi i server correttamente e apre Microsoft Edge).

Per avviare il progetto SoTourist è necessario installare alcuni strumenti di base:

Per prima cosa bisogna installare Node.js dal sito ufficiale (https://nodejs.org/), così da rendere disponibili i comandi node e npm.

Successivamente, si installano gli strumenti necessari per il frontend.

Dal terminale, **eseguire**:

npm install -g @angular/cli @ionic/cli

npm install @capacitor/core @capacitor/cli

Per la parte backend, nella cartella del backend è indispensabile installare la libreria che permette di interagire con il database SQLite:

npm install sqlite3

A questo punto, in entrambe le cartelle di frontend e backend, è necessario installare le dipendenze locali del progetto eseguendo:

npm install

**Soltanto dopo questi passaggi l’applicazione può essere eseguita correttamente sia lato frontend che backend.**

È stato inoltre creato a priori un account d’esempio con le seguenti credenziali d’accesso all’app:

* **E-mail**: [sotourist@gmail.com](mailto:sotourist@gmail.com)
* **Password**: SoTourist\_123

**Front-End**

**Struttura cartelle:**

* src/app
  + /components (cartella dei componenti)
  + /guards (cartella delle guards per proteggere alcune rotte)
  + /models (oggetti personalizzati)
  + /pages (contiene tutte le pagine)
  + /services (contiene i vari services)
  + /utils (contiene tutti gli utils)

**Funzionalità offerte**

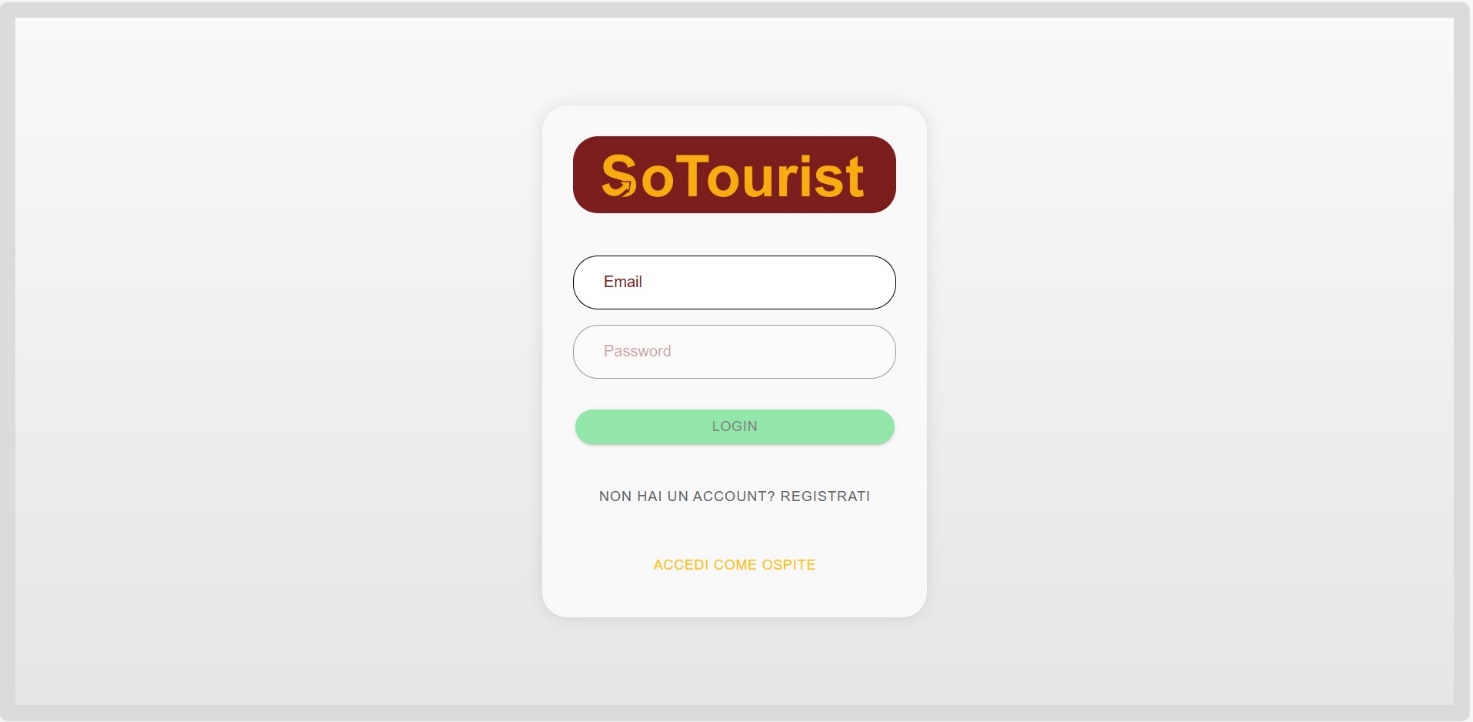
**1: Login**

Al primo accesso, all’utente viene presentata la pagina di **login**, dove può autenticarsi inserendo le proprie credenziali (e-mail e password). La password deve rispettare i seguenti criteri di validità:

* almeno 8 caratteri
* almeno una lettera maiuscola
* almeno un numero
* almeno un carattere speciale

In caso di credenziali errate, mancanti o se la password non rispetta i criteri richiesti, viene visualizzato un messaggio di errore tramite un popup e l’utente resta sulla pagina di login. Se invece le informazioni inserite sono corrette e rispettano i requisiti, l’autenticazione ha successo e l’utente viene reindirizzato automaticamente alla pagina Home.

La pagina è definita in “*src/app/pages/login”*. La struttura della pagina viene definita in *“/login.page.html”*.



In figura viene mostrato il box di login dove vengono usati componenti ionic come ion-input e ion-button. In dettaglio troviamo:

* **Logo** (esteso)
* **E-mail**: campo dove verrà inserita l’e-mail per autenticarsi
* **Password**: campo dove verrà inserita la password per autenticarsi, è attivo solo se l’e-mail è stata precedentemente inserita
* **Tasto di Login:** disabilitato fino a quando i campi E-mail e Password non riempiti
* Opzione cliccabile per la **registrazione**, visualizzata se l'utente non possiede un account
* Opzione cliccabile per **l'accesso come ospite**, che comporta funzionalità limitate rispetto all'utente registrato

Nel file “*/login.page.ts*” è implementata la logica della pagina di login, compresa l’invocazione delle API del server Express per la validazione delle credenziali inserite. Inoltre, viene gestito il reindirizzamento dell’utente alla pagina di registrazione tramite l’apposito pulsante.

Le funzionalità presenti sono:

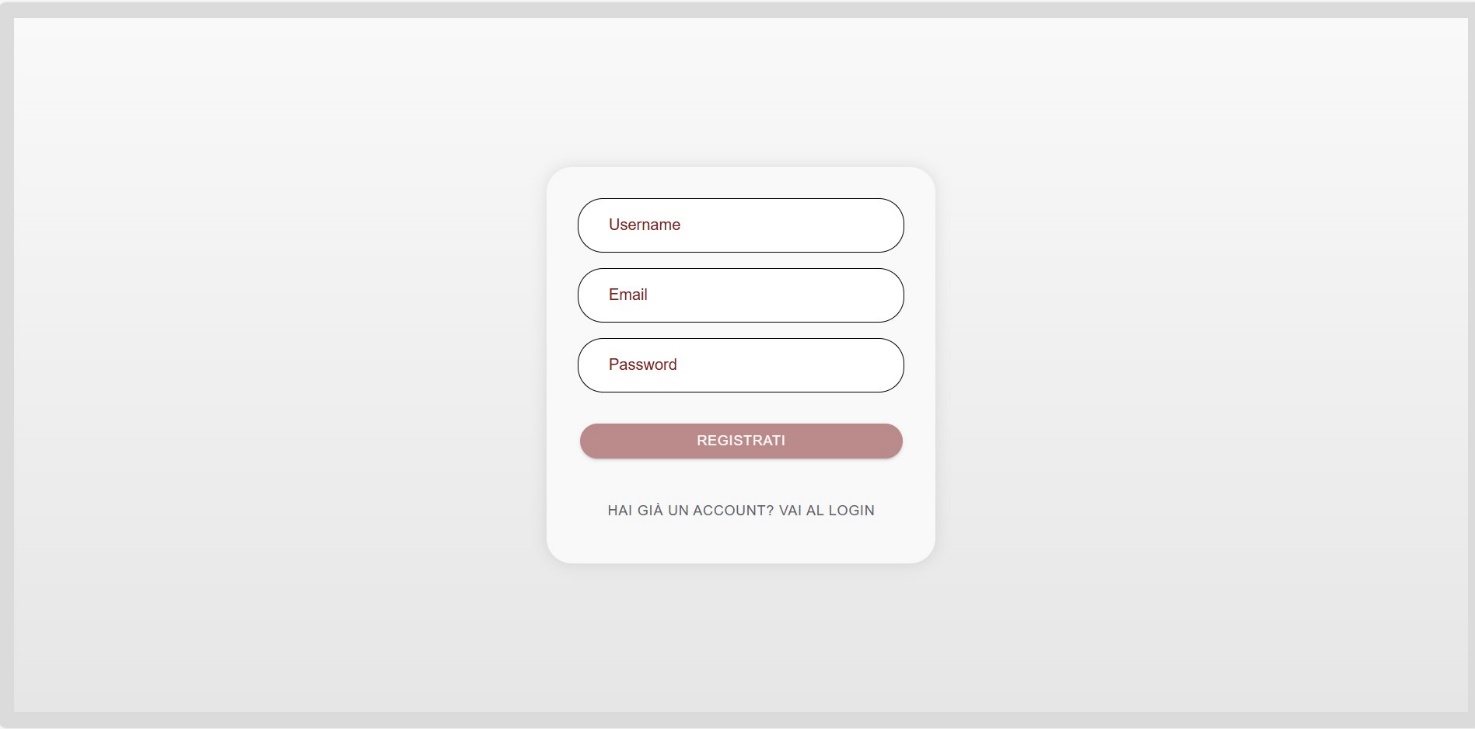
* **saveProfile**: salva in locale i dati dell’utente che serviranno per tenerne traccia nella sessione e anche in caso di rientro nell’app (login automatico se già loggato)
* **loginAsGuest**: gestisce il login nel caso si acceda come ospite
* **focusPassword:** indirizza l’utente sul campo password dopo invio
* **triggerLogin:** login rapido tramite tasto invio

**2:** **Registrazione**

L’utente può **registrarsi** all’applicazione creando un nuovo account tramite la schermata di registrazione. L’utente dovrà inserire i seguenti dati obbligatori: **Username**, **E-mail**, **Password**.

Nel caso di informazioni errate o incomplete, verrà ritornata una stringa che mostra l’errore tramite un popup e l’utente rimarrà sulla schermata di registrazione.

Nel caso di registrazione avvenuta correttamente, l’utente verrà reindirizzato automaticamente alla schermata di login, dove potrà accedere con le credenziali appena create.



L’immagine mostra la schermata di registrazione definita in “*src/app/pages/registrazione*”.

La struttura della pagina è definita nel file *“/registrazione.page.html”* e vengono utilizzati i componenti Ionic ion-input e ion-button. Gli elementi sono disposti centralmente, con stile migliorato per facilitare la compilazione. In particolare, sono presenti**:**

* **Username**
* **Email** (Deve includere):
  + Almeno un carattere
  + Carattere ‘@’
  + Almeno un carattere
  + Un punto
  + Almeno un carattere
* **Password** (richiede almeno):
  + Un carattere maiuscolo
  + Uno minuscolo
  + Una cifra
  + Un carattere speciale
  + Almeno 8 caratteri
* **Bottone “Registrati”,** che avvia il processo di registrazione
* **Bottone per tornare alla pagina di login,** nel caso l’utente abbia già un account

Nel file *“/registrazione.page.ts”* viene definita la logica della pagina, tra cui:

* La gestione dei dati inseriti nei campi con il two-way binding [(ngModel)]
* La comunicazione con il backend Express tramite il servizio AuthService, per la registrazione dell’utente
* L’utilizzo del servizio ToastService per mostrare all’utente messaggi di errore o conferma

Le funzioni principali presenti sono:

* **onRegister:** si occupa di inviare i dati al server back-end per registrare l’utente nel database. Se l’operazione va a buon fine, mostra un messaggio di successo e reindirizza alla schermata di login. In caso contrario, mostra un Messaggio di errore.
* **goToLogin:** serve per tornare alla schermata di login.

**3: Home**

La schermata **Home** rappresenta il punto di ingresso principale dell’app SoTourist.

Offre una panoramica dello stato attuale dell’utente (**viaggi in corso** o **imminenti**), suggerimenti su destinazioni e itinerari consigliati, oltre a fornire accesso rapido alle principali funzionalità grazie alla barra di navigazione.

La pagina è definita in “*src/app/pages/home”*. La struttura è contenuta nel file “*home.page.html”*, mentre la logica è gestita in “*home.page.ts”*.

Immagine che contiene testo, schermata, Sito Web, Pubblicità online

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

La pagina si presenta con l’header contenente il titolo dell’app gestito tramite il componente <app-header>.

La parte iniziale della pagina presenta una **Hero dinamica** dove:

* Se l’utente non ha viaggi attivi, viene mostrata un’immagine descrittiva accompagnata da un messaggio (“La tua prossima avventura ti aspetta”) e una call-to-action per creare rapidamente un nuovo viaggio.
* Se l’utente **ha un viaggio attivo o imminente** viene mostrato **nome della città**, **periodo di viaggio**, copertina dinamica, e un bottone per accedere direttamente all’itinerario.

Quando l’utente effettua lo scroll, viene mostrata una versione compatta della hero (**sticky**) per mantenere il contesto del viaggio attivo o imminente.

Successivamente troviamo la sezione **“Destinazioni trend”** con città popolari suggerite, e quando si clicca su un’immagine inizia la creazione dell’itinerario per quella città. Inoltre, questa sezione, presenta uno scroll orizzontale tra le varie foto delle città.

Troviamo anche la sezione **“Itinerari consigliati”**, ovvero una lista di itinerari già pronti mostrati tramite un’ion-card contenente un’immagine di copertina, nome della città, stile della vacanza e durata, e si può accedere all’itinerario creato semplicemente cliccando sulla card.

A fine pagina invece troviamo una citazione finale del Maestro Oogway, come chiusura ispirazionale.

Le principali funzionalità della pagina sono:

* **openCreate**: avvia il flusso di creazione, opzionalmente pre-selezionando una città
* **openItinerary**: apre un itinerario specifico
* **getFormattedCity**, **getTripDays**, **getPhotoUrl**: funzioni di utilità per il rendering dinamico delle informazioni.

La dinamicità della pagina è gestita tramite le seguenti variabili:

* **currentTrip**, **nextTrip**: viaggi in corso o imminenti
* **featuredItineraries**: array di itinerari consigliati
* **trending**: array di città “trend”
* **isShrunk**: booleano per gestire lo stato scroll del layout

**4: Navigazione tramite Tabs**

L’applicazione SoTourist implementa una barra di navigazione inferiore (*tabs*) per semplificare e velocizzare l’accesso alle sezioni principali dell’interfaccia. Questo approccio garantisce un'esperienza utente coerente e ottimizzata specialmente per dispositivi mobile.

La navigazione a tab è definita in “*src/app/pages/tabs”*. La struttura si trova in “*tabs.page.html”*, mentre la logica di gestione e routing è distribuita tra “*tabs.page.ts”* e “*tabs.routes.ts”*.

**Immagine che contiene testo, schermata, logo, simbolo

Il contenuto generato dall'IA potrebbe non essere corretto.**

La pagina utilizza il componente <ion-tabs> con una barra inferiore <ion-tab-bar slot="bottom">.

Ogni bottone è definito tramite <ion-tab-button> e include:

* Un’**icona dinamica** (ion-icon) che cambia (si riempie) in base al tab attivo (il contenuto dell’icona diventa colorato se attivo);
* Un **indicatore visivo personalizzato** (div.tab-pill) per evidenziare il tab selezionato
* La destinazione (href) verso la route corrispondente

I pulsanti disponibili sono:

* **Home** (/tabs/home) – Icona: home / home-outline: riporta alla pagina home
* **Viaggi** (/tabs/viaggi) – Icona: airplane / airplane-outline: riporta alla pagina viaggi
* **Impostazioni** (/tabs/settings) – Icona: settings / settings-outline: riporta alla pagina di gestione del profilo.

Il file “*tabs.page.ts”* gestisce la selezione dei pulsanti aggiornando la variabile *selectedTab*, mentre “*tabs.routes.ts”* definisce le rotte per caricare le varie pagine dinamicamente.

Quando l’utente seleziona un pulsante, l’app naviga alla route associata senza ricaricare l’intera pagina (navigazione nativa in stile mobile), e gli elementi visuali forniscono un feedback immediato sul pulsante attivo.

**5: Viaggi**

La pagina “Viaggi” rappresenta il punto centrale della gestione degli itinerari per l’utente. Da qui è possibile consultare i viaggi in corso, imminenti, futuri o in bozza, accedere allo storico e creare nuovi itinerari.

La pagina è definita in “*src/app/pages/viaggi”*. La struttura è contenuta nel file “*viaggi.page.html”*, mentre la logica è implementata in “*viaggi.page.ts”*.

**Immagine che contiene testo, schermata, Pagina Web, Sito Web

Il contenuto generato dall'IA potrebbe non essere corretto.**

La pagina presenta un’header <app-header> che mostra il titolo "SoTourist" con icona list-outline.

Se l’utente è registrato, viene mostrato il componente <app-user-hero> dove all’interno troviamo: il **numero di viaggi futuri**, il **numero di viaggi passati** e il **numero di luoghi visitati**.

Il contenuto principale della pagina si divide in 3 sezioni:

* **Viaggio in corso:** presenta un itinerario attivo;
* **Viaggio imminente**: presenta un viaggio con data di partenza a breve;
* **Prossimi viaggi**: itinerari con date future;
* **Bozze**: itinerari ancora non generati, e quindi salvati solo localmente.

Ogni viaggio è visualizzato tramite un componente <app-trip-card> o <app-unfinished-card> con pulsanti per aprire o eliminare l’itinerario.

Se l’utente non ha viaggi, nella sezione principale viene mostrato il logo, un saluto personalizzato e un’icona add-outline che guida l’utente a iniziare la creazione di un itinerario.

Immagine che contiene testo, schermata, logo, design

Il contenuto generato dall'IA potrebbe non essere corretto.

Infine, in basso a destra, troviamo un **Floating Action Button (FAB)**: un pulsante con due azioni distinte:

* **Crea** (sinistra): apre la pagina di creazione itinerario
* **Storico** (destra): apre la pagina “Storico viaggi”

Le funzioni principali della pagina sono:

* **openItinerary**: apre il dettaglio di un itinerario selezionato.
* **deleteTrip** / **deleteDraft**: rimuove rispettivamente viaggi confermati o bozze.
* **goToCreate**: naviga alla schermata di creazione.
* **openStorico**: naviga allo storico dei viaggi.

La dinamicità della pagina viene gestita attraverso le seguenti variabili:

* **inCorso**, **imminente**, **futuri**, **drafts**: array che rappresentano rispettivamente i viaggi in corso, i viaggi imminenti, i viaggi futuri e le bozze;
* **userId**, **username**, **isGuest**, **loaded**: rappresentano i dati utente, se è un’ospite o meno, e lo stato di caricamento.

**6: Crea un itinerario**

L’utente può creare un nuovo itinerario personalizzato seguendo un processo guidato in 4 step: selezione città, scelta dell’alloggio, definizione delle date e conferma del riepilogo. Ogni passaggio è progettato per offrire un’interazione chiara e progressiva, garantendo una user experience fluida. Al termine della creazione dell’itinerario, i dati inseriti verranno salvati in locale.

La schermata è definita in “*src/app/pages/crea”*. Il layout viene gestito in “*crea.page.html”*, mentre la logica è in “*crea.page.ts”*. I vari step sono gestiti tramite logica a "carousel orizzontale", spostando la vista in base al valore della variabile step. I vari step presenti sono:

|  |  |
| --- | --- |
|  | * **Step 1 – Città**: utilizza il componente personalizzato <app-google-autocomplete> con filtro per sole città (['(cities)']). Il bottone “Avanti” viene abilitato solo dopo la selezione di una città valida. |
| Immagine che contiene testo, schermata, design  Il contenuto generato dall'IA potrebbe non essere corretto. | * **Step 2 – Alloggio**: l’utente sceglie se inserire un hotel o un indirizzo manuale tramite un ion-segment. Anche qui viene usato <app-google-autocomplete> con tipi ['lodging'] o ['address'] in base alla selezione. |
| Immagine che contiene testo, schermata, design  Il contenuto generato dall'IA potrebbe non essere corretto. | * **Step 3 – Date**: viene usato il componente <app-range-calendar-lite> che permette la selezione di un intervallo di date con un’interfaccia a calendario. |
| **Immagine che contiene testo, schermata  Il contenuto generato dall'IA potrebbe non essere corretto.** | * **Step 4 – Riepilogo e conferma**: mostra un riepilogo delle informazioni inserite (città, alloggio, date) e permette la conferma tramite un pulsante “Conferma”. |

Tutti i pulsanti di navigazione sono gestiti tramite le funzioni di step e validazione e sono disabilitati se i dati richiesti non sono completi o validi.

Le funzioni logiche presenti in “*crea.page.ts”* sono:

* **handleCityPlace**: gestisce la selezione della città e aggiorna i dati.
* **handleAccommodationPlace**: salva i dati dell’alloggio.
* **onDatesSelected**: aggiorna le date selezionate.
* **setAccommodation**: salva l’alloggio e passa allo step successivo.
* **setDates**: salva le date e passa allo step finale.
* **prevStep**: torna allo step precedente.
* **confirm**: finalizza l’itinerario (logica di conferma).

I cambiamenti da uno step all’altro sono gestiti tramite la variabile **step**, i vari input dell’utente vengono salvati nelle variabili **cityInput**, **accommodationInput**, **datesInput** e per abilitare i pulsanti dopo che l’input è stato validato vengono utilizzati dei flag di validazione denominati **isCityValid**, **isAccommodationValid.**

**7: Storico Viaggi**

La pagina “Storico” mostra all’utente l’elenco dei viaggi conclusi in passato. Da qui è possibile rivedere, consultare o eliminare itinerari precedenti. Questa funzionalità è utile per avere memoria dei viaggi effettuati o per riutilizzarli come base per nuovi itinerari.

La schermata è definita in *src/app/pages/storico-viaggi*. La struttura è contenuta nel file *storico-viaggi.page.html*, mentre la logica è implementata in *storico-viaggi.page.ts*.

Immagine che contiene testo, schermata, design

Il contenuto generato dall'IA potrebbe non essere corretto.

Anche qui troviamo il titolo della pagina “Storico”, gestito tramite il componente <app-navigation-bar>.

Nella sezione principale, se sono presenti viaggi precedenti (pastTrips.length > 0), troviamo le rispettive card all’interno di un contenitore div.trip-list. Per ogni card (<app-trip-card>) che mostra le informazioni del viaggio e permette due azioni:

* **open**: per visualizzare i dettagli dell’itinerario
* **remove**: per eliminare il viaggio dallo storico

Immagine che contiene testo, schermata, design

Il contenuto generato dall'IA potrebbe non essere corretto.Se non ci sono viaggi passati, viene mostrato un messaggio testuale: “Non hai ancora viaggi passati.”

Le funzionalità presenti in questa pagina sono:

* **goToTrip**: naviga alla visualizzazione di un viaggio selezionato.
* **deleteTrip**: rimuove il viaggio dallo storico, aggiornando la lista visibile.

Il tutto è gestito tramite una variabile **pastTrips**, un array contenente tutti i viaggi conclusi salvati dall’utente.

**8: Itinerario**

Questa pagina permette all’utente di completare e personalizzare un itinerario di viaggio già creato (bozza), inserendo luoghi da visitare, preferenze di viaggio e uno stile di vacanza.

La pagina è definita in *src/app/pages/itinerario*. La struttura si trova in *itinerario.page.html*, mentre la logica è implementata in *itinerario.page.ts*.

Immagine che contiene testo, schermata, software

Il contenuto generato dall'IA potrebbe non essere corretto.

La pagina è suddivisa in 2 sezioni principali:

* **Luoghi da visitare**: dove l’utente può aggiungere luoghi che non vuole perdersi (*tripMustSee*), locali gastronomici da visitare (*tripEatPlaces*) e luoghi già visitati se è già stato in quella città (*tripAlreadyVisited*). Ogni categoria è rappresentata con un’ion-item interattiva che mostra gli elementi sotto forma di chip, oppure un’icona “+” se vuota. L’aggiunta avviene tramite modale con ricerca su Google Places attraverso il componente <app-google-autocomplete>.

**N.B.**: La Personalizzazione dei luoghi (primo riquadro in alto) è dedicata soltanto agli utenti abbonati come “Premium” o “Gold”, che potranno servirsi di arricchire la loro esperienza andando ad arricchire l’algoritmo di generazione dell’itinerario con questi parametri aggiuntivi.

* **Preferenze di viaggio**: dove l’utente può indicare quale mezzo utilizzerà per spostarsi prevalentemente durante la vacanza (mezzo preferito come auto, bici, a piedi ecc..), la tipologia di vacanza che preferisce (standard, relax, shopping, ecc.) ed eventualmente può chiedere consigli all’AI (**N.B.** “Domanda all’AI” è una funzionalità non ancora disponibile.)

Terminata la personalizzazione dell’itinerario, esso verrà effettivamente generato

(la generazione personalizzata viene spiegata nel file itinerarycontroller.js, dove spiegheremo l’impatto di queste scelte nella generazione dell’itinerario)  
Per ogni opzione (es: se clicco su “Luoghi imperdibili”) viene presentata una modale fluttuante, adattate dinamicamente al tipo di informazione da inserire.

Immagine che contiene testo, schermata, Carattere, logo

Il contenuto generato dall'IA potrebbe non essere corretto. Immagine che contiene testo, schermata, Carattere, design

Il contenuto generato dall'IA potrebbe non essere corretto.

Le funzionalità principali della pagina sono:

* **openModal**: apre la modale associata alla categoria selezionata.
* **onPlaceSelectedInModal**: aggiunge un luogo selezionato nella lista temporanea.
* **selectValue**: seleziona una voce per trasporto o stile.
* **saveModal**, **closeModal**: salva o annulla i dati inseriti nel modale.
* **generateItinerary**: avvia il processo di generazione dell’itinerario.

I vari input dell’utente, dati temporanei e salvataggi definitivi sono gestiti dalle variabili

**tripMustSee** (array di luoghi da visitare), **tripEatPlaces** (array dei ristoranti da visitare), **tripAlreadyVisited** (array di luoghi già visitati), **tripTransport**, **tripPrompt**, **trip.style** (preferenze dell’utente), **modalMode**, **modalVisible**, **modalTempPlaces** (stato e dati del modale), **isLoading**, **isLocalTrip** (gestiscono stato di caricamento e condizione per la generazione).

**9: Personalizzazione Itinerario <DA FARE>**

Questa funzionalità consente all’utente di modificare manualmente la disposizione dei luoghi all’interno del proprio itinerario, suddividendoli per giorno e fascia oraria. È particolarmente utile per riorganizzare tappe proposte, aggiungere o rimuovere luoghi e adattare l’esperienza alle proprie preferenze personali.

La schermata è definita in *src/app/pages/personalizzazione*. La struttura è contenuta nel file *personalizzazione.page.html*, mentre la logica è implementata in *personalizzazione.page.ts*.

Immagine che contiene testo, software, Sito Web, Pagina Web

Il contenuto generato dall'IA potrebbe non essere corretto.

La pagina presenta una **barra laterale sinistra** che mostra la lista dei giorni dell’itinerario, e permette di selezionare il giorno attivo tramite click.

Per ogni giorno selezionato, l’utente può personalizzare le tre **fasce orarie** (Mattina, Pomeriggio, Sera) grazie al supporto del **drag-and-drop** (grazie a cdkDropList) per riordinare i luoghi all’interno della fascia. Ogni luogo viene mostrato come *app-luogo-card* ed è possibile eliminare dei luoghi dall’itinerario utilizzando l’icona trash-outline, se essi non sono bloccati (alloggio).

In fondo a ogni fascia, è presente un bottone “**Aggiungi luogo**”.

* + Quando attivato, si apre un **pannello di autocomplete** dove l’utente può scegliere di aggiungere un ristorante o un luogo di interesse turistico, usando il componente <app-google-autocomplete>.
* **Salvataggio modifiche**:
  + In basso a destra è presente un pulsante FAB con icona checkmark-done-outline per salvare le modifiche effettuate.

**Logica della pagina (TypeScript):**

La logica controlla le modifiche per giorno e fascia, integra l’aggiunta di luoghi e aggiorna l’itinerario in tempo reale.

* **Variabili principali**:
  + days(): restituisce i dati dei giorni dell’itinerario.
  + slots: array con le fasce orarie da gestire.
  + activeDay(): indice del giorno attualmente selezionato.
  + autocompleteOpen: stato del pannello di aggiunta luogo.
  + tripBounds: limiti geografici per l’autocomplete.
* **Funzioni principali**:
  + selectDay(i): seleziona il giorno corrente.
  + drop(event, slot): gestisce il riordinamento degli elementi tramite drag-and-drop.
  + removePlace(slot, idx): rimuove un luogo da una fascia oraria.
  + addPlace(slot): avvia il flusso di aggiunta di un nuovo luogo.
  + startAutocomplete(type): attiva il pannello per aggiungere un luogo del tipo indicato.
  + onPlaceSelected(event): aggiunge il luogo selezionato al giorno/fascia corrente.
  + saveItinerary(): salva le modifiche e aggiorna l’itinerario.

**10: Modifica Date**

Questa funzionalità consente all’utente di modificare le date associate ad un itinerario già creato, questa pagina viene chiamata se l’utente genera erroneamente due vacanze nello stesso arco temporale, anche in minima parte, o decide di modificare le date prima ancora di generare l’itinerario finale.  
utile per l’utente per poter aggiornare il calendario del proprio itinerario senza doverlo ricreare da zero.

La schermata è definita *in src/app/pages/modifica-date*. La struttura è contenuta nel file *modifica-date.page.html*, mentre la logica è implementata in *modifica-date.page.ts*.

Immagine che contiene testo, schermata, Carattere, design

Il contenuto generato dall'IA potrebbe non essere corretto.

La pagina utilizza il componente <app-navigation-bar> con il titolo “Modifica Date” e la possibilità di tornare indietro. In alto viene visualizzata la **città dell’itinerario** corrente, seguito dal componente

personalizzato <app-range-calendar-lite> che permette di selezionare un nuovo intervallo di date.

Le date sono limitate da: **minDate**: la data attuale (oggi) e **maxDays**: numero massimo di giorni selezionabili, ma quest’ultimo è opzionale e viene preso soltanto quando si vogliono modificare le date dei viaggi già creati dall’account ufficiale di SoTourist (PS: questo concetto viene esplicato meglio nella funzionalità “Home”).

Quando l’utente seleziona le nuove date, queste vengono mostrate in una sezione di riepilogo sottostante al calendario, seguito dal pulsante **"Salva e continua"** che viene abilitato solo se entrambe le date (newStartDate e newEndDate) sono state selezionate.

Le funzionalità presenti sono:

* **getFormattedCity**: restituisce il nome della città formattato correttamente per la visualizzazione.
* **onDatesSelected**: viene chiamata alla selezione delle nuove date nel calendario e aggiorna **newStartDate** e **newEndDate**.
* **saveDates**: salva le nuove date nell’itinerario e reindirizza l’utente al passo successivo (alla pagina panoramica se si tratta di un viaggio già caricato, o alla pagina “viaggi” se si tratta di un itinerario creato da un utente).

Le variabili utilizzate sono: **trip** che rappresenta l’itinerario in modifica, **newStartDate**, e **newEndDate** che sono le nuove date selezionate dall’utente, **today** è la data minima selezionabile, ed infine **maxDays** che è il massimo numero di giorni consentiti.

**11: Panoramica**

La pagina “Panoramica” offre una visione sintetica e interattiva dell’intero itinerario generato, suddiviso per giornate. Consente all’utente di visualizzare rapidamente il programma giorno per giorno, accedere ai dettagli e modificare la pianificazione.

La schermata è definita in *src/app/pages/panoramica*. La struttura è descritta in *panoramica.page.html*, mentre la logica è gestita in *panoramica.page.ts*.

Immagine che contiene testo, schermata, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

La pagina presenta una **Hero dinamica** che mostra l’intestazione visiva dell’itinerario, con:

*Nome città*, *Date del viaggio* e *Alloggio.* Lo sfondo è un’immagine (*heroPhotoUrl*) con overlay opaco adattivo e titolo ridimensionabile dinamicamente.

Al di sotto troviamo la lista dei giorni dell’itinerario dove ogni giorno viene rappresentato con un ion-card contenente:

* titolo “Giorno X”
* pulsante per accedere alla modifica dell’itinerario
* data
* numero attività
* **Barra di avanzamento** suddivisa per fasce orarie (Mattina, Pomeriggio, Sera), con indicazione visiva del completamento.
* In fondo, una sezione mostra un messaggio se non ci sono giorni da visualizzare.

Le funzioni principali della pagina sono:

* **getFormattedCity**, **getFormattedAccommodation**: restituiscono valori formattati.
* **getDayItems**: restituisce le attività associate a un determinato giorno.
* **getDayDate**: restituisce la data formattata del giorno.
* **getStepProgress**: restituisce la percentuale di completamento della fascia oraria per il giorno i.
* **openDay**: apre i dettagli del giorno.
* **vaiAPersonalizzazione**: permette la modifica delle attività (naviga alla sezione di personalizzazione).

Le variabili utilizzate sono: **trip** che contiene l’itinerario attuale, **daysCount** che rappresenta il numero di giorni totali nell’itinerario, infine abbiamo **heroPhotoUrl**, **overlayOpacity**, **titleFontSize** che gestiscono stile e comportamento della sezione hero.

**12: Mappa**

La funzionalità "Mappa" consente all’utente di visualizzare graficamente l’itinerario giornaliero generato, esplorare i luoghi previsti per la giornata selezionata e consultarne i dettagli. L'interfaccia è interattiva e integrata con Google Maps.

La pagina è definita in *src/app/pages/map*. La struttura si trova in *map.page.html*, mentre la logica è gestita in *map.page.ts*.



Nello sfondo troviamo la **mappa** di Google maps, la quale è inizializzata in un contenitore div con id="map" e visualizza i luoghi previsti per il giorno selezionato. Su entrambi i lati troviamo due selettori:

* il primo ci permette di selezionare il **giorno** (es. Giorno 1, Giorno 2…) e tramite un menu a tendina l’utente può selezionare il giorno dell’itinerario (es. Giorno 1, Giorno 2...).
* il secondo ci permette di individuare le tappe per **momento della giornata** (Mattina, Pomeriggio, Sera)

Nella parte inferiore, invece, troviamo un **drawer,** ovvero una sezione espandibile che mostra le **schede dei luoghi** (<app-luogo-card>) previsti per il giorno e fascia selezionata. Le card possono essere toccate per evidenziare il luogo sulla mappa o aperte per scoprirne dettagli. Quando si clicca su una card-luogo viene mostrata una modale (ion-model) che mostra: una **foto** (se disponibile), **nome, indirizzo, e valutazione** del luogo, un **pulsante** per aprire il luogo in Google Maps**.**

Le funzionalità principali della pagina sono:

* **selectDay**: aggiorna il giorno selezionato.
* **selectTimeSlot**: imposta la fascia oraria corrente.
* **openPlace**: seleziona un luogo sulla mappa e apre i dettagli.
* **openInGoogleMaps**: apre il luogo selezionato in Google Maps tramite coordinate.
* **toggleDayList**, **toggleTimeList**: apertura/chiusura dei menu dropdown.

La gestione dinamica della pagina avviene tramite delle variabili:

* **days, currentDay, selectedTimeSlot** servono per la gestione del giorno e della fascia oraria selezionata.
* **todayPlaces** è array dei luoghi da mostrare per il giorno/fascia corrente.
* **drawerExpanded, detailOpen, selectedIndex** permettono di gestire dinamicamente l’interfaccia utente.

**13: Settings**

La pagina “Settings” rappresenta il punto di accesso centrale per tutte le funzioni legate alla gestione del profilo utente, upgrade, impostazioni avanzate, e informazioni legali. Fornisce una panoramica intuitiva e accessibile su tutte le aree personali e amministrative dell’app.

La schermata è definita in *src/app/pages/settings*. La struttura si trova in *settings.page.html*, mentre la logica di gestione è implementata in *settings.page.ts*.

Immagine che contiene schermata, testo, design

Il contenuto generato dall'IA potrebbe non essere corretto.

La pagina presenta un **Header** dato dal componente <app-header> che mostra il nome dell’applicazione “SoTourist”.

Il primo blocco è dato dalla sezione **Profilo utente** che mostra, attraverso un ion-card, l’**icona profilo**, **nome**, **email** e **badge del piano** e al click su questa sezione si viene reindirizzati alla pagina **Profilo** (routerLink="/profilo").

Poi troviamo un pulsante evidenziato per effettuare l’upgrade ad un piano **Premium** (routerLink="/upgrade").

Successivamente abbiamo la voce “**Impostazioni avanzate**” che porta alla schermata dedicata alle **preferenze tecniche** **e di sistema** (routerLink="/impostazioni").

Infine, troviamo il blocco dedicato alle informazioni generali sull’applicazione (come la **versione** della stessa, attualmente la 0.8.2) e al **regolamento della privacy e delle licenze** che contiene contenuti riguardanti *Copyright*, *trattamento dei dati secondo il GDPR*, *uso delle API di Google*, *icone e immagini*, *termini di servizio* (ultima data di modifica di questa sezione giuridica: Giugno 2025).

Immagine che contiene testo, schermata, Carattere, documento

Il contenuto generato dall'IA potrebbe non essere corretto.Sia la sezione della versione, sia la sezione del regolamento sono gestite tramite un ion-modal.

L’ultimo blocco della pagina riguarda il tasto di **Logout / Login** dove:

* se l’utente è loggato, compare il bottone **Logout**.
* se è ospite, viene mostrato il bottone **Login** per passare a un account registrato.

Le funzionalità principali della pagina sono:

* **getBadgeClass**: restituisce la classe CSS per il badge in base al piano.
* **openAbout**: apre il modale “Info & versione”.
* **openPrivacy**: apre il modale “Privacy e licenze”.
* **confirmLogout** / **logout**: gestiscono la disconnessione dell’utente mostrando prima un messaggio di conferma.

Le variabili utilizzate per gestire questa pagina sono:

* **userId**, **username**, **email**, **subscriptionPlan**, **isGuest:** dati che riguardano l’utente
* **showAboutModal**, **showAboutModalPrivacy**: booleani che servono per la gestione delle due modali.

Oltre a gestire il routing della gestione dei diversi abbonamenti, la pagina *profilo.page.ts* presenta le seguenti funzionalità:

* **getBadgeClass**: restituisce la classe CSS per visualizzate il badge in base al piano
* **goToUpgrade**: naviga alla schermata di upgrade/gestione abbonamento
* **changePassword**: attiva la modifica password
* **confirmDeleteAccount**: mostra la conferma per eliminare l’account.

Tutte le funzionalità e la dinamicità della pagina viene gestita attraverso le seguenti variabili: **userId** (ID dell’utente), **username** (username dell’utente), **email** (email dell’utente), **subscriptionPlan** (data di sottoscrizione al piano d’abbonamento scelto), **subscriptionExpiry** (data di scadenza del piano d’abbonamento scelto), **registrationDate** (data di registrazione del profilo).

**14: Profilo**

La sezione “Profilo” consente all’utente di visualizzare e gestire le proprie informazioni personali, il piano di abbonamento e le opzioni di sicurezza dell’account.

La pagina è definita in *src/app/pages/profilo*. La struttura si trova in *profilo.page.html*, mentre la logica di gestione è contenuta in *profilo.page.ts*.

**Immagine che contiene testo, schermata, software, Pagina Web

Il contenuto generato dall'IA potrebbe non essere corretto.**

La pagina presenta una sezione dove sono presenti le **informazioni dell’utente** mostrate attraverso un ion-card che mostra:

* l’**icona profilo** (gestita dal componente <app-profile-icon>, personalizzabile)
* il **nome utente**, **email**, **piano di abbonamento** (dinamico e stilizzato tramite getBadgeClass) e **data di registrazione**

Successivamente troviamo la sezione **Abbonamento** implementata da un ion-list con intestazione e due item:

1. Visualizzazione del piano attivo e, se non standard, mostra anche la **data di scadenza** dell’abbonamento;
2. Pulsante per **gestire o aggiornare l’abbonamento**

Infine, troviamo la sezione dedicata alla **Sicurezza dell’account** dove abbiamo le seguenti azioni:

* **Cambia password** che permette di cambiare la propria password;
* **Elimina account**, che permette di eliminare l’account dall’applicazione ed è evidenziata con stile diverso per indicare l’azione critica.

**15: Cambio credenziali**

L’applicazione offre la possibilità agli utenti autenticati di modificare i propri dati personali, come Username, Email e Password, tramite la schermata dedicata al cambio credenziali.

La schermata è accessibile solo agli utenti loggati e si trova in src/app/pages/change-password.

Immagine che contiene testo, schermata, design

Il contenuto generato dall'IA potrebbe non essere corretto.

Il layout della pagina è definito nel file /change-password.page.html.

Vengono utilizzati i componenti Ionic come ion-input e ion-button per la gestione dei campi e delle azioni.

Sono presenti due sezioni distinte:

Modifica di Username ed Email

L’utente può aggiornare il proprio nome e indirizzo email. Il pulsante "Salva Mail" si abilita solo se i dati inseriti sono diversi da quelli attualmente salvati e se entrambi i campi non sono vuoti.

Cambio della Password

L’utente deve inserire la password attuale, la nuova password e la conferma della nuova password. Il pulsante "Salva Password" si abilita solo se tutti i campi sono compilati.

**Nota bene:** (L’email e la nuova password e conferma password devono rispettar le caratteristiche descritte in pagina registrazione)

Nel file /change-password.page.ts viene definita la logica della pagina, tra cui:

Il recupero delle credenziali attuali dal localStorage al caricamento della pagina

La gestione del salvataggio dei dati tramite il servizio AuthService

L’utilizzo del servizio ToastService per mostrare messaggi di successo, errore o avviso all’utente

Le funzioni principali presenti sono:

* **refreshAccount**: Recupera i dati attuali di Username ed Email dal localStorage e li mostra all’utente nella schermata.
* **saveMail:** Consente di aggiornare Username ed Email solo se modificati. Invia la richiesta al backend tramite AuthService e aggiorna i dati salvati localmente. Mostra un messaggio di conferma o errore in base all’esito**.**
* **savePassword: Consente di cambiare la password, ma solo se la nuova password coincide con la conferma. In caso di successo, viene mostrato un messaggio di conferma. In caso di errore o password sbagliata, viene mostrato un messaggio di errore. Se le nuove password non coincidono, viene mostrato un avviso.**

**16: Impostazioni avanzate**

Da questa pagina, l’utente può personalizzare alcune preferenze dell’app per migliorarne l’esperienza d’uso, tra cui il tema grafico (chiaro o scuro) e la lingua dell’interfaccia.

La schermata è definita in *src/app/pages/avanzate*. La struttura è descritta in *avanzate.page.html*, mentre la logica della pagina è gestita in *avanzate.page.ts*.

****

La pagina mostra il titolo della pagina all’interno del componente <app-navigation-bar> con possibilità di tornare alla pagina precedente.

La lista delle preferenze è racchiusa in un <ion-list> e contiene:

* un **toggle tema chiaro/scuro**: un componente personalizzato <app-toggle>che permette all’utente di passare da tema chiaro a tema scuro. L’etichetta e l’icona cambiano dinamicamente in base allo stato (luna per tema scuro o sole per tema chiaro).
* il **selettore lingua**: tramite un <ion-select> è possibile selezionare la lingua dell’interfaccia. Attualmente è disponibile solo l’opzione “Italiano”.

Le funzionalità presenti sono:

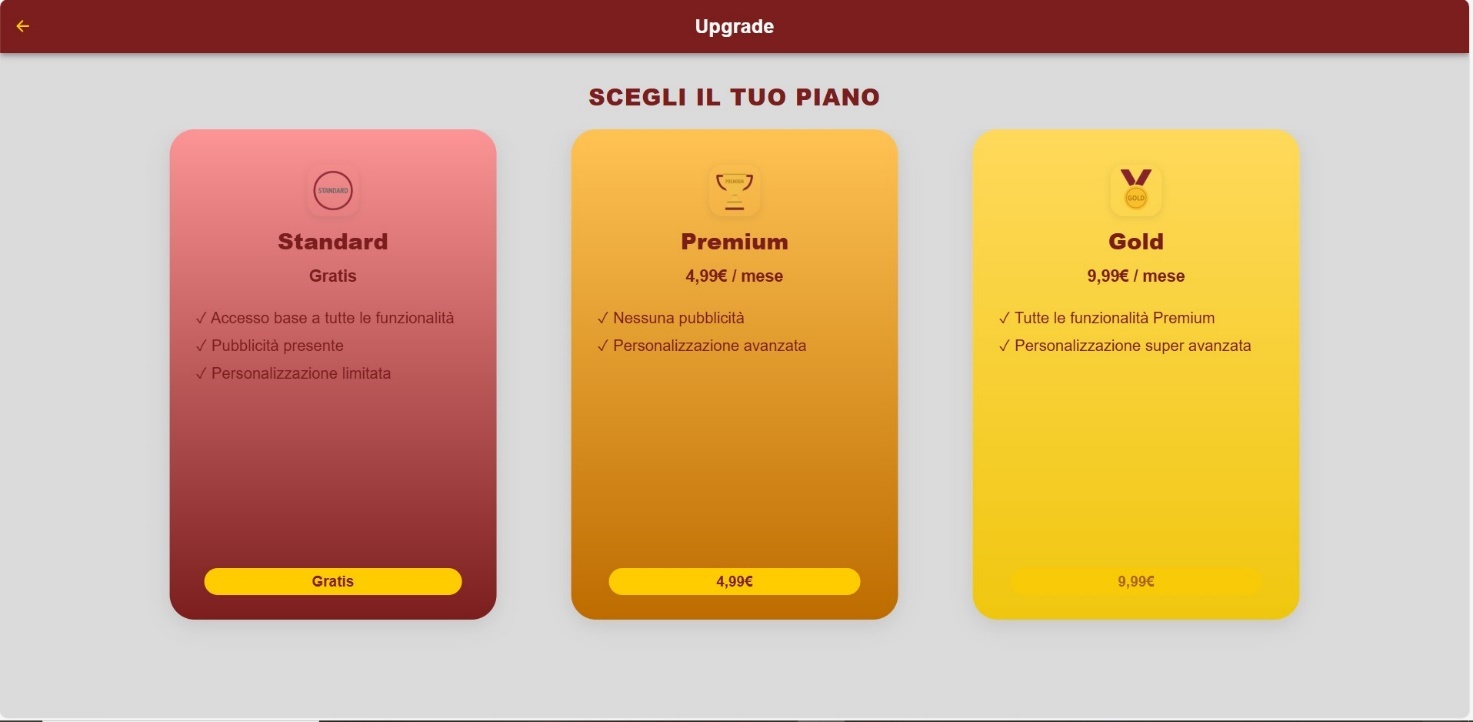
* **toggleDarkMode**: cambia dinamicamente il tema dell’app da chiaro a scuro e viceversa, aggiornando lo stile globale (presente nella pagina *variable.scss*)
* **changeLanguage**: aggiorna la lingua corrente dell’interfaccia. Anche se al momento è disponibile solo l’italiano, il sistema è predisposto per future estensioni multilingua.

Le impostazioni definite dalle due funzionalità descritte in precedenza vengono salvate nelle variabili **darkMode** per il tema e **language** per la lingua scelta.

**17: Upgrade Piano Abbonamento**

La funzionalità “Upgrade” consente agli utenti di scegliere tra diversi piani di abbonamento dell’app SoTourist, confrontando caratteristiche e prezzi in un’interfaccia visuale e interattiva. Questo sistema supporta la monetizzazione e la personalizzazione dell’esperienza utente.

La pagina è definita in *src/app/pages/upgrade*. La struttura è contenuta nel file *upgrade.page.html*, mentre la logica è gestita in *upgrade.page.ts*.



La schermata utilizza un layout a schede con **swiper orizzontale** (<swiper-container>) per scorrere tra i tre piani:

* **Standard (Gratis)**:
  + Accesso base
  + Presenza di pubblicità (non implementata)
  + Personalizzazione limitata
* **Premium (4,99€/mese)**:
  + Nessuna pubblicità
  + Personalizzazione avanzata (l’utente premium non può modificare l’itinerario manualmente dalla pagina “Personalizza itinerario”)
* **Gold (9,99€/mese)**:
  + Tutti i vantaggi Premium
  + Personalizzazione super avanzata

Ogni piano è rappresentato da un componente visuale (vertical-plan-card) che mostra:

* **Icona**
* **Nome piano**
* **Prezzo**
* **Lista dei vantaggi**
* **Pulsante di attivazione**

Lo swiper è centrato, con layout responsive e tocco abilitato per il mobile.

Le funzionalità presenti sono:

* **onStandard**, **onPremium**, **onGold**: attivano il piano selezionato, modificando lo stato utente e salvando l’informazione lato server o local storage.
* **getSlidesPerView**, **getSpaceBetween**: funzioni per layout adaptative

La dinamicità della pagina è gestita tramite le seguenti variabili:

* **currentType**: rappresenta il piano attivo o selezionato
* **isScrollable**: determina se lo swiper è scrollabile

**Componenti:**

Nel progetto sono presenti dei componenti che permettono il riutilizzo di codice  
I seguenti componenti sono definiti nella cartella src/app/components:

**1: generation-overlay**

Componente grafico di overlay che mostra un'animazione di caricamento con un messaggio opzionale. Serve per segnalare all'utente che è in corso un'operazione bloccante come la generazione dell'itinerario o altre operazioni di attesa. Migliora l'usabilità evitando che l'utente interagisca con la pagina durante i processi critici.

La schermata è definita in src/app/components/generation-overlay. La struttura HTML si trova in generation-overlay.component.html, mentre la logica è gestita in generation-overlay.component.ts.  
Il componente è standalone.

La pagina è composta da 3 parti: un contenitore che copre tutta la pagina come sfondo, un’animazione grafica poi gestita dal file SCSS e infine il messaggio (quest’ultimo opzionale)

Nel file .ts si gestiscono gli input del componente:

* **variabili principali:**
  + @Input() message: stringa opzionale da mostrare nell'overlay come messaggio descrittivo. Se null o vuota, viene nascosta l'area del messaggio.
* **Funzioni principali:**
  + ngOnInit(): funzione standard del ciclo di vita, in questo caso non implementa logica specifica.

**2: google-autocomplete**

Questo è un componente Google Autocomplete, serve per far comparire i suggerimenti di Google Places mentre l'utente scrive in un campo di input.

Funzionamento:

Appena l’utente inizia a scrivere, dopo un piccolo ritardo di 150 millisecondi (per ridurre il numero di richieste usate), viene interrogato il servizio di Autocomplete di Google. Se il testo inserito è almeno lungo 2 caratteri, il componente mostra i suggerimenti sotto al campo.

Quando l’utente clicca su uno di questi suggerimenti:

La lista scompare, l’input viene svuotato.

Viene recuperato il dettaglio completo del luogo (tipo nome, indirizzo, coordinate, ecc.).

Se il dettaglio era già stato scaricato in precedenza (c'è una piccola cache interna), viene subito emesso l’evento con i dati.

Se non era in cache, viene fatta la chiamata a Google per ottenere i dati completi. Tuttavia, la lista viene già rimossa dall’interfaccia, quindi l’utente non percepisce il tempo di attesa.

È presente anche una protezione per evitare che il campo perda il focus cliccando sulla lista dei suggerimenti.

Il componente è standalone, pertanto può essere riutilizzato in qualsiasi parte dell’applicazione senza richiedere dipendenze aggiuntive.

È possibile personalizzare:

* **Il testo segnaposto (placeholder).**
* Il tipo di input (type).
* I tipi di luoghi da suggerire (types).
* I limiti geografici della ricerca (bounds) e se si vuole forzare la ricerca solo dentro quei limiti (restrictToBounds).

è un campo di ricerca intelligente, usabile per indirizzi o punti d’interesse, che migliora l’esperienza utente con autocomplete personalizzati.

**3: app-header**

Componente riutilizzabile per la gestione della barra superiore delle pagine. Mostra il titolo della pagina e, opzionalmente, gestisce aspetti grafici come l'ombra inferiore o la navigazione. Permette di uniformare l'aspetto dell'intestazione e di semplificare la navigazione all'interno dell'applicazione.

La schermata è definita in src/app/components/app-header. La struttura HTML si trova in app-header.component.html, mentre la logica è gestita in app-header.component.ts. Il componente è standalone e usa componenti Ionic.

Struttura della pagina (HTML):

* **ion-header:** contenitore principale della barra superiore. Può avere o meno un'ombra inferiore a seconda della variabile box\_shadow.
* **ion-toolbar**: barra degli strumenti contenente titolo e pulsanti.
* **ion-title:** titolo della pagina, valorizzato dinamicamente tramite {{ title }}. Al click sul titolo viene richiamata la funzione NavHome(), che porta l'utente alla home.
* **ion-buttons:** area predisposta per inserire eventuali pulsanti a destra della barra, attualmente vuota ma pronta per future estensioni.

**Elementi dinamici:**

L'ombra inferiore della barra viene controllata dalla variabile box\_shadow, che applica o rimuove la classe CSS no-box-shadow.

Il titolo viene popolato dinamicamente tramite il valore passato all'@Input() title.

La struttura prevede opzioni per pulsanti aggiuntivi, se richiesti in futuro.

Nel file .ts si gestiscono gli input e le funzioni di navigazione:

**variabili principali:**

* **@Input() box\_shadow**: booleano che determina se la barra deve avere o meno l'ombra inferiore.
* **@Input() title**: stringa che rappresenta il titolo da mostrare nella barra.
* **@Input() showBackButton**: variabile predisposta per controllare l'eventuale visualizzazione di un bottone "indietro", attualmente non utilizzata nel template.
* **@Input() iconName**: nome dell'icona da usare, attualmente non utilizzato nel template.

**Funzioni principali:**

* **NavHome:** viene richiamata al click sul titolo e porta l'utente alla home page dell'app (/tabs/home).

**4: app-luogo-card**

Componente riutilizzabile che mostra in modo compatto o esteso le informazioni relative a un luogo all'interno di liste, anteprime o selezioni.

Consente agli utenti di visualizzare in modo rapido i dettagli essenziali dei luoghi, con gestione intelligente dell'immagine e layout adattabile. Fondamentale per la coerenza grafica nelle schermate che elencano tappe, suggerimenti o risultati di ricerca.

La schermata è definita in src/app/components/luogo-card. La struttura HTML si trova in luogo-card.component.html, mentre la logica è gestita in luogo-card.component.ts.

Il componente è standalone e utilizza componenti Ionic.

Struttura della pagina:

* Card descrittiva del luogo
* Immagine del luogo
* Indirizzo
* Distanza dalla prossima tappa (se disponibile)
* Valutazione in stelle (se presente)
* Fascia di prezzo con descrizione testuale
* Collegamento al sito web (se disponibile)
* Orari di apertura (se presenti)

Elementi dinamici:

Visualizzazione estesa o compatta gestita dalla variabile extends.

Mostra immagine o icona generica in base alla disponibilità della foto.

Informazioni facoltative mostrate solo se presenti nei dati del luogo.

**Logica della card:**

**Variabili principali:**

* **@Input() place**: oggetto Place contenente tutte le informazioni del luogo.
* **@Input() selected**: booleano che indica se la card è selezionata graficamente.
* **@Input() index**: numero della posizione nella lista, se necessario.
* **@Input() extends**: booleano che determina se mostrare la versione estesa della card.

**Eventi principali:**

* **@Output() clicked**: evento emesso al click sulla card, utile per comunicare la selezione al componente genitore.

**Funzioni principali:**

* **ngOnInit():** al caricamento, verifica se è presente photoFilename ma manca photoUrl. In tal caso, costruisce l'URL per il recupero dell'immagine dal server e controlla se esiste davvero. Se sì, aggiorna place.photoUrl di conseguenza.
* **checkImageExists:** funzione privata che verifica l'effettiva esistenza di un'immagine all'URL indicato, evitando errori o immagini rotte.

**5: app-navigation-bar**

Barra di navigazione personalizzata che fornisce un'intestazione coerente alle pagine dell'applicazione, con titolo centrato e pulsante "indietro" configurabile. Migliora la navigazione tra le pagine e l'aspetto grafico, garantendo uno stile uniforme. Il pulsante "indietro" che può gestire sia navigazione custom che fallback (azione predefinita).

La schermata è definita in src/app/components/navigation-bar. La struttura è interamente gestita nel template inline presente nel file .ts, mentre la logica è nello stesso file. Il componente è standalone e utilizza componenti Ionic.

**Struttura della pagina:**

* **ion-header:** contenitore principale dell'intestazione.
* **ion-toolbar**: barra degli strumenti con sfondo colorato in base al tema primario dell'app.
* **ion-buttons:** Contiene un bottone con icona "arrow-back-outline" che richiama la funzione goBack() al click.
* **ion-title:** Mostra il titolo della pagina, centrato e con stile personalizzato (bianco e in grassetto).
* **ion-buttons:** Contiene un bottone trasparente e inattivo, utilizzato esclusivamente per mantenere la simmetria visiva rispetto al bottone "indietro" a sinistra.

**Elementi dinamici:**

* Il titolo viene popolato dinamicamente tramite la variabile title.
* Il comportamento del bottone "indietro" dipende dalla presenza o meno della variabile backUrl.

**Logica del componente:**

* **Variabili principali:**
  + **@Input() title**: stringa che rappresenta il titolo da mostrare al centro della barra.
  + **@Input() backUrl**: stringa opzionale che indica la rotta a cui tornare al click sul bottone "indietro". Se non valorizzato, viene eseguito un comportamento di ritorno standard tramite il servizio NavigationService.
* **Funzioni principali:**
  + **goBack():** gestisce il comportamento del bottone "indietro". Se backUrl è valorizzato, effettua una navigazione diretta alla rotta indicata. Altrimenti, utilizza il metodo navigation.back('/') del NavigationService.

**6: app-profile-icon**

Componente grafico per la gestione dell'immagine profilo utente. Mostra l'icona o la foto profilo e, se abilitato, permette di modificarla o eliminarla tramite un'interfaccia semplice e intuitiva. Consente di personalizzare visivamente il profilo utente e si integra con il backend per il salvataggio e il recupero dell'immagine.

La schermata è definita in src/app/components/profile-icon. La struttura della pagina si trova nel file profile-icon.component.html, mentre la logica è gestita in profile-icon.component.ts. Il componente è standalone e utilizza componenti Ionic insieme ai servizi personalizzati del progetto.

**Struttura della pagina:**

* **avatar-wrapper**: contenitore principale che mostra l'immagine o l'icona predefinita. La dimensione è variabile in base alla proprietà size. Se la modalità editable è attiva, viene applicato uno stile aggiuntivo per indicare la possibilità di modifica.
* **img:** mostra l'immagine profilo se presente e non è forzato il placeholder.
* **ion-icon:** in alternativa, mostra un'icona standard a forma di persona se l'immagine non è disponibile**.**
* **input file:** campo nascosto per la selezione di un'immagine dal dispositivo, attivato solo se la modalità modifica è attiva.

**Elementi dinamici:**

* La dimensione del contenitore è controllata dinamicamente tramite la variabile size.
* La visualizzazione dell'immagine o dell'icona dipende dalla presenza della variabile image e dal flag forcePlaceholder.
* La modalità modifica viene abilitata o disabilitata tramite la variabile editable**.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Logica del componente:**

* **Variabili principali:**
  + **@Input() userId:** identificativo dell'utente, utilizzato per recuperare o aggiornare l'immagine profilo dal backend.
  + **@Input() size:** dimensione in pixel del contenitore dell'immagine**.**
  + **@Input() editable:** booleano che abilita o disabilita la modifica dell'immagine.
  + **@Input() forcePlaceholder**: forza la visualizzazione del placeholder anche se è presente un'immagine.
  + **@Input() src**: immagine preimpostata opzionale.
  + **@Output() changed**: evento emesso al cambio dell'immagine, con il nuovo valore in formato base64 o stringa vuota in caso di eliminazione.
* **Funzioni principali:**
  + **ngOnInit():** al caricamento, recupera l'immagine profilo dal backend se è presente userId.
  + **ngOnChanges():** in caso di cambiamento del userId, ricarica l'immagine dal backend.
  + **triggerFileInput():** se la modalità modifica è attiva, mostra un action sheet che permette di scegliere se modificare o eliminare l'immagine.
  + **onImageSelected():** gestisce la selezione di un'immagine dal dispositivo, la converte in base64, la invia al backend e aggiorna la preview**.**
  + **deleteImage():** elimina l'immagine profilo sia dal backend che dalla visualizzazione locale**.**

**7: app-range-calendar-lite**

Questo componente fornisce un calendario compatto e personalizzato per la selezione di un intervallo di date. È pensato per essere semplice da integrare e per offrire un'esperienza utente immediata, ideale ad esempio per la selezione delle date di un viaggio. Utilizza la libreria esterna Litepicker, integrata in modalità inline, così da mostrare il calendario direttamente nell'interfaccia senza pop-up esterni.

La schermata è definita in src/app/components/range-calendar-lite. La struttura grafica è nel file range-calendar-lite.component.html, mentre la logica si trova in range-calendar-lite.component.ts. Il componente è standalone.

**Struttura della pagina:**

* **calendarContainer:** è il contenitore su cui viene montato dinamicamente il calendario Litepicker. Nel file .html è rappresentato da un normale contenitore <div>, che viene associato tramite @ViewChild.

**Elementi dinamici:**

* L'intero calendario viene generato a runtime all'interno del contenitore specificato, grazie all'inizializzazione della libreria Litepicker.
* La data minima selezionabile viene impostata dinamicamente tramite la variabile minDate.
* È possibile definire un numero massimo di giorni selezionabili tramite la variabile maxDays.
* L'evento di selezione delle date viene comunicato al componente genitore tramite l'output datesSelected.

**Logica del componente:**

* **Variabili principali:**
  + @Input() minDate: data minima selezionabile. Può essere una stringa o un oggetto Date.
  + @Input() maxDays: numero massimo di giorni selezionabili. Parametro opzionale.
  + @Output() datesSelected: evento emesso quando l'utente seleziona correttamente un intervallo di date. Restituisce un oggetto con from e to in formato YYYY-MM-DD.
* **Funzioni principali:**
  + ngAfterViewInit(): viene eseguito dopo il rendering del componente. Inizializza il calendario Litepicker con le opzioni desiderate:
    - Modalità inline sempre visibile.
    - Selezione di un intervallo di date (singleMode: false).
    - Blocco delle selezioni retroattive (selectForward: true).
    - Localizzazione in italiano.
    - Gestione del limite massimo di giorni selezionabili.

In caso di superamento del limite massimo di giorni, il calendario mostra un avviso tramite il ToastService e annulla la selezione. Altrimenti, emette l'evento datesSelected con le date selezionate.

**8: app-toggle**

Questo componente rappresenta un interruttore grafico personalizzato (toggle switch), utile per attivare o disattivare opzioni in modo intuitivo. Rispetto ai classici checkbox, offre un aspetto più moderno e facilmente riconoscibile per indicare lo stato attivo o disattivo di un'opzione.

La schermata è definita in src/app/components/toggle. La struttura grafica è descritta nel file toggle.component.html, mentre la logica si trova in toggle.component.ts. Il componente si integra con il sistema di form reattivi di Angular.

**Struttura della pagina:**

* custom-toggle: contenitore principale dello switch, con stile personalizzato.
* input type="checkbox": checkbox nativo nascosto graficamente, ma che controlla effettivamente lo stato del toggle.
* span.slider: elemento grafico che rappresenta la leva dello switch, la cui posizione cambia in base allo stato checked.

**Elementi dinamici:**

* Lo stato del toggle (acceso o spento) è determinato dalla variabile checked.
* Al click o cambio di stato del checkbox viene richiamata la funzione toggle() che aggiorna lo stato e notifica eventuali listener.

**Logica del componente:**

* **Variabili principali:**
  + @Input() checked: booleano che rappresenta lo stato attuale del toggle (true = acceso, false = spento).
  + @Output() change: evento emesso ogni volta che il valore cambia, trasmettendo il nuovo stato (true o false).
* **Funzioni principali:**
  + toggle(): inverte lo stato del toggle, aggiorna il valore sia internamente che per eventuali form Angular, ed emette l'evento change.
  + writeValue(val): metodo previsto da ControlValueAccessor, permette di aggiornare il valore del toggle da un form esterno.
  + registerOnChange(fn): registra la funzione da chiamare quando il valore cambia, per la compatibilità con i form.
  + registerOnTouched(fn): registra la funzione da chiamare quando il toggle viene toccato o interagito.

**9: app-trip-card**

Questo componente rappresenta una card compatta che riassume le informazioni principali di un itinerario turistico salvato dall'utente. Permette di visualizzare in modo immediato città, date del viaggio, struttura scelta e durata, offrendo al tempo stesso pulsanti per la gestione rapida come l'apertura o l'eliminazione dell'itinerario.

La schermata è definita in src/app/components/trip-card. La struttura grafica si trova nel file trip-card.component.html, mentre la logica è gestita in trip-card.component.ts. Il componente è standalone e utilizza componenti Ionic.

**Struttura della pagina:**

* **ion-card:** contenitore principale della card con classe personalizzata boxes-style.
* **trip-layout:** suddivisione in due colonne, con immagine a sinistra e dettagli a destra.
* **trip-photo:** mostra la foto di copertina dell'itinerario, caricata dinamicamente tramite la variabile coverUrl.
* **trip-details:** contiene tutte le informazioni testuali e i controlli dell'itinerario:
  + Pulsante rosso con icona trash-outline per eliminare l'itinerario.
  + Nome della città, formattato tramite funzione dedicata.
  + Date del viaggio, formattate nel formato gg/MM/yyyy.
  + Nome della struttura ricettiva (alloggio), formattato.
  + Numero di giorni complessivi del viaggio, calcolato dinamicamente.

**Elementi dinamici:**

* L'immagine di copertina viene determinata dinamicamente al caricamento, tramite la funzione getPhotoUrl.
* I testi (città, alloggio, date, durata) vengono generati o formattati a partire dai dati contenuti nell'oggetto trip.
* Il pulsante elimina è visibile e attivo solo se necessario, ed emette un evento per consentire la rimozione dal componente padre.

**Logica del componente:**

* **Variabili principali:**
  + @Input() trip: oggetto TripWithId contenente tutte le informazioni dell'itinerario.
  + @Output() open: evento emesso quando l'utente apre l'itinerario (attualmente non usato direttamente, si naviga tramite router).
  + @Output() remove: evento emesso al click sull'icona elimina, con l'itineraryId dell'itinerario da rimuovere.
  + coverUrl: URL dell'immagine di copertina calcolato al caricamento.
* **Funzioni principali:**
  + ngOnInit(): al caricamento del componente, genera l'URL della copertina.
  + onClick(): naviga alla pagina panoramica dell'itinerario passando l'id nei parametri della rotta.
  + onDelete(event): emette l'evento di rimozione ed evita che il click propaghi alla funzione di apertura.
  + getFormattedCity(): restituisce il nome della città, formattato tramite utility esterna.
  + getFormattedAccommodation(): restituisce il nome dell'alloggio, formattato tramite utility esterna.
  + calculateTripLength(start, end): calcola la durata in giorni tra due date, con gestione degli arrotondamenti.

**10: app-unfinished-card**

Questo componente rappresenta la card dedicata agli itinerari "incompleti" o in fase di modifica, mostrando in modo compatto le informazioni principali come città, date, alloggio e durata del viaggio. Integra inoltre una gestione automatica dei conflitti di date, evidenziando visivamente eventuali sovrapposizioni con altri itinerari già esistenti. Consente all'utente di accedere rapidamente alla modifica o alla gestione dell'itinerario.

La schermata è definita in src/app/components/unfinished-card. La struttura grafica si trova nel file unfinished-card.component.html, mentre la logica è gestita in unfinished-card.component.ts. Il componente è standalone e utilizza componenti Ionic.

**Struttura della pagina:**

* **ion-card:** contenitore principale della card con classe personalizzata boxes-style.
* L'intera card può avere la classe aggiuntiva conflict se è stato rilevato un conflitto di date.
* trip-layout: struttura interna della card.
* trip-center: contiene dettagli e pulsanti di gestione.
* trip-details: informazioni testuali sull'itinerario:
* Nome della città, formattato.
* Date del viaggio, supporta sia formato startDate/endDate che start/end.
* Nome della struttura ricettiva.
* Numero di giorni del viaggio.
* side-buttons: pulsanti di azione:
  + Pulsante modifica (edit-btn), visibile solo se non c'è conflitto.
  + Pulsante elimina (delete-btn).
  + trip-arrow: freccia direzionale visibile solo se non c'è conflitto, indica la possibilità di accedere all'itinerario.

**Elementi dinamici:**

* La visualizzazione della freccia e del pulsante modifica dipende dallo stato hasConflict.
* Le date vengono gestite in modo flessibile, accettando sia dati provenienti dal backend (startDate/endDate) che dati temporanei o da bozza (start/end).
* Se viene rilevato un conflitto di date, la card si colora graficamente e il comportamento al click cambia.

**Logica del componente:**

* **Variabili principali:**
  + @Input() trip: oggetto TripWithId contenente i dati dell'itinerario.
  + @Input() intercept: se attivo, blocca la navigazione e si limita ad emettere l'evento open.
  + @Output() open: evento emesso al click sull'itinerario se intercept è attivo.
  + @Output() remove: evento emesso al click sull'icona elimina, con l'itineraryId da rimuovere.
  + hasConflict: flag che indica la presenza di un conflitto di date, calcolato automaticamente.
  + isGuest: indica se l'utente è loggato come ospite.
  + userId: identificativo dell'utente.
* **Funzioni principali:**
  + ngOnInit(): al caricamento, recupera l'userId e verifica eventuali conflitti di date tramite il ItineraryService. Se l'utente è ospite o mancano le date, salta il controllo.
  + checkConflicts(): interroga il backend per verificare se le date selezionate si sovrappongono ad altri itinerari.
  + onDelete(event): emette l'evento remove e blocca la propagazione del click.
  + onClick(): se non è attivo intercept, gestisce il comportamento al click:
  + Se c'è conflitto, reindirizza alla pagina di modifica date.
  + Altrimenti, reindirizza alla pagina itinerario.
  + getCityName(): formatta il nome della città eliminando CAP e abbreviazioni inutili.
  + getAccommodationName(): restituisce il nome dell'alloggio formattato.
  + calculateTripLength(start, end): calcola il numero di giorni del viaggio.
  + getCoverPhotoUrl(): genera l'URL della copertina, con immagine di default se necessario.
  + editItinerario(event): apre direttamente la pagina di modifica date dell'itinerario.

**11: app-user-hero**

Questo componente rappresenta una sezione riassuntiva del profilo utente, pensata per mostrare in modo compatto e immediato le informazioni principali dell'account. Visualizza il nome dell'utente, la foto profilo e alcune statistiche chiave come numero di itinerari futuri, itinerari completati e tappe visitate. È ideale per essere posizionato nella parte superiore della pagina profilo, fungendo da "hero" visivo.

La schermata è definita in src/app/components/user-hero. La struttura grafica si trova nel file user-hero.component.html, mentre la logica è gestita in user-hero.component.ts. Il componente è standalone e utilizza componenti Ionic e il componente personalizzato app-profile-icon.

**Struttura della pagina:**

* user-box: contenitore principale della sezione.
* hero-user: area superiore che contiene:
* Il componente app-profile-icon per mostrare l'immagine profilo dell'utente, con dimensione fissa a 70px e modalità non modificabile.
* Il nome dell'utente visualizzato in evidenza.
* user-info: area inferiore con le statistiche:
  + Numero di itinerari futuri (upcomingCount) con icona walk-outline.
  + Numero di itinerari completati (pastCount) con icona checkmark-done-outline.
  + Numero totale di tappe visitate (visitedPlacesCount) con icona location-outline.
* user-stats: contiene tre box con icona e numero:

**Elementi dinamici:**

* Il nome utente e le statistiche sono valorizzati dinamicamente tramite le variabili di input.
* La foto profilo viene caricata tramite il componente app-profile-icon utilizzando l'userId fornito.

**Logica del componente:**

* **Variabili principali:**
  + @Input() userId: identificativo dell'utente, utilizzato dal componente app-profile-icon.
  + @Input() username: nome da mostrare accanto alla foto profilo.
  + @Input() upcomingCount: numero di itinerari futuri.
  + @Input() pastCount: numero di itinerari completati.
  + @Input() visitedPlacesCount: numero totale di tappe visitate.
* **Funzioni principali:**
  + ngOnInit(): al momento non implementa logica specifica, ma è pronto per eventuali inizializzazioni future.

**Guards:**

Nel progetto sono presenti delle guards che permettono di proteggere le rotte e di controllare l'accesso a determinate pagine in base allo stato dell'utente o alla presenza di autorizzazioni necessarie. Le guards vengono utilizzate per migliorare la sicurezza e l'affidabilità dell'applicazione, impedendo che un utente non autorizzato o in una situazione non valida possa accedere a schermate riservate o Non valide nella sua situazione.

Le seguenti guards sono definite nella cartella src/app/guards

**1: GoldGuard**

Questa guard viene utilizzata per proteggere l'accesso a determinate rotte riservate agli utenti con account di tipo Gold. Garantisce che solo gli utenti autorizzati possano visualizzare funzionalità o pagine premium. In caso contrario, l'utente viene automaticamente reindirizzato alla pagina di upgrade o, se non autenticato, alla pagina di login.

La guard è definita nel file src/app/guards/gold.guard.ts e viene applicata direttamente alle rotte desiderate nel file di configurazione delle rotte dell'applicazione.

**Funzionamento:**

* La guard implementa l'interfaccia CanActivate e si attiva ogni volta che un utente tenta di accedere a una rotta protetta.

**Passaggi logici:**

Recupera l'userId tramite il servizio AuthService.  
Se l'utente non è autenticato (userId assente):

Reindirizza automaticamente alla pagina di login.

Impedisce l'accesso alla rotta.

Se l'utente è autenticato:

Esegue una chiamata al backend per recuperare il tipo di account (userType).

Se il tipo è "gold":

Permette l'accesso alla rotta.

In caso contrario:

Reindirizza alla pagina di upgrade.

Blocca l'accesso alla rotta.

**Gestione errori:**

In caso di errore nella verifica del tipo utente (problemi di rete o backend), l'utente viene reindirizzato alla pagina di login e l'accesso viene negato.

**2: PremiumGuard**

Questa guard viene utilizzata per proteggere le rotte riservate agli utenti con un account di tipo Premium o superiore. Serve a garantire che solo gli utenti autorizzati possano accedere a funzionalità avanzate o pagine dedicate a chi ha sottoscritto un piano a pagamento. Se l'utente non possiede i requisiti, viene automaticamente reindirizzato alla pagina di upgrade o, se non autenticato, alla pagina di login.

La guard è definita nel file src/app/guards/premium.guard.ts e può essere associata alle rotte desiderate tramite il sistema di routing dell'app.

**Funzionamento:**

* La guard implementa l'interfaccia CanActivate e si attiva ogni volta che un utente tenta di accedere a una rotta protetta.

**Passaggi logici:**

Recupera l'userId tramite il servizio AuthService.  
Se l'utente non è autenticato (userId assente):

Reindirizza automaticamente alla pagina di login.

Impedisce l'accesso alla rotta.

Se l'utente è autenticato:

Esegue una chiamata al backend per recuperare il tipo di account (userType).

Se il tipo è "premium" o "gold":

Permette l'accesso alla rotta.

In caso contrario:

Reindirizza alla pagina di upgrade.

Blocca l'accesso alla rotta.

**Gestione errori:**

In caso di errore nella verifica del tipo utente (problemi di rete o backend), l'utente viene reindirizzato alla pagina di login e l'accesso viene negato.

**3: NoGuestGuard**

Questa guard viene utilizzata per impedire agli utenti ospiti (guest) di accedere a determinate rotte riservate agli utenti registrati. Garantisce che solo chi ha effettuato un login standard o possiede un account premium/gold possa visualizzare alcune pagine o utilizzare funzioni specifiche. Gli utenti guest vengono reindirizzati alla pagina di login con un messaggio di avviso.

La guard è definita nel file src/app/guards/no-guest.guard.ts e si applica alle rotte che devono essere visibili solo a utenti autenticati in modo completo.

**Funzionamento:**

La guard implementa l'interfaccia CanActivate e si attiva ogni volta che un utente prova ad accedere a una rotta protetta.

**Passaggi logici:**

Recupera il tipo di utente tramite il metodo getCurrentUserType() del servizio AuthService.

Se il tipo utente è "guest":

Viene mostrato un messaggio di avviso tramite il ToastService ("Login richiesto").

L'utente viene reindirizzato alla pagina di login.

L'accesso viene bloccato, restituendo un oggetto UrlTree che forza la navigazione.

Se l'utente non è guest:

L'accesso alla rotta viene consentito.

**Back-end:**

Il back-end è ospitato in un server Expres che contine il DataBase usando SQLite e le varie API per comunicare con il Front-end.

Il DB è strutturato come raffigurato nel seguente modello ER:

**Users:**

* userId TEXT PRIMARY KEY,
* username TEXT NOT NULL,
* email TEXT NOT NULL UNIQUE,
* password TEXT NOT NULL,
* type TEXT NOT NULL DEFAULT 'standard',
* subscriptionEnd TEXT
* profileImage TEXT, registrationDate TEXT);

**Itineraries:**

* itineraryId TEXT PRIMARY KEY,
* userId TEXT,
* city TEXT NOT NULL,
* accommodation TEXT,
* startDate TEXT,
* endDate TEXT,
* style TEXT,
* coverPhoto TEXT,
* deleted INTEGER DEFAULT 0,
* FOREIGN KEY (userId) REFERENCES users(userId)

**Places:**

* placeId TEXT PRIMARY KEY,
* itineraryId TEXT NOT NULL,
* name TEXT NOT NULL,
* day INTEGER,
* timeSlot TEXT,
* lat REAL,
* lng REAL,
* address TEXT,
* photoUrl TEXT,
* type TEXT,
* note TEXT, photoFilename TEXT, rating REAL, priceLevel INTEGER, website TEXT, openingHours TEXT,
* FOREIGN KEY (itineraryId) REFERENCES itineraries(itineraryId)

**File: controllers/authController.js**

Questo controller gestisce tutte le funzionalità legate all'autenticazione e alla gestione degli utenti nell'applicazione SoTourist. Si occupa di registrazione, login, modifica delle informazioni, gestione abbonamenti e immagine profilo.

Il file importa i seguenti moduli:

generateId: Funzione custom per generare identificativi univoci degli utenti.

bcrypt: Libreria per la cifratura delle password.

downgradeIfExpired: Funzione che controlla eventuale scadenza dell'abbonamento e effettua downgrade automatico se necessario.

db: Modulo che gestisce la connessione al database SQLite.

Funzioni presenti

register(req, res) — Registrazione nuovo utente

Permette di registrare un nuovo utente. I passaggi sono:

Controlla la presenza di tutti i dati obbligatori (username, email, password).

Verifica se l'email è già presente nel database.

Cifra la password con bcrypt.

Genera un ID univoco per l'utente.

Se l'utente seleziona un piano premium o gold, imposta una data di fine abbonamento a 30 giorni.

Salva tutti i dati nel database.

updatePassword(req, res) — Modifica della password

Consente a un utente di modificare la propria password:

Recupera la password attuale dal database.

Verifica che la password attuale corrisponda a quella inserita.

Cifra la nuova password e la aggiorna nel database.

login(req, res) — Login utente

Gestisce l'accesso dell'utente:

Recupera i dati dell'utente tramite email.

Verifica la corrispondenza della password usando bcrypt.

Se necessario, effettua downgrade automatico in caso di abbonamento scaduto.

Restituisce le informazioni essenziali dell'utente al frontend.

deleteUser(req, res) — Eliminazione utente

Consente di eliminare un utente:

Recupera l'utente tramite userId.

Prima di eliminare, scollega gli itinerari dell'utente impostando il campo userId a NULL nella tabella itineraries.

Elimina definitivamente l'utente.

updateUser(req, res) — Modifica nome o email

Permette di aggiornare username e/o email:

Recupera i dati attuali dell'utente.

Aggiorna solo i campi forniti nella richiesta, mantenendo gli altri invariati.

upgradeToPremium(req, res) — Aggiornamento abbonamento

Consente di attivare un piano premium o gold:

Controlla che il piano richiesto sia valido.

Calcola la data di scadenza (30 giorni successivi).

Aggiorna tipo e data di scadenza nel database.

cancelPremium(req, res) — Annullamento abbonamento

Permette di annullare un abbonamento premium:

Controlla che l'utente non sia già standard.

Riporta il tipo a standard e azzera la data di scadenza.

getUserType(req, res) — Verifica tipo utente

Restituisce il tipo di abbonamento dell'utente:

Controlla eventuale scadenza dell'abbonamento.

Se necessario, esegue downgrade automatico.

Restituisce tipo e data di scadenza.

updateProfileImage(req, res) — Gestione immagine profilo

Gestisce il salvataggio o l'eliminazione dell'immagine profilo in formato base64:

Se viene passato un campo vuoto, elimina l'immagine.

Se viene passato un'immagine valida in base64, la salva nel database.

getProfileImage(req, res) — Recupero immagine profilo

Recupera l'immagine profilo dell'utente:

Se non esiste, restituisce null.

Se esiste, restituisce l'immagine in base64.

getRegistrationDate(req, res) — Data di registrazione

Restituisce la data di registrazione dell'utente.

Considerazioni

Il controller utilizza una combinazione di callback e funzioni asincrone async/await. I controlli sugli errori sono presenti in tutte le funzioni per garantire stabilità e corretto flusso delle operazioni. Inoltre, la gestione degli abbonamenti prevede un meccanismo automatico di downgrade se la data di scadenza viene superata, garantendo coerenza tra i dati lato server e l'interfaccia utente.

**File: controllers/itineraryController.js**

Questo controller rappresenta il **cuore del backend** dell'applicazione SoTourist. Si occupa di generare dinamicamente l'itinerario giornaliero dell'utente in base a una serie di parametri personalizzabili e vincoli geografici. Grazie all'integrazione con le API di Google Maps, la logica di costruzione tiene conto di:

* Lo stile turistico scelto dall'utente (ad es. FoodTour, Shopping, Standard...)
* Il mezzo di trasporto preferito (a piedi, auto, bici, bus)
* I luoghi da evitare (avoid)
* I luoghi obbligatori (must-see e must-eat)
* La posizione dell'alloggio, se fornita

**Funzioni presenti**

**getItinerary(req, res)**

**Generazione dinamica dell'itinerario completo**

**Parametri in ingresso:**

* city: Città selezionata dall'utente
* totalDays: Numero di giorni dell'itinerario
* accommodation: Indirizzo o nome dell'alloggio (facoltativo)
* transport: Mezzo di trasporto scelto (walk, car, bike, bus)
* style: Stile dell'itinerario (Standard, Museums, Shopping, FoodTour)
* mustSee: Array di placeId che devono obbligatoriamente essere inseriti
* mustEat: Array di placeId di luoghi dove mangiare da inserire con priorità
* avoid: Array di placeId da evitare durante la generazione

**Funzionamento dettagliato:**

**1. Recupero dati e configurazioni**

* Viene recuperata l'immagine di copertina della città (se presente in locale) tramite photoManager.
* Viene calcolato il centro città con le API di geocodifica Google.
* Vengono definiti i parametri di distanza tra tappe in base al mezzo scelto:
  + **walk**: massimo 1km tra tappe
  + **car**: fino a 10km
  + **bike**: fino a 5km
  + **bus**: fino a 10km

**2. Gestione alloggio**

Se viene fornito un alloggio:

* Si effettua una ricerca precisa tramite l'API findplacefromtext.
* Se trovato, viene salvato come luogo "speciale" con tipo accommodation.
* Le coordinate dell'alloggio diventano l'ancora di partenza per le tappe.

**3. Chaining dinamico delle tappe**

Il meccanismo di **chaining dinamico** garantisce che ogni tappa successiva venga generata o selezionata in relazione:

* Alla posizione della tappa precedente
* Al vincolo di distanza impostato dal mezzo
* Ai luoghi già utilizzati (used) o da evitare (avoidSet)

Questo previene duplicati e assicura coerenza spaziale nell'itinerario.

Ogni giornata è suddivisa in 3 slot:

* morning
* afternoon
* evening

Per ogni slot viene generato l'elenco delle tappe seguendo lo **stile** selezionato, che determina:

* Quali tipi di luoghi cercare
* Quanti luoghi per slot
* Se priorità su attrazioni culturali, shopping, gastronomia, ecc.

La generazione segue questa logica:

* Se il tipo è "eat" e ci sono ancora mustEat disponibili, vengono inseriti con priorità.
* Altrimenti, si esegue una ricerca tramite nearbysearch su Google Places filtrando:
  + distanza da anchor (tappa precedente o alloggio)
  + tipo di luogo
  + keyword specifica dello stile
  + esclusione dei luoghi già usati o evitati
* Se nessun risultato è disponibile, viene inserito un luogo placeholder.

Ogni nuova tappa diventa la nuova "anchor" per la successiva, creando un percorso coerente e realistico.

**4. MustSee e inserimento ottimale**

Dopo aver generato tutte le giornate:

* Si analizzano i luoghi must-see.
* Per ciascuno si calcola, **matematicamente**, il miglior punto d'inserimento nell'itinerario:
  + Si valutano tutte le possibili posizioni tra le tappe già presenti.
  + Si calcola il "delta distanza" generato inserendo il nuovo luogo.
  + Si sceglie la posizione che minimizza il percorso complessivo.

In questo modo i must-see vengono inseriti senza compromettere il realismo o la coerenza spaziale dell'itinerario.

**5. Ritorno all'alloggio**

Se è presente un alloggio, l'ultima tappa della sera viene forzata come ritorno all'alloggio, chiudendo il percorso giornaliero in modo logico.

**getSinglePlace(req, res)**

**Ricerca puntuale di un singolo luogo**

Questa funzione permette al frontend di cercare un luogo specifico da inserire nell'itinerario:

* Utilizza le coordinate o il centro città come punto di riferimento.
* Effettua una ricerca tramite Google Places.
* Recupera i dettagli completi con foto e coordinate.
* Scarica e salva in locale l'immagine, se disponibile.

**Considerazioni finali**

Grazie a questo controller:

* L'itinerario è **altamente personalizzato**, dinamico e contestuale.
* Le scelte dell'utente (mezzo, stile, luoghi da evitare, preferenze) impattano realmente sulla generazione.
* Viene garantito un percorso **realistico e coerente**, simulando un'esperienza turistica plausibile.
* La presenza di fallback (luoghi placeholder) garantisce robustezza anche in caso di dati mancanti. ( ancora da inserire un fall back intelligente )
* L'inserimento ottimizzato dei must-see è uno degli aspetti più avanzati e intelligenti della logica.

**File: controllers/tripController.js**

Il controller **tripController.js** gestisce tutte le operazioni relative agli itinerari degli utenti nell'applicazione SoTourist. Si occupa della creazione, modifica, eliminazione, recupero e duplicazione degli itinerari, oltre alla gestione delle tappe associate.

**Moduli utilizzati:**

* generateId: per generare ID univoci per itinerari e tappe.
* db: connessione al database SQLite.
* checkOverlap: funzione utility che verifica la sovrapposizione tra date.
* getOrDownloadPhoto, getCityCoverPhoto: gestiscono il recupero o il download delle immagini legate alle città e ai luoghi.

**Funzioni presenti**

**1. getItineraries(req, res)**

Restituisce la lista degli itinerari di un utente, con possibilità di filtrarli:

* all: tutti gli itinerari attivi.
* current: itinerari attivi oggi.
* upcoming: il prossimo itinerario futuro.
* future: tutti gli itinerari futuri, eccetto il prossimo.
* past: itinerari passati.

I dati vengono filtrati e ordinati in base alla data attuale.

**2. addItinerary(req, res)**

Crea un nuovo itinerario per un utente:

* Verifica che non ci siano sovrapposizioni di date con altri itinerari attivi.
* Recupera un'immagine di copertina della città tramite getCityCoverPhoto.
* Salva i dati nel database con un ID univoco.

Controlla parametri obbligatori: city, startDate, endDate.

**3. deleteItinerary(req, res)**

Elimina logicamente un itinerario, impostando il campo deleted a 1.

**4. getItinerariesByCity(req, res)**

Restituisce tutti gli itinerari pubblici (non eliminati) per una determinata città, utile per la ricerca globale o l'esplorazione.

**5. getItineraryById(req, res)**

Recupera un itinerario completo tramite ID:

* Restituisce anche le tappe, suddivise in:
  + morning
  + afternoon
  + evening
  + ordered (ordine cronologico completo)

Le tappe vengono convertite in un formato più leggibile per il frontend, includendo dettagli come coordinate, foto, tipo, note.

**6. updateItinerary(req, res)**

Modifica un itinerario esistente:

* Controlla che l'itinerario esista e appartenga all'utente.
* Se vengono fornite nuove date, verifica che non si sovrappongano ad altri itinerari.
* Applica le modifiche su città, date, stile, alloggio, copertina.

**7. addPlacesToItinerary(req, res)**

Aggiunge una o più tappe a un itinerario:

* Supporta invio singolo o array di tappe.
* Ogni tappa riceve un ID univoco.
* Se presente photoReference, scarica l'immagine tramite getOrDownloadPhoto.
* Salva nel database tutti i dettagli delle tappe.

**8. checkDateOverlap(req, res)**

API che permette al frontend di verificare se un nuovo itinerario si sovrappone ad altri esistenti.

Utile per controlli lato frontend prima della creazione o modifica.

**9. updateItineraryPlaces(req, res)**

Sovrascrive tutte le tappe di un itinerario:

* Elimina tutte le tappe attuali.
* Inserisce quelle nuove fornite nel body.
* Utilizza una prepared statement per efficienza e sicurezza.

**10. copyItinerary(req, res)**

Duplica un itinerario su un altro utente o su un'altra data:

* Verifica la validità delle nuove date.
* Duplica l'itinerario mantenendo città, stile e copertina.
* Copia solo le tappe che rientrano nella nuova durata.
* Genera un nuovo itineraryId per il duplicato.

**Considerazioni**

Questo controller rappresenta il **gestore centrale della persistenza e manipolazione degli itinerari**, assicurando:

* Integrità delle date (nessuna sovrapposizione).
* Corretta associazione tra utenti e itinerari.
* Gestione completa delle tappe, incluse immagini e dati aggiuntivi.
* Possibilità di duplicare itinerari, utile per creare bozze o modelli predefiniti.

Le operazioni sul database vengono eseguite tramite SQLite in modalità sincrona o asincrona, garantendo performance adeguate per un'app mobile.

**Aspetti avanzati:**

* **Gestione immagini**: Viene integrato il download automatico delle immagini dei luoghi, riducendo il carico del frontend.
* **Filtri dinamici**: La funzione getItineraries consente filtri lato server, migliorando l'efficienza lato client.
* **Duplicazione intelligente**: La funzione copyItinerary considera la durata effettiva, evitando errori legati a tappe fuori dal range.

**Cartella: db**

Questa cartella contiene i file responsabili della configurazione, inizializzazione e gestione della connessione al database SQLite dell'applicazione SoTourist.

**3. File: database.db**

Questo è il file fisico SQLite che contiene i dati reali salvati dall'applicazione.

* Viene generato automaticamente all'avvio del server se non esiste.
* Contiene in formato binario tutte le tabelle e i dati strutturati definiti in init.js.
* È un file locale e portabile, perfetto per prototipi e applicazioni che non richiedono un server database esterno.

**Considerazioni finali**

L'organizzazione della cartella db offre una gestione semplice, portabile e robusta della persistenza dati lato backend. La scelta di SQLite garantisce:

* **Semplicità di installazione** (nessun server esterno richiesto).
* **Compatibilità multipiattaforma**.
* **Prestazioni adeguate** per progetti prototipali e applicazioni mobile con backend leggero.

Grazie a init.js è possibile ricreare facilmente la struttura del database in caso di reset o reinstallazione, rendendo l'applicazione rapidamente avviabile anche su nuovi dispositivi o ambienti.

**1. File: connection.js**

Questo file si occupa di creare e gestire la connessione principale al database SQLite.

**Funzionamento:**

* Importa il modulo sqlite3 in modalità verbose, che fornisce log utili in fase di debug.
* Costruisce il percorso assoluto del file database.db utilizzando path.
* Crea un'istanza del database, che viene poi esportata per essere utilizzata in tutto il backend.

**Vantaggi:**

* Centralizza la connessione al database, evitando duplicazione di codice.
* Grazie al percorso assoluto, il file database.db viene sempre trovato correttamente, indipendentemente dalla posizione da cui viene eseguito il server.

**2. File: init.js**

Questo file inizializza la struttura del database se non ancora presente.

**Funzionamento:**

* Si collega al database database.db.
* In modalità serialize (esecuzione sincrona delle query), crea le tabelle necessarie utilizzando CREATE TABLE IF NOT EXISTS, così da evitare duplicazioni o errori se il database esiste già.

**Tabelle create:**

1. **users**
   * Contiene i dati degli utenti.
   * Campi principali:
     + userId: identificativo univoco.
     + username, email, password: dati base per autenticazione.
     + type: tipo utente (standard, premium, gold...).
     + subscriptionEnd: data fine abbonamento.
     + profileImage: immagine profilo in formato base64.
2. **itineraries**
   * Rappresenta gli itinerari creati dagli utenti.
   * Campi principali:
     + itineraryId: identificativo univoco.
     + userId: può essere NULL in caso di itinerari pubblici o non associati.
     + city, accommodation, startDate, endDate, style, coverPhoto.
     + deleted: flag logico per eliminazioni soft.
   * È presente una chiave esterna verso la tabella users.
3. **places**
   * Contiene le tappe associate agli itinerari.
   * Campi principali:
     + placeId: identificativo univoco.
     + itineraryId: associazione all'itinerario.
     + name, day, timeSlot, lat, lng, address, photoUrl, type, note.
   * È presente una chiave esterna verso la tabella itineraries.

**Nota:** La struttura del database è progettata per essere semplice ma completa, garantendo integrità dei dati grazie alle chiavi esterne.

Alla fine, il database viene chiuso con db.close(), poiché questo file deve solo occuparsi della creazione delle tabelle.

**File: routes/auth.js**

Questo file definisce le rotte API relative alla **gestione degli utenti e dell'autenticazione** nell'applicazione SoTourist.

Viene utilizzato il modulo express.Router() per organizzare in modo ordinato gli endpoint, separando la definizione delle rotte dalla logica implementata nei controller.

**Importazioni principali:**

* express: framework backend utilizzato.
* authController: contiene tutta la logica delle operazioni sugli utenti.

**Rotte definite**

**Autenticazione**

| **Metodo** | **Percorso** | **Descrizione** | **Funzione Controller** |
| --- | --- | --- | --- |
| POST | /register | Registrazione utente | register |
| POST | /login | Login utente | login |

**Gestione utente**

| **Metodo** | **Percorso** | **Descrizione** | **Funzione Controller** |
| --- | --- | --- | --- |
| DELETE | /users/:userId | Elimina utente | deleteUser |
| PUT | /users/:userId | Modifica username o email | updateUser |
| PUT | /users/:userId/password | Modifica password | updatePassword |
| PUT | /users/:userId/profile-image | Salva o elimina immagine profilo | updateProfileImage |
| GET | /users/:userId/profile-image | Recupera immagine profilo | getProfileImage |
| GET | /users/:userId/registration-date | Recupera data di registrazione | getRegistrationDate |

**Gestione abbonamenti**

| **Metodo** | **Percorso** | **Descrizione** | **Funzione Controller** |
| --- | --- | --- | --- |
| POST | /users/:userId/upgrade | Aggiorna utente a premium o gold | upgradeToPremium |
| POST | /users/:userId/cancel | Annulla abbonamento premium/gold | cancelPremium |
| GET | /users/:userId/type | Recupera tipo utente e verifica downgrade | getUserType |

**Considerazioni**

Questo file si limita a:

* Definire in modo chiaro e leggibile tutte le rotte legate agli utenti.
* Collegare ogni rotta alla relativa funzione del controller.
* Mantenere il codice organizzato e facilmente estendibile.

Seguendo la struttura RESTful, le operazioni sono coerenti con le convenzioni standard:

* POST per creazione o azioni specifiche.
* GET per recupero dati.
* PUT per modifiche.
* DELETE per eliminazione.

**File: routes/itinerary.js**

Questo file definisce le rotte API relative alla generazione automatica degli itinerari turistici, una delle funzionalità più importanti e distintive dell'applicazione SoTourist.

Viene utilizzato il modulo express.Router() per organizzare le rotte collegate al controller itineraryController, che contiene la logica avanzata di generazione e ricerca dei luoghi.

Importazioni principali:

* express: framework backend utilizzato.
* { getItinerary, getSinglePlace }: funzioni del controller che gestiscono la generazione dell'itinerario e la ricerca di un singolo luogo.

Rotte definite

| Metodo | Percorso | Descrizione | Funzione Controller |
| --- | --- | --- | --- |
| POST | / | Genera un nuovo itinerario dinamico partendo da payload JSON | getItinerary |
| GET | / | Genera un itinerario tramite parametri in query string *(per compatibilità o test)* | getItinerary |
| GET | /single-place | Ricerca dettagliata di un singolo luogo | getSinglePlace |

Dettagli tecnici

POST /

Riceve un payload JSON strutturato contenente informazioni su:

Città

Stile desiderato

Mezzo di trasporto

Giorni totali

Preferenze utente (must-see, must-eat, avoid)

Alloggio

Passa i dati al controller, che genera dinamicamente l'itinerario personalizzato.

GET /

Permette di generare l'itinerario passando i parametri nella query string.

Utile per test rapidi o compatibilità con richieste più semplici.

La logica di generazione è identica a quella del metodo POST.

GET /single-place

Permette di ricercare un luogo specifico partendo da un nome o una query testuale.

Utilizza la posizione (coordinate o centro città) per contestualizzare la ricerca.

Recupera tutti i dettagli del luogo, incluse immagini e coordinate precise.

Considerazioni

Questa rotta costituisce l'interfaccia tra il frontend e la logica avanzata di generazione degli itinerari:

Permette di creare percorsi turistici su misura basati sulle preferenze dell'utente.

Grazie al supporto sia per POST che per GET, offre flessibilità nelle modalità di invio dei dati.

Mantiene il codice del server organizzato, separando chiaramente la definizione delle rotte dalla logica implementativa.

File: routes/trip.js

Questo file definisce tutte le rotte RESTful relative alla gestione degli itinerari nell'applicazione SoTourist. Le operazioni coprono l'intero ciclo di vita di un itinerario: creazione, modifica, eliminazione, recupero, gestione delle tappe e duplicazione.

Il file si basa sul modulo express.Router() per organizzare in modo ordinato gli endpoint, separando la definizione delle rotte dalla logica presente nel tripController.

Importazioni principali:

* express: framework backend utilizzato.
* tripController: contiene tutta la logica delle operazioni sugli itinerari e le tappe.

Rotte definite

Gestione Itinerari di un utente

| Metodo | Percorso | Descrizione | Funzione Controller |
| --- | --- | --- | --- |
| GET | /users/:userId/itineraries | Recupera tutti gli itinerari di un utente, con filtri | getItineraries |
| POST | /users/:userId/itineraries | Crea un nuovo itinerario | addItinerary |
| DELETE | /users/:userId/itineraries/:itineraryId | Elimina un itinerario specifico | deleteItinerary |
| PUT | /users/:userId/itineraries/:itineraryId | Modifica un itinerario esistente | updateItinerary |

Gestione tappe di un itinerario

| Metodo | Percorso | Descrizione | Funzione Controller |
| --- | --- | --- | --- |
| POST | /users/:userId/itineraries/:itineraryId/places | Aggiunge una o più tappe a un itinerario | addPlacesToItinerary |
| PUT | /users/:userId/itineraries/:itineraryId/places | Sovrascrive tutte le tappe dell'itinerario | updateItineraryPlaces |

Gestione globale itinerari (tra più utenti)

| Metodo | Percorso | Descrizione | Funzione Controller |
| --- | --- | --- | --- |
| GET | /itineraries | Recupera itinerari pubblici per città | getItinerariesByCity |
| GET | /itineraries/:itineraryId | Recupera un singolo itinerario completo tramite ID | getItineraryById |
| POST | /itineraries/:itineraryId/copy/:userId | Duplica un itinerario su un altro utente | copyItinerary |

Controllo sovrapposizione date

| Metodo | Percorso | Descrizione | Funzione Controller |
| --- | --- | --- | --- |
| GET | /users/:userId/itineraries/check-overlap | Verifica sovrapposizioni di date con altri itinerari | checkDateOverlap |

Considerazioni

Questa rotta rappresenta l'interfaccia completa tra il frontend e la gestione strutturata degli itinerari. Permette di:

* Creare, modificare o eliminare itinerari in modo sicuro.
* Gestire in modo flessibile le tappe.
* Evitare conflitti temporali tra itinerari grazie ai controlli sulle date.
* Condividere itinerari tra utenti con il meccanismo di copia.
* Recuperare facilmente dati utili per funzionalità come:
  + Visualizzazione itinerari pubblici per città.
  + Mostrare l'itinerario dettagliato con tutte le tappe.

L'organizzazione segue lo standard REST, rendendo il backend facilmente comprensibile, scalabile ed estensibile.

File: services/photoManager.js

Questo modulo si occupa della gestione avanzata delle immagini dei luoghi e delle città, integrando le API di Google Places e ottimizzando il salvataggio locale delle immagini per ridurre le chiamate ripetute al server Google.

Grazie a questo servizio, il backend:

* Recupera dinamicamente le foto dei luoghi e delle città.
* Salva le immagini localmente nella cartella /uploads.
* Riutilizza le immagini già scaricate, migliorando le performance e riducendo i costi delle API.

Funzioni presenti

1. ensureDir(dir)

Verifica che una cartella esista, altrimenti la crea.

Utilizzo interno per garantire che le sottocartelle come /uploads/places o /uploads/covers siano sempre disponibili.

2. getOrDownloadPhoto(fileId, photoReference, subfolder = 'places')

Scarica la foto di un luogo da Google Places e la salva in locale, solo se non già presente.

Parametri:

* fileId: Identificativo da usare per il nome del file (solitamente placeId o itineraryId).
* photoReference: Riferimento univoco della foto ottenuto dalle API di Google Places.
* subfolder: Sottocartella di salvataggio, predefinita su places, può essere covers per le immagini città.

Funzionamento:

* Costruisce il percorso locale dell'immagine.
* Se l'immagine è già presente in locale, la restituisce immediatamente.
* Se non presente:
  + Esegue una richiesta verso le API di Google per scaricare l'immagine.
  + La salva come file .jpg nella sottocartella indicata.
  + Restituisce il percorso relativo al file salvato.

Vantaggi:

* Riduce le chiamate verso Google.
* Garantisce caricamenti più rapidi per le immagini nel frontend.
* Mantiene organizzate le immagini per tipo (places per i luoghi, covers per le copertine delle città).

3. getCityCoverPhoto(rawCity)

Ricerca e scarica un'immagine rappresentativa per la città indicata.

Funzionamento:

* Esegue una ricerca testuale tramite le API di Google Places con la query "Monumenti famosi a [città]".
* Se tra i risultati è presente almeno un luogo con una foto disponibile:
  + Recupera il placeId e il photoReference.
  + Utilizza getOrDownloadPhoto per scaricare e salvare l'immagine nella sottocartella covers.
  + Restituisce il percorso dell'immagine salvata.
* Se non viene trovata un'immagine adatta, restituisce null.

Considerazioni

Questo servizio è fondamentale per migliorare l'esperienza utente:

* Garantisce che le immagini dei luoghi e delle città siano disponibili anche offline o in caso di problemi temporanei con le API.
* Riduce il carico e i costi derivanti da chiamate ripetute a Google Places.
* Mantiene separata e ben organizzata la gestione delle immagini rispetto alla logica principale dei controlle

**Cartella: uploads**

La cartella **uploads** contiene tutte le immagini utilizzate dall'applicazione SoTourist, salvate in locale per ridurre il numero di chiamate verso le API di Google e migliorare le prestazioni.

Viene creata automaticamente dal backend al momento del salvataggio di nuove immagini, tramite il servizio photoManager.js.

**Struttura interna**

All'interno della cartella si trovano due sottocartelle principali:

**1. uploads/covers**

Contiene le **immagini di copertina degli itinerari**, ovvero le foto rappresentative della città scelta dall'utente.

**Caratteristiche:**

* Le immagini vengono scaricate tramite la funzione getCityCoverPhoto di photoManager.js.
* Ogni immagine ha come nome il placeId del luogo da cui è stata recuperata, seguito dall'estensione .jpg.
  + Esempio:  
    uploads/covers/ChIJyWEHuEmuEmsRm9hTkapTCrk.jpg
* Se la stessa città viene scelta più volte o in più itinerari, la foto viene **riutilizzata**, evitando nuove richieste alle API di Google.
* Questo approccio riduce il traffico verso Google e velocizza il caricamento delle immagini.

**2. uploads/places**

Contiene le **foto dei singoli luoghi** inseriti negli itinerari, come attrazioni turistiche, ristoranti o punti di interesse.

**Caratteristiche:**

* Le immagini vengono scaricate tramite la funzione getOrDownloadPhoto di photoManager.js.
* Anche in questo caso, il nome di ogni immagine corrisponde al placeId del luogo seguito da .jpg.
  + Esempio:  
    uploads/places/ChIJN1t\_tDeuEmsRUsoyG83frY4.jpg
* Se lo stesso luogo viene inserito in più itinerari o compare più volte nella generazione automatica, la foto viene recuperata direttamente dalla cartella, senza effettuare nuove chiamate verso Google.

**Vantaggi di questo approccio**

* **Riduzione drastica delle chiamate alle API Google**, con conseguente risparmio di costi e rispetto dei limiti giornalieri.
* **Prestazioni migliorate**, in quanto le immagini vengono caricate da disco locale invece che da internet.
* **Riutilizzo intelligente**, basato sull'ID univoco fornito da Google, che garantisce che ogni foto corrisponda sempre allo stesso luogo.

**Considerazioni finali**

Questa organizzazione consente al backend di SoTourist di gestire in modo efficiente le immagini necessarie all'applicazione, migliorando l'esperienza utente e rendendo il sistema più performante e scalabile.

**File: utils/dateUtils.js**

Questo file contiene funzioni di utilità per la gestione e il controllo delle date, utilizzate principalmente dal backend per garantire l'integrità degli itinerari e prevenire sovrapposizioni temporali.

**Funzioni presenti**

**1. datesOverlap(start1, end1, start2, end2)**

**Scopo:**  
Verifica se due intervalli di date si sovrappongono.

**Parametri:**

* start1, end1: date di inizio e fine del primo intervallo.
* start2, end2: date di inizio e fine del secondo intervallo.

**Restituisce:**  
true se c'è sovrapposizione tra i due intervalli, false altrimenti.

**Funzionamento:**

* Converte tutte le date in oggetti Date per confrontarle correttamente.
* Verifica che l'inizio del primo intervallo sia precedente o uguale alla fine del secondo **e** che la fine del primo intervallo sia successiva o uguale all'inizio del secondo.

**2. checkOverlap(startDate, endDate, itineraries, excludeId = null)**

**Scopo:**  
Controlla se un nuovo intervallo di date si sovrappone ad almeno un itinerario esistente.

**Parametri:**

* startDate, endDate: date del nuovo intervallo da controllare.
* itineraries: array di oggetti itinerario già presenti nel sistema.
* excludeId *(opzionale)*: ID di un itinerario da escludere dal controllo (utile quando si modifica un itinerario esistente).

**Restituisce:**  
true se viene trovata una sovrapposizione, false altrimenti.

**Funzionamento:**

* Scorre l'array degli itinerari.
* Se viene fornito un excludeId, salta l'itinerario con quell'ID.
* Per gli altri itinerari, usa la funzione datesOverlap per verificare la sovrapposizione.

**Considerazioni**

Questo modulo è un esempio di buona pratica di separazione delle responsabilità:

* Isola la logica delle date in un file dedicato.
* Rende il codice dei controller più leggibile e riutilizzabile.
* Migliora la manutenibilità e facilita eventuali estensioni future.

L'uso di queste funzioni è fondamentale nel backend per:

* Evitare la creazione di itinerari con date sovrapposte.
* Garantire la coerenza temporale dei dati.
* Eseguire controlli robusti sia in fase di creazione che di modifica degli itinerari.

**File: utils/idGenerator.js**

Questo file contiene una funzione di utilità per la generazione di identificativi univoci (ID), utilizzati in tutto il backend per garantire che ogni nuova entità creata (utente, itinerario, tappa) abbia un identificativo sicuro e non duplicabile.

**Funzione presente**

**generateId(prefix = '')**

**Scopo:**  
Genera un ID univoco, composto da un prefisso personalizzabile, un timestamp e una parte casuale.

**Parametri:**

* prefix *(opzionale)*: stringa da anteporre all'ID, utile per distinguere il tipo di entità (es. "user\_", "trip\_", "place\_").

**Restituisce:**  
Una stringa alfanumerica univoca, ad esempio:  
trip\_171947987251310345

**Funzionamento dettagliato**

* **Parte casuale:**  
  Viene generato un numero casuale compreso tra 0 e 99.999, riducendo ulteriormente la possibilità di duplicati, anche in caso di generazioni multiple nello stesso istante.
* **Timestamp:**  
  Viene aggiunto il valore restituito da Date.now(), che rappresenta il numero di millisecondi trascorsi dal 1 gennaio 1970. Questo garantisce unicità temporale.
* **Prefisso:**  
  Permette di distinguere facilmente il tipo di entità a cui appartiene l'ID, migliorando la leggibilità e la tracciabilità.

**Esempi di utilizzo**

* Per creare un nuovo utente:  
  generateId('user\_') → user\_171947987251356
* Per un nuovo itinerario:  
  generateId('trip\_') → trip\_171947987251378
* Per una nuova tappa:  
  generateId('place\_') → place\_171947987251398

**Considerazioni**

Questa funzione è una soluzione semplice ma efficace per la generazione di ID univoci all'interno del progetto:

* Evita conflitti tra entità diverse.
* Non richiede l'utilizzo di librerie esterne.
* Garantisce buona probabilità di unicità anche in ambienti multiutente o con richieste ravvicinate.

**File: utils/subscriptionChecker.js**

Questo modulo fornisce una funzione di utilità per la gestione automatica dello stato di abbonamento degli utenti all'interno dell'applicazione SoTourist.

**Funzione presente**

**downgradeIfExpired(user)**

**Scopo:**  
Verifica se l'abbonamento di un utente è scaduto e, in tal caso, lo riporta automaticamente allo stato standard.

**Funzionamento dettagliato**

* La funzione accetta in ingresso un oggetto user che rappresenta l'utente, con almeno i seguenti campi:
  + type: tipo di abbonamento (standard, premium, gold, ecc.)
  + subscriptionEnd: data di scadenza dell'abbonamento (formato YYYY-MM-DD)
* Se il tipo di utente è diverso da standard e:
  + la data di scadenza non è presente, oppure
  + la data di scadenza è antecedente alla data odierna,

allora:

* + il tipo di utente viene impostato su standard
  + la data di scadenza viene azzerata (null)
* La funzione restituisce l'oggetto user, modificato se necessario.

**Considerazioni**

Questa funzione è utilizzata in modo trasparente in diverse parti del backend, in particolare nel controller degli utenti (authController), per:

* Verificare lo stato di abbonamento ogni volta che un utente effettua il login o richiede il tipo.
* Garantire che le informazioni sull'abbonamento siano sempre aggiornate e corrette senza richiedere azioni manuali o cron job separati.

In questo modo, il sistema gestisce in autonomia il downgrade degli utenti quando il loro periodo di abbonamento scade.

**File: index.js**

**File principale di avvio del server Backend**

Il file **index.js** costituisce il punto di ingresso del server backend dell’applicazione SoTourist. Si occupa della configurazione iniziale del server Express, della gestione delle rotte e dell’avvio del server in ascolto.

**Funzioni principali svolte dal file:**

* Caricamento delle variabili di ambiente tramite il file .env.
* Configurazione del framework Express.
* Gestione delle immagini salvate in locale nella cartella uploads.
* Configurazione delle politiche CORS per consentire al frontend di comunicare con il backend.
* Definizione delle rotte principali che espongono le API verso il frontend.
* Avvio del server in ascolto sulla porta specificata.

**Descrizione dettagliata:**

**1. Gestione variabili ambiente:**  
Il file utilizza il pacchetto dotenv per caricare in modo sicuro le variabili contenute nel file .env, come ad esempio la chiave API di Google o la porta su cui far partire il server.

**2. Configurazione Express:**  
Viene creata un’istanza dell’app Express che gestirà tutte le richieste in arrivo.

**3. Gestione immagini statiche:**  
Viene definita una regola per rendere accessibile la cartella uploads dal frontend. Questo permette di visualizzare correttamente le immagini dei luoghi e delle città, precedentemente scaricate e salvate dal backend.

**4. Configurazione CORS:**  
È abilitata la politica CORS aperta, che permette al frontend (sia mobile che web) di inviare richieste al server, anche se si trova su un dominio o una porta diversa.

**5. Gestione dei payload:**  
Viene configurato il server per accettare richieste con corpo in formato JSON o URL encoded, con un limite di dimensione elevato per poter gestire anche immagini in formato base64.

**6. Definizione delle rotte API:**  
Il server monta i diversi gruppi di rotte:

* /api/itinerary: per la generazione automatica degli itinerari.
* /api/auth: per la gestione degli utenti e dell’autenticazione.
* /api: per tutte le operazioni sugli itinerari e le tappe.

**7. Avvio del server:**  
Alla fine del file viene avviato il server, che si mette in ascolto sulla porta specificata nelle variabili di ambiente o, in assenza di configurazione, sulla porta 3000 in locale.

**Considerazioni:**

Il file index.js rappresenta il cuore della configurazione del backend. Permette di avviare il server in modo semplice e ordinato, integrando la gestione dei file statici, delle rotte e delle configurazioni generali necessarie per il corretto funzionamento dell’applicazione SoTourist.

Grazie a questa struttura, il backend è modulare, facilmente mantenibile e pronto a comunicare correttamente con il frontend mobile o web.