



# Adventure Together

**Prova Finale di Ingegneria del Software**

**Politecnico di Milano**

*Codice Persona:* 10966014

*Matricola:* 241414

*Autore:* Luca Moretti

# Indice

Info.....	3
Dominio Applicativo .....	3
Contesto .....	3
Sintesi delle funzionalità .....	3
Dettaglio funzionalità per categoria utente.....	4
Architettura .....	5
Architettura tecnologia applicazione .....	5
Gestione dei dati (Database).....	5
Componenti aggiuntivi – Stripe .....	5
Componenti aggiuntivi – Java Mail Sender.....	6
Gestione Sicurezza .....	6
Use Case Diagram.....	8
Package Diagram.....	9
Class Diagram – package Model.....	11
Class Diagram – package Repository .....	15
Class Diagram – package Service .....	16
Class Diagram – package DTO.....	18
Class Diagram – package Controller.....	19
Class Diagram – packages Util, Scheduler, Config.....	21
Design Pattern – State Pattern and Observer Pattern .....	23
Design Pattern – Decorator Pattern.....	27
Casi Rilevanti – Booking process .....	29
Template Thymeleaf .....	34
Application Properties e Environment Variables .....	35
Dipendenze Maven .....	35
Piano di Test .....	35
Strategia di testing .....	35
Strumenti usati.....	36
Esempi di Test .....	36
Sintesi dei Test e copertura .....	39

## Info

*Prova Finale di Ingegneria del Software*

*Laurea Triennale in Ingegneria Informatica - Politecnico di Milano*

*Nome progetto:* AdventureTogether

*Codice persona:* 10966014

*Matricola:* 241414

*Autore:* Luca Moretti

*Link all'archivio GitHub:* [https://github.com/lucamoretti155/adventure\\_together](https://github.com/lucamoretti155/adventure_together)

## Dominio Applicativo

### Contesto

Organizzare viaggi di gruppo e itinerari personalizzati è un'attività complessa che richiede coordinamento, comunicazione e strumenti digitali adeguati. Negli ultimi anni, il turismo esperienziale ha visto crescere l'interesse verso viaggi condivisi, dove persone con passioni comuni si incontrano per vivere avventure organizzate. Tuttavia, la gestione di questi processi è spesso frammentata: gli organizzatori utilizzano fogli di calcolo, chat non strutturate e sistemi di pagamento separati, mentre i viaggiatori faticano a trovare proposte affidabili e a gestire prenotazioni in modo sicuro.

Piattaforme come StoGranTour e ViaggiAvventura hanno già introdotto un modello efficace, creando spazi digitali dove gli organizzatori possono proporre itinerari e i viaggiatori possono iscriversi. AdventureTogether si colloca in questo contesto, adottando un approccio simile e consolidato, con l'obiettivo di offrire un'esperienza completa e integrata per entrambe le parti. La piattaforma consente di gestire in modo centralizzato tutte le fasi del processo: dalla creazione dell'itinerario alla pubblicazione dei viaggi, fino alla prenotazione e alle recensioni post-viaggio.

AdventureTogether si propone quindi come una soluzione che segue un modello già validato nel settore, offrendo un ambiente unico, semplice e sicuro, che riduce le complessità operative e favorisce la trasparenza, migliorando l'esperienza di chi viaggia e di chi organizza.

### Sintesi delle funzionalità

L'elemento cardine della piattaforma è rappresentato dall'Itinerario di Viaggio (**TripItinerary**), che costituisce il modello di riferimento per ogni proposta di viaggio. Ogni Itinerario include un titolo, una descrizione e un'immagine rappresentativa oltre a dettagli quali la durata complessiva, le tappe giornaliere, gli aeroporti di partenza, il numero minimo e massimo di partecipanti, le categorie tematiche, i Paesi coinvolti (e loro Area Geografica).

A partire dall'Itinerario, vengono generati i viaggi (**Trip**), ossia le concretizzazioni effettive dei viaggi con date di partenza e ritorno specifiche, finestre temporali per le prenotazioni e costi definiti.

Le prenotazioni, gestite attraverso l'entità **Booking**, permettono ai viaggiatori di selezionare un **Trip**, indicare i partecipanti, scegliere un'assicurazione e completare il pagamento in modo sicuro. In base al numero di partecipanti e allo stato delle prenotazioni, i **Trip** possono transitare tra diversi stati a seconda delle circostanze e delle decisioni amministrative.

Gli attori principali sono:

- **Guest**

Gli utenti non autenticati possono accedere all'offerta completa di itinerari disponibili in piattaforma, consultano i dettagli dei vari itinerari, le date di svolgimento, i prezzi e le recensioni. Possono infine decidere di registrarsi come Traveler inserendo le info richieste.

- **Traveler**

Il viaggiatore accede alla piattaforma per consultare i viaggi disponibili, filtrandoli per categoria, Paese o area geografica. Può visualizzare i dettagli di un itinerario, iscriversi a un viaggio (anche per più partecipanti), scegliere il tipo di assicurazione e completare il pagamento simulato. Dopo la conclusione del viaggio, il Traveler può lasciare una recensione con punteggio e commento.

- **Planner**

L'organizzatore è responsabile della creazione degli itinerari e dei viaggi. Può inserire itinerari dettagliati, associarli a categorie e Paesi, e pubblicare viaggi con date e prezzi. Ha accesso alla lista dei viaggi creati e può gestire i partecipanti, aggiornare informazioni.

- **Admin**

L'amministratore ha il ruolo di supervisione e gestione globale. Può creare utenti Planner e Admin, gestire categorie e Paesi, e monitorare l'intero sistema. Ha in aggiunta tutte le funzionalità comprese nel perimetro di un Planner. Può inoltre gestire e cancellare viaggi non ancora confermati. L'admin ha infine la responsabilità di mantenere la qualità dei contenuti e garantire il corretto funzionamento della piattaforma.

## Dettaglio funzionalità per categoria utente

### **Guest**

- Registrazione nuovo account Traveler
- Visualizzazione elenco itinerari e viaggi disponibili
  - Filtri per categoria
  - Filtri per Paese
  - Filtri per Area Geografica
- Visualizzazione dettagli di un singolo itinerario/viaggio
- Visualizzazione recensioni e punteggio medio
- Reset password

### **Traveler**

- Login
- Prenotazione di un viaggio
  - Inserimento partecipanti
  - Scelta assicurazione
  - Pagamento simulato
- Visualizzazione elenco viaggi prenotati
- Visualizzazione dettagli singola prenotazione
- Inserimento recensione su viaggio concluso
- Visualizzazione recensioni personali

### **Planner**

- Creazione di un itinerario di viaggio
- Creazione di un viaggio associato a itinerario esistente
- Visualizzazione elenco itinerari creati
- Visualizzazione elenco itinerari disponibili in piattaforma
- Visualizzazione elenco viaggi creati (con filtro data di partenza)
- Visualizzazione elenco viaggi disponibili in piattaforma (con filtro data di partenza)
- Gestione itinerari
  - Aggiornamento informazioni viaggio
- Visualizzazione elenco partecipanti per singolo viaggio

### **Admin**

(in aggiunta alle funzionalità del Planner)

- Gestione utenti
  - Creazione account Admin e Planner
  - Attivazione/disattivazione utenti
- Gestione categorie
  - Inserimento nuove categorie
- Gestione Paesi e Aree geografiche
  - Inserimento nuovi Paesi e Aree geografiche
- Gestione viaggi
  - Cancellazione viaggio non ancora confermato

# Architettura

## Architettura tecnologia applicazione

La soluzione è basata su una web application che integra in un unico componente le funzioni di Frontend e Backend, sviluppata in linguaggio **Java** e basata sul Framework **Spring Boot**, con integrazione del motore di template **Thymeleaf**.

L'ambiente di sviluppo e di esecuzione della Demo è **IntelliJ IDEA Ultimate**, che offre strumenti avanzati per il refactoring, il debugging e l'integrazione con i principali framework Java, rendendo il processo di sviluppo più fluido e produttivo. Come web server e servlet container per il codice Java è utilizzato **Apache Tomcat**.

La scelta di **Spring Boot** è motivata dalla sua capacità di fornire una gestione moderna e automatizzata delle sottostrutture del software. Grazie alle **@Annotation** e alle configurazioni predefinite, consente di concentrarsi sulla progettazione e sulla logica applicativa specifica del problema, evitando di implementare manualmente componenti di base come l'accesso al database, la gestione dei test o l'integrazione con altri moduli.

Un ulteriore vantaggio è la possibilità di aggiungere facilmente **dipendenze** tramite **Spring Initializr** o il file pom.xml. Tra queste, l'inclusione di **Lombok** rappresenta un notevole miglioramento in termini di efficienza. Questa libreria riduce la quantità di codice boilerplate grazie ad annotazioni come:

- **@Data**: genera automaticamente getter, setter, `toString()`, `equals()` e `hashCode()`, semplificando la gestione delle entità.
- **@AllArgsConstructor**: crea un costruttore con tutti i campi, utile per l'inizializzazione rapida degli oggetti.
- **@RequiredArgsConstructor**: genera un costruttore per i campi final o annotati con `@NonNull`, favorendo l'immutabilità e la sicurezza del codice.

Queste annotazioni rendono il codice più leggibile, riducono gli errori e velocizzano lo sviluppo.

**Thymeleaf**, perfettamente integrato con Spring Boot, è stato scelto per la sua versatilità e per la capacità di lavorare direttamente sul codice **HTML** delle pagine senza mescolare codice Java. Thymeleaf estende i tag HTML con attributi aggiuntivi che consentono la visualizzazione dei dati provenienti dal **Model**, incluse elaborazioni condizionali e cicli.

Per una presentazione grafica più gradevole, le pagine generate tramite Thymeleaf includono riferimenti alla libreria **CSS Bootstrap 5.0**, che fornisce stili predefiniti, palette di colori e un sistema di griglie responsive, garantendo un design moderno e conforme alle best practice di sviluppo web.

## Gestione dei dati (Database)

Per la memorizzazione e gestione dei dati, la soluzione prevede l'utilizzo di un **server MySQL**, che si interfaccia tramite l'implementazione **JPA** di **Hibernate**. Questa scelta semplifica notevolmente la scrittura del codice, consentendo al programmatore di operare a livello applicativo, dichiarando le necessità tramite **@Annotation**, mentre la libreria si occupa di tradurre tali annotazioni e le chiama in linguaggio **SQL** specifico per MySQL.

I principali vantaggi di questa scelta sono:

- **Astrazione dal linguaggio SQL**: grazie a JPA/Hibernate, non è necessario scrivere query SQL complesse; il mapping tra oggetti Java e tabelle è gestito automaticamente.
- **Portabilità**: il codice è indipendente dal database specifico; cambiando il driver, è possibile migrare facilmente verso altri DB (PostgreSQL, Oracle, ecc.).
- **Gestione automatica delle relazioni**: supporto nativo per relazioni **One-to-One**, **One-to-Many**, **Many-to-Many**, con gestione delle chiavi esterne e join.
- **Compatibilità con Spring Data JPA**: che aggiunge repository predefiniti e metodi CRUD pronti all'uso, riducendo ulteriormente il codice boilerplate.

## Componenti aggiuntivi – Stripe

Per la gestione dei pagamenti da parte del **Traveler**, è stata utilizzata la piattaforma **Stripe** ([www.stripe.com](http://www.stripe.com)), che offre API robuste e sicure, progettate per garantire transazioni affidabili e conformi agli standard di sicurezza. L'integrazione è stata realizzata tramite una dipendenza **Maven**, che consente di incorporare le librerie Stripe nel progetto in modo semplice e immediato.

Per la demo è stato creato un account personale Stripe, utilizzato in modalità test per simulare il flusso di pagamento.

La simulazione è stata gestita tramite **Stripe CLI**, che permette di emulare eventi e webhook in locale:

- Il comando stripe listen --forward-to http://localhost:XXXX/webhook/stripe intercetta gli eventi di pagamento e li inoltra all'applicativo.
- Il listener REST interno verifica il **webhook secret** e risponde con **HTTP 200 OK** quando il pagamento viene elaborato correttamente.
- Eventi di test, come stripe trigger payment\_intent.succeeded, consentono di simulare transazioni andate a buon fine.

Questo approccio consente di testare l'intero flusso di pagamento in locale, senza necessità di deploy remoto, garantendo sicurezza e affidabilità.

## Componenti aggiuntivi – Java Mail Sender

Per l'invio di messaggi e-mail agli utenti dell'applicativo – come il reset della password, le notifiche di conferma viaggio e i vari promemoria – è stata utilizzata l'interfaccia **JavaMailSender**, perfettamente integrata con **Spring Boot** e aggiunta al progetto tramite la dipendenza **Maven** spring-boot-starter-mail.

Questa soluzione consente di inviare messaggi in formato **MIME** avanzato, indispensabile per includere link cliccabili e migliorare la presentazione dei contenuti. Il servizio opera come un normale client SMTP, con supporto per connessioni sicure tramite STARTTLS, garantendo affidabilità e protezione dei dati.

Per la demo è stata creata una casella di posta **Gmail** dedicata, utilizzata come server SMTP per l'invio delle comunicazioni.

## Gestione Sicurezza

### Autenticazione degli Utenti

L'autenticazione è gestita tramite **Spring Security**, che intercetta le credenziali inviate dal form di login e, in caso di successo, crea in sessione il **SecurityContextHolder** contenente le informazioni necessarie per la gestione dell'utente autenticato. Spring Security è integrato nativamente in **Spring Boot**, richiedendo solo configurazioni minime per essere operativo.

Il login avviene tramite un form dedicato che invia le credenziali con una richiesta **POST** all'endpoint configurato (/auth/authenticate). La verifica è delegata al servizio **CustomUserDetailsService**, che implementa l'interfaccia **UserDetailsService** e recupera l'utente dal database. Il servizio controlla che l'utente esista, sia attivo e assegna il ruolo corretto, aggiungendolo alla lista delle **GrantedAuthority** per la gestione delle autorizzazioni.

Le password sono memorizzate nel database in forma cifrata tramite **BCryptPasswordEncoder**, che applica un algoritmo di hashing con **salt** dinamico. Questo approccio rende estremamente difficile la decodifica inversa e protegge da attacchi a dizionario o brute force. La verifica della password avviene applicando lo stesso algoritmo di hashing con salt, garantendo coerenza e sicurezza.

### Autorizzazione degli Utenti

Le autorizzazioni sono gestite direttamente dalla **SecurityFilterChain**, configurata nella classe **SecurityConfig**. Questa definisce le regole di accesso basate sui ruoli associati all'utente. Nell'applicazione sono previsti tre ruoli principali:

- **ADMIN**
- **PLANNER**
- **TRAVELER**

Le regole di accesso si basano sui **path delle richieste**:

- Pagine pubbliche ("/", "/home/\*\*", "/auth/\*\*", "/public", "/trips/\*\*", "/search/\*\*", "/error/\*\*", "/uploads/\*\*") e risorse statiche ("js/\*\*", "/css/\*\*", "/images/\*\*") → accessibili a tutti.
- /admin/\*\* → accessibile solo agli utenti con ruolo **ADMIN**.
- /planner/\*\* → accessibile ad **ADMIN** e **PLANNER**.
- /traveler/\*\* e /bookings/\*\* → accessibili a qualsiasi utente autenticato.
- Endpoint webhook Stripe (/stripe/webhook) → sempre consentito per garantire la ricezione degli eventi di pagamento.
- Qualsiasi altra richiesta → richiede autenticazione.

Questa configurazione centralizzata nel **filter chain** semplifica il controllo delle autorizzazioni, evitando logiche ridondanti nei controller. Per scenari complessi, è possibile utilizzare annotazioni come @PreAuthorize per controlli granulari.

#### Gestione CSRF e Remember-Me

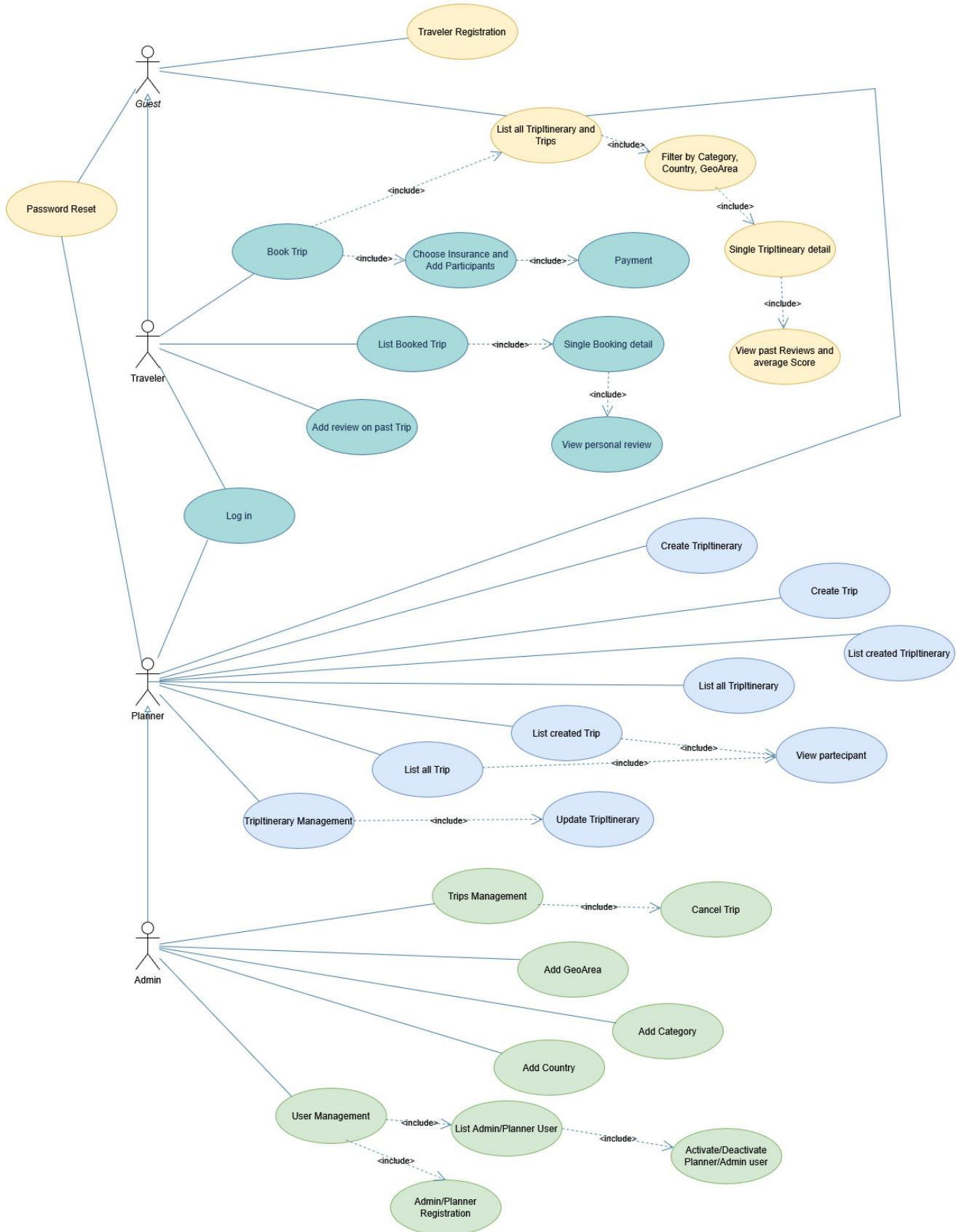
- **CSRF:**  
Attivo per le pagine web, ma disabilitato per gli endpoint REST e webhook (/api/\*\*, /stripe/webhook) per evitare conflitti con chiamate esterne.
- **Remember-Me:**  
Abilitato per mantenere la sessione attiva fino a 7 giorni, migliorando l'esperienza utente senza richiedere login frequenti.

#### Motivazioni delle Scelte:

- **Spring** **Security:**  
Offre un framework completo e integrato per la gestione di autenticazione e autorizzazione, riducendo il rischio di vulnerabilità e semplificando lo sviluppo.
- **BCrypt:**  
Garantisce un hashing sicuro con salt dinamico, proteggendo le password da attacchi comuni e rendendo più difficile la compromissione dei dati in caso di violazione del database.

# Progettazione del sistema

## Use Case Diagram

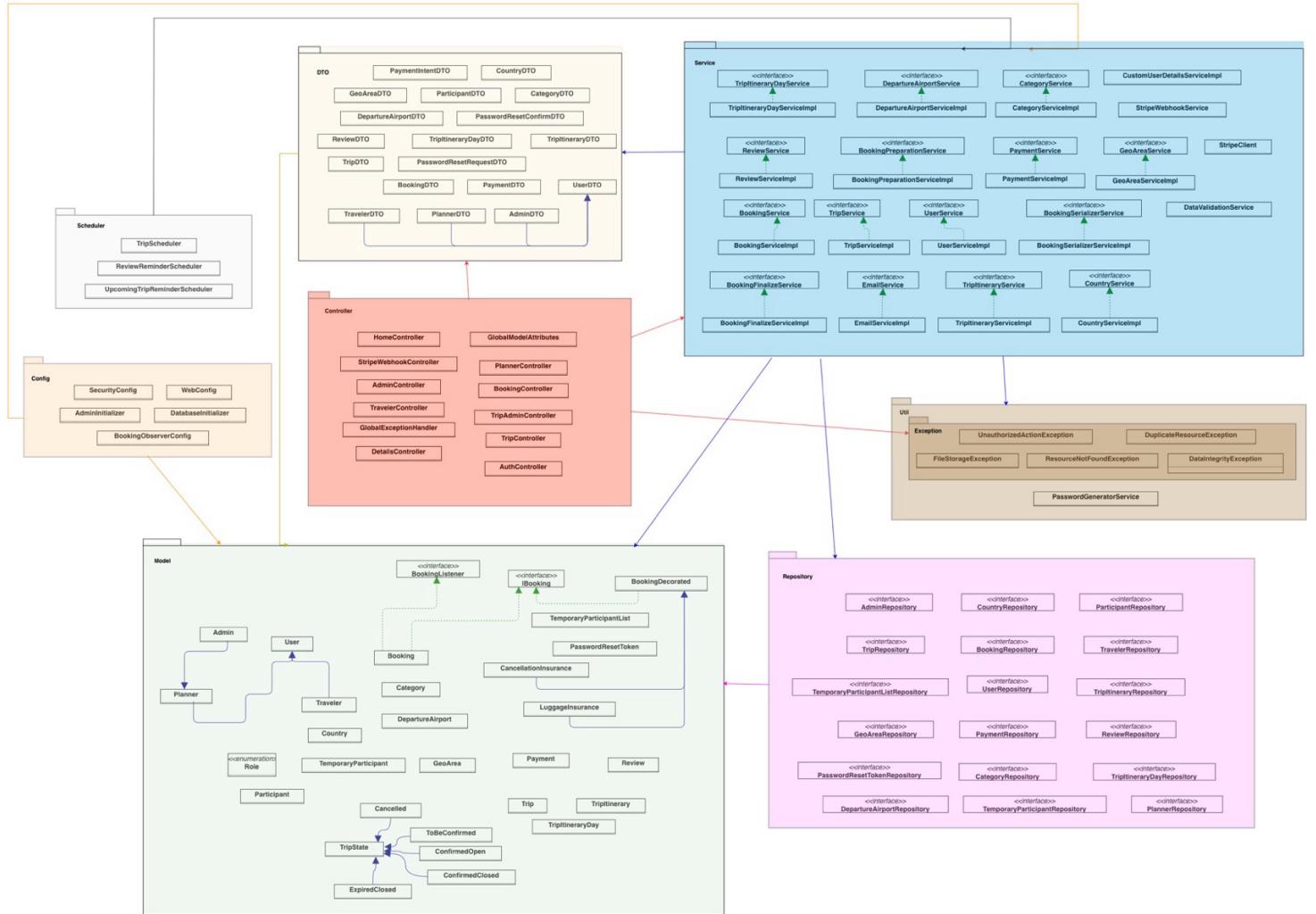


Nello Use Case sono raggruppate tutti i requisiti funzionali dell'applicazione e le loro relazioni. Vediamo appunto i 4 attori, Guest, Traveler, Planner e Admin, collegati a tutte le azioni che possono svolgere.

Tutti gli utenti registrati hanno le stesse funzionalità previste per il Guest ad eccezione della registrazione. Infatti, mentre la registrazione Traveler è gestibile in autonomia da ogni utente Guest, la registrazione Planner e la registrazione Admin è invece gestibile da un Admin già registrato in piattaforma. Inoltre se un utente è già registrato come Planner o Admin non può registrarsi anche come Traveler.

L'Admin generalizza Planner, in quanto ha accesso a entrambe le funzionalità previste sia per Admin che per Planner.

## Package Diagram



Il progetto è strutturato seguendo una chiara architettura stratificata, organizzata secondo i principi del pattern **MVC** (Model–View–Controller). Tale approccio separa nettamente la logica di dominio, la gestione dei dati e la presentazione, migliorando la leggibilità del codice e facilitando l’evoluzione del sistema.

Alla base dell'applicazione si trova il package **Model**, che contiene le entità del dominio e rappresenta lo stato persistente del sistema. Sopra questo livello opera il package **Repository**, responsabile dell'accesso ai dati tramite **Spring Data JPA**. Il package **Service** implementa la logica applicativa vera e propria, orchestrando le operazioni sulle entità e applicando le regole di business. Il package **Controller** costituisce il punto di contatto con l'utente: riceve le richieste HTTP, le valida, invoca i servizi e restituisce le viste appropriate. La comunicazione tra Controller e Service è mediata dai **DTO** (Data Transfer Object), contenuti in un package dedicato. L'utilizzo dei DTO consente di evitare l'esposizione diretta delle entità di dominio, riduce l'accoppiamento tra livelli e permette un maggiore controllo sui dati inviati e ricevuti dal client, contribuendo a una migliore sicurezza e manutenibilità. Completano la struttura i package **Config**, **Scheduler** ed **Util**, che forniscono funzionalità trasversali di configurazione, gestione dei processi pianificati e centralizzazione degli errori applicativi.

## **Model**

Contiene tutte le entità del dominio mappate con JPA (Trip, Booking, Traveler, Payment, Review, ecc.) e gli oggetti del modello comportamentale, inclusi gli stati del Trip (State pattern), i decorator dell'assicurazione e le relazioni tra i partecipanti. È il cuore dell'applicazione: rappresenta i dati persistiti e la logica minima legata al dominio.

Dipendenze in ingresso:

- Service (usa e modifica le entity)
- Repository (persiste le entity)
- DTO (effettua conversioni da/verso Model)
- Config (per la configurazione di sicurezza in base ai ruoli)

## **Repository**

Contiene tutte le interfacce che estendono Spring Data JPA (es. TripRepository, BookingRepository, UserRepository). Si occupa dell'accesso ai dati, isolando l'applicazione dai dettagli della persistenza.

Dipendenze:

- Dipende dal package Model (opera sulle entity).
- È usato unicamente dal package Service.

## **Service**

Racchiude l'intera business logic dell'applicazione. Comprende sia le interfacce dei servizi sia le implementazioni concrete (TripServiceImpl, BookingServiceImpl, PaymentServiceImpl, ecc.).

Qui risiede la logica applicativa complessa: validazioni, gestione stati, processi di booking, pagamenti, stripe client, scheduler, invio email.

Dipendenze:

- Usa Repository per la persistenza.
- Usa Model per manipolare le entity.
- Usa DTO per gli scambi di dati verso Controller.
- Usa Exception per gestire errori applicativi.
- Richiamato da Controller e Scheduler.

## **DTO**

Contiene gli oggetti di trasferimento dati utilizzati per comunicare tra Controller e Service.

Mappa in modo controllato le informazioni provenienti dalle entità, evitando di esporre direttamente il Model.

Dipendenze:

- Effettua conversioni da/verso Model.
- È usato sia da Controller che da Service.

## **Controller**

Gestisce i flussi web: riceve richieste HTTP, valida input, invoca i service e restituisce viste o dati.

Include controller dedicati ai vari ruoli (Admin, Planner, Traveler), al booking, ai Trip, ai dettagli e anche l'handler globale delle eccezioni.

Dipendenze:

- Chiama il package Service.
- Usa DTO per input/output.
- Usa le Exception per gestire situazioni di errore.
- Collega il Model unicamente tramite DTO o tramite l'attributo globale "itineraries".

## **Util**

Raccoglie tutte le eccezioni personalizzate dell'applicazione (ResourceNotFoundException, DuplicateResourceException, UnauthorizedActionException, ecc.) e utility correlate come il PasswordGeneratorService.

Serve a centralizzare la gestione degli errori e produrre messaggi significativi.

Dipendenze:

- Usato da Service e Controller.

## Scheduler

Contiene gli scheduler che eseguono operazioni pianificate, come reminder per le recensioni o controlli sullo stato dei Trip.

Dipendenze:

- Chiama esplicitamente i Service per eseguire la logica programmata.

## Config

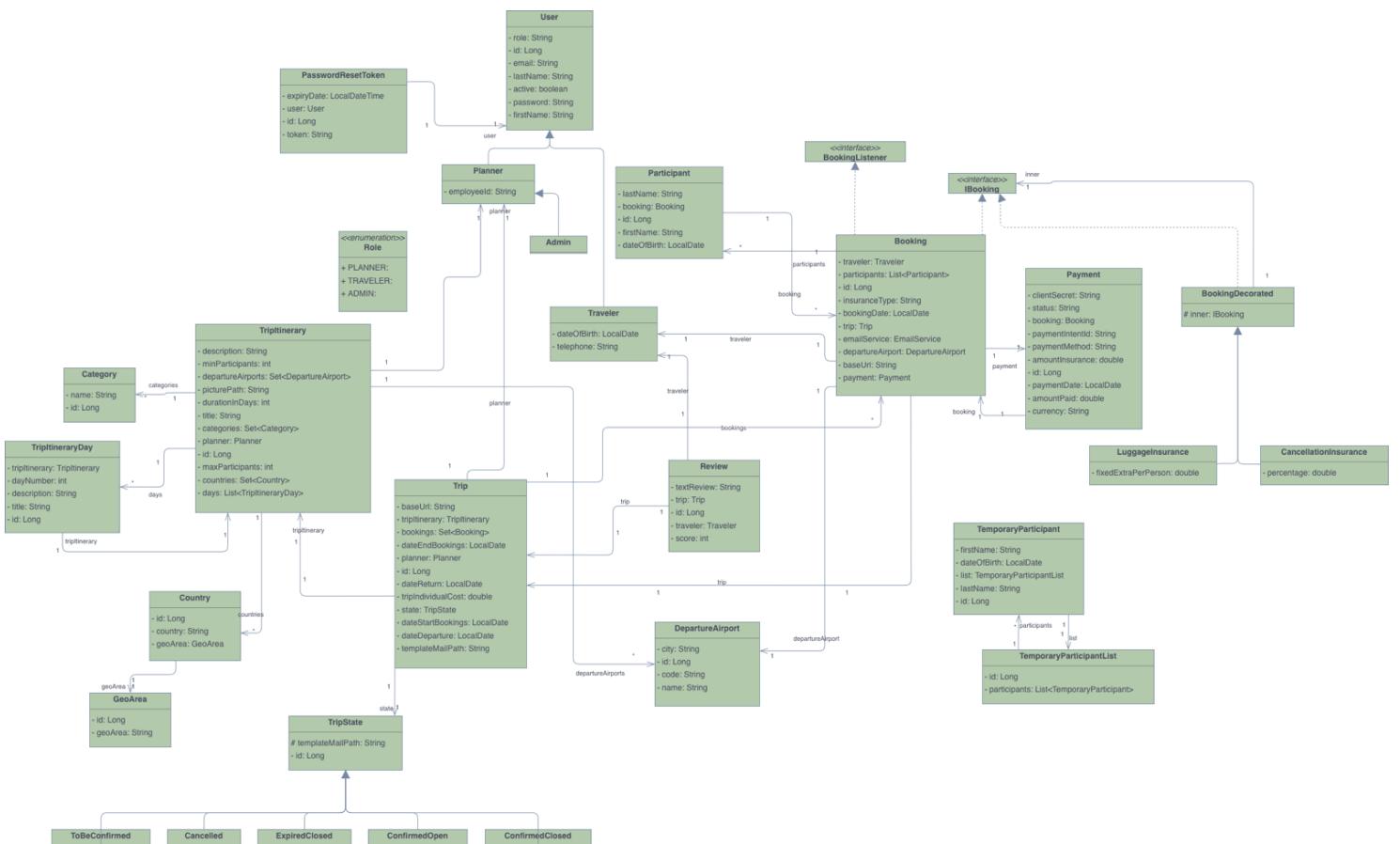
Contiene le configurazioni di Spring Boot, tra cui sicurezza (SecurityConfig), inizializzazione database, web config, osservatori e initializer.

Definisce come il framework deve avviare e strutturare l'applicazione.

Dipendenze:

- Usa i Service (es. CustomUserDetailsService per Security).
- In alcuni casi usa Model (ruoli utente).

## Class Diagram – package Model



Il package model contiene tutte le entità del dominio dell'applicazione *Adventure Together*. Queste classi rappresentano i concetti principali della piattaforma — viaggi, itinerari, utenti, prenotazioni, pagamenti e categorie — e costituiscono il livello più basso e fondamentale dell'intera architettura. Le entità sono mappate tramite JPA e riflettono in modo fedele le regole e i processi del sistema.

## Trip, Triplinerary e TriplineraryDay

Il cuore del modello è rappresentato dal concetto di viaggio organizzato.

## Trip

La classe *Trip* rappresenta una singola partenza di uno specifico viaggio. Contiene attributi fondamentali come:

- date di apertura e chiusura delle prenotazioni
- data di partenza e di ritorno
- costo individuale
- stato attuale del viaggio
- associazioni con Planner, Triptinerary, prenotazioni, categorie e aeroporti

Il Trip utilizza il *pattern State* per gestire il proprio ciclo di vita (da “ToBeConfirmed” fino agli stati finali). Ogni Trip è sempre associato a:

- un Triptinerary, che definisce la struttura del viaggio
- un Planner, responsabile dell’organizzazione
- più Booking, che rappresentano le prenotazioni dei viaggiatori

Il Trip è inoltre una delle entità esposte agli utenti finali, poiché per ogni Itinerary vengono mostrati i Trip disponibili (con date e prezzi specifici).

### TripState e sottoclassi

Le classi che modellano gli stati del Trip (ToBeConfirmed, ConfirmedOpen, ConfirmedClosed, ExpiredClosed, Cancelled) rappresentano un uso del *pattern State* (approfondimento nella sezione dedicata). Ogni stato definisce i comportamenti per:

- handle() → transizioni automatiche
- cancel() → cancellazione del viaggio (solo admin)

Sono mappate in una *Single Table* con discriminatore. I dettagli completi sono trattati nella sezione dedicata ai Design Pattern.

## Triptinerary

Rappresenta il modello di un viaggio, ovvero la parte descrittiva e strutturale. Contiene:

- titolo, descrizione ed immagine principale
- durata totale
- numero minimo e massimo di partecipanti
- associazioni con Country, Category e DepartureAirport
- la lista dei TriptineraryDay che descrivono giorno per giorno le attività previste

Un itinerario può essere associato a molteplici Trip, ciascuno con date e costi differenti, e viene creato soltanto dai Planner o dagli Admin.

## TriptineraryDay

Modella il contenuto dettagliato di una singola giornata del viaggio. Ogni giorno include:

- numero progressivo
- descrizione estesa delle attività
- relazione con il Triptinerary al quale appartiene

La coppia (*itineraryId*, *dayNumber*) è unica per prevenire duplicazioni.

## User, Traveler, Planner, Admin, Role

Il sistema di utenti è implementato tramite ereditarietà.

## User (classe astratta)

È la superclasse di tutti gli utenti della piattaforma e implementa *UserDetails* per l'integrazione con Spring Security. Contiene:

- email (usata come username)
- password
- nome, cognome
- flag "active"
- ruolo dell'utente (enum Role)
- metodi richiesti da Spring Security

L'ereditarietà utilizza la strategia JOINED per mantenere le sottoclassi in tabelle separate.

## Traveler

È l'utente finale che prenota i viaggi. Estende User, definisce attributi specifici per i viaggiatori e partecipa alle relazioni con Booking, Review e Participant.

## Planner

L'utente che organizza i viaggi. Creato dagli Admin, al primo accesso è obbligato a impostare una nuova password. Gestisce Trip e Triplinerary.

## Admin

Estende Planner e non introduce nuovi attributi, ma ha privilegi amministrativi completi. Può creare Planner e Admin, modificare categorie, aeroporti, geoArea e cancellare Trip.

## Role

Enum dei ruoli di sistema (TRAVELER, PLANNER, ADMIN); utilizzato per autorizzare funzioni e definire i permessi.

## Review

Rappresenta una recensione lasciata da un Traveler per un Trip a cui ha partecipato. Contiene:

- punteggio (1–5)
- testo della recensione
- associazioni ManyToOne verso Trip e Traveler

Le recensioni sono disponibili solo dopo il completamento del viaggio.

## Payment

Modella il pagamento di una singola prenotazione. Include:

- data del pagamento (impostata automaticamente)
- importo base
- costo dell'assicurazione
- relazione OneToOne verso Booking

Viene generato automaticamente al momento della conferma della prenotazione.

## Participant e TemporaryParticipant

### Participant

Rappresenta una persona inclusa in una prenotazione. Un Booking può avere più participant, anche con nomi duplicati, poiché ogni prenotazione ha il proprio set di partecipanti (anche ripetuti tra prenotazioni diverse).

## **TemporaryParticipant e TemporaryParticipantList**

Utilizzati durante il processo di prenotazione, prima che i dati vengano definitivi. Consentono di gestire dinamicamente i partecipanti inseriti dall'utente step-by-step.

## **GeoArea, Country, DepartureAirport, Category**

Classi di supporto che consentono la categorizzazione e filtrazione dei viaggi.

- **GeoArea:** macro area geografica (es. Europa, Asia).
- **Country:** nazione appartenente a una GeoArea.
- **DepartureAirport:** aeroporto di partenza (es. MXP, FCO).
- **Category:** tipo di viaggio (Avventura, Relax, Cultura).

Queste entità sono gestibili dagli Admin.

## **Prenotazioni: Booking, IBooking, BookingDecorator**

La prenotazione è uno degli elementi più ricchi del modello.

### **Booking**

La classe **Booking** rappresenta il fulcro del processo di prenotazione all'interno della piattaforma. Ogni prenotazione collega un *Traveler* a uno specifico *Trip* e contiene tutte le informazioni necessarie per finalizzare l'acquisto del viaggio.

In particolare, una Booking include:

- il costo base del viaggio;
- il costo dell'assicurazione, calcolato inizialmente come il 10% del costo base;
- il riferimento al *Traveler* che effettua la prenotazione;
- il *Trip* selezionato;
- l'aeroporto di partenza scelto;
- la lista dei *Participant*, che rappresentano tutte le persone incluse nella prenotazione;
- un'istanza di *Payment*, generata automaticamente al momento della conferma.

La classe **Booking** rappresenta il fulcro del processo di prenotazione all'interno della piattaforma. Ogni prenotazione collega un *Traveler* a uno specifico *Trip* e contiene tutte le informazioni necessarie per finalizzare l'acquisto del viaggio.

In particolare, una Booking include:

- il costo base del viaggio;
- il costo dell'assicurazione, calcolato inizialmente come il 10% del costo base;
- il riferimento al *Traveler* che effettua la prenotazione;
- il *Trip* selezionato;
- l'aeroporto di partenza scelto;
- la lista dei *Participant*, che rappresentano tutte le persone incluse nella prenotazione;
- un'istanza di *Payment*, generata automaticamente al momento della conferma.

Per modellare in modo estensibile il calcolo del costo totale della prenotazione, **Booking** implementa l'interfaccia **IBooking**, che espone i metodi per ottenere i costi del viaggio e dell'assicurazione. Su questa interfaccia si innesta un sistema flessibile basato sul *pattern Decorator* (approfondito in una sezione dedicata), che permette di applicare dinamicamente coperture assicurative aggiuntive.

Oltre alla gestione dei costi, la classe **Booking** partecipa a un meccanismo di notifica basato su un accenno al *pattern Observer* (anche qui approfondito nella sezione dedicata).

## PasswordResetToken

Rappresenta un token per il reset della password, collegato a uno User e dotato di data di scadenza. È utilizzato sia per i reset volontari sia per i reset obbligatori dei nuovi Planner/Admin.

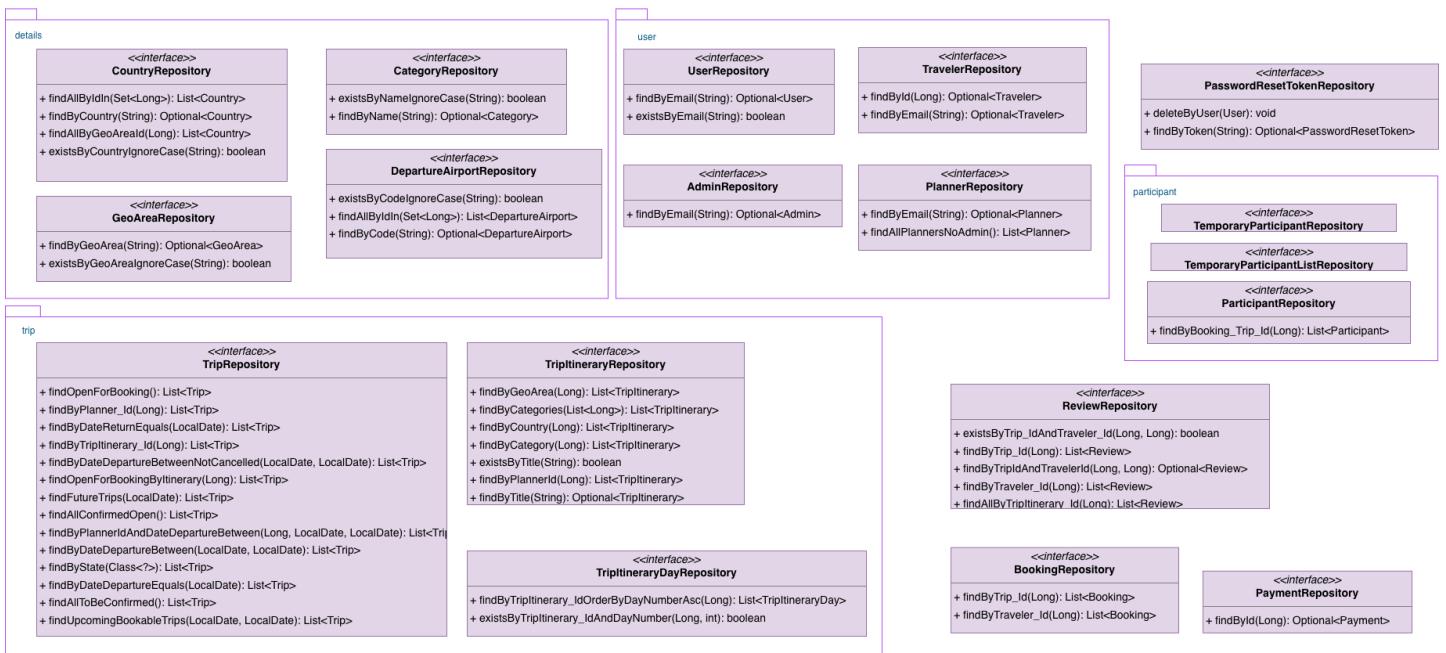
## Organizzazione interna

Per rendere più leggibile e manutenibile la struttura, alcune parti del package sono state suddivise in sotto-package\* tematici:

- **model.trip**  
Contiene le classi relative a Trip, Triplinerary, TriplineraryDay, TripState (e relative classi concrete)
- **model.user**  
Raccoglie le classi legate agli utenti (User, Traveler, Planner, Admin e Role)
- **model.participant**  
Contiene le classi dedicate ai partecipanti effettivi e alle strutture temporanee utilizzate durante la procedura di prenotazione (Participant, TemporaryParticipant, TemporaryParticipantList)
- **model.details**  
Include le classi di supporto e di dettaglio come Country, Category, GeoArea e DepartureAirport
- **model.booking**  
Include le classi dedicate alla prenotazione (IBooking, BookingListener, Booking, BookingDecorator e relative classi concrete)

\*I sotto-package non sono stati riportati nel class diagram per non sovraccaricare il grafico.

## Class Diagram – package Repository



Il package **repository** contiene tutte le interfacce responsabili dell'accesso e della manipolazione dei dati persistiti nel database tramite Spring Data JPA. Queste interfacce fungono da strato di comunicazione tra il dominio applicativo e il livello di persistenza, mappando in modo diretto la struttura del package model: per ogni entità principale esiste infatti un repository dedicato.

## Ruolo del package

- Gestisce le operazioni CRUD standard ereditate da **JpaRepository**.
- Definisce metodi di query aggiuntivi attraverso la *query creation by method name* di Spring Data JPA.
- Offre metodi personalizzati mediante annotazioni **@Query**, utili per interrogazioni più complesse, join multipli o filtri avanzati.

- Fornisce un accesso centralizzato e tipizzato ai dati, mantenendo coerente la separazione tra business logic (service) e persistenza (repository).

## Organizzazione interna

Anche qui, per rendere più leggibile e manutenibile la struttura, alcune parti del package sono state suddivise in sotto-package tematici:

- **repository.trip**

Contiene i repository relativi a Trip, Triptinerary, TriptineraryDay

- **repository.user**

Raccoglie i repository delle entità legate agli utenti (User, Traveler, Planner, Admin)

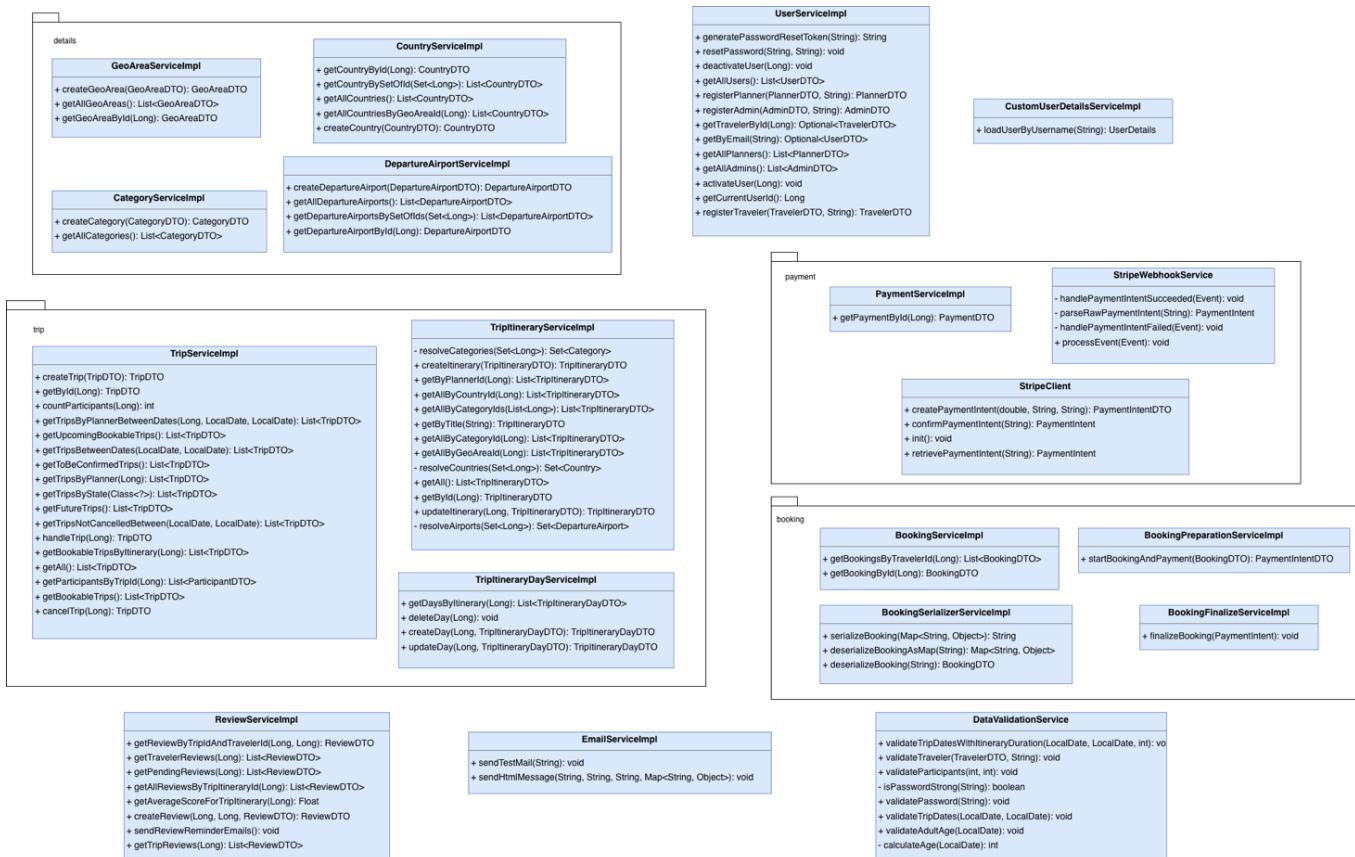
- **repository.participant**

Contiene i repository dedicati ai partecipanti dei booking e alle strutture temporanee utilizzate durante la procedura di prenotazione.

- **repository.details**

Include i repository relativi alle entità di supporto e di dettaglio come Country, Category, GeoArea e DepartureAirport.

## Class Diagram – package Service



Il package **service** costituisce il livello centrale della logica applicativa della piattaforma *Adventure Together*. Esso incapsula tutte le regole di business, le validazioni e i flussi operativi che governano viaggi, itinerari, utenti, prenotazioni e pagamenti. I servizi fungono da ponte tra i controller (che gestiscono le richieste utente) e il livello di persistenza (repository), interpretando la logica del dominio e coordinando le interazioni tra le diverse entità.

Ogni componente del package è organizzato seguendo un modello *interfaccia + implementazione*, con tutte le classi concrete raccolte nel sotto-package **impl**. I servizi lavorano principalmente con **DTO**, così da mantenere separati il dominio persistito e i dati scambiati verso l'esterno, favorendo sicurezza, chiarezza e testabilità.

Data la dimensione funzionale della piattaforma, il package è suddiviso in quattro sotto-package tematici, ognuno dedicato a un ambito applicativo specifico:

## **service.trip — Gestione viaggi e itinerari**

Raccoglie i servizi dedicati alla creazione, aggiornamento e consultazione di Trip, Triptinerary e TriptineraryDay. Qui risiede la logica fondamentale del dominio travel, tra cui:

- costruzione e modifica degli itinerari dettagliati,
- gestione delle giornate dell'itinerario,
- applicazione delle regole di prenotabilità e dei vincoli di partecipanti,
- controllo delle date di apertura e chiusura delle prenotazioni,
- gestione dell'intero ciclo di vita dei viaggi tramite pattern State (creazione, conferma, scadenza, cancellazione),
- visualizzazione filtrata dei viaggi per planner, area geografica, paese o categoria.

Questi servizi orchestранo le operazioni tra più repository e garantiscono la coerenza tra itinerari, viaggi e partecipanti.

## **service.booking — Gestione completa delle prenotazioni**

Contiene i servizi dedicati a tutti i passaggi del processo di prenotazione: dalla preparazione iniziale fino alla finalizzazione dopo il pagamento.

Il sotto-package gestisce:

- preparazione del booking (con calcolo costi, assicurazioni, verifica disponibilità),
- serializzazione e deserializzazione del booking in vista dell'integrazione con Stripe,
- ricostruzione del booking dai metadata del PaymentIntent,
- salvataggio effettivo della prenotazione nel database,
- aggiornamento dello stato del viaggio in base ai partecipanti,
- invio delle email di conferma,
- visualizzazione delle prenotazioni per traveler e planner.

È una delle parti più articolate dell'applicazione, dato che coordina utenti, viaggi, pagamenti, DTO complessi e notifiche.

## **service.payment — Integrazione con Stripe e gestione pagamenti**

Questo sotto-package gestisce tutto ciò che riguarda il pagamento tramite Stripe, sia lato API che lato webhook. Qui si trovano:

- il client responsabile della creazione e gestione dei *PaymentIntent*,
- la logica per confermare e recuperare pagamenti,
- la gestione degli eventi webhook
- il collegamento tra l'esito del pagamento e lo stato del booking,
- la visualizzazione dei pagamenti lato utente/amministrazione.

Il sistema assicura che i pagamenti siano sincronizzati con il flusso di prenotazione e che eventuali errori o stati intermedi vengano gestiti correttamente.

## **service.details — Gestione delle entità di supporto**

Comprende i servizi relativi alle entità "dettaglio" del dominio turistico (Country, Category, GeoArea e DepartureAirport).

Questi servizi forniscono operazioni di base (creazione, aggiornamento, eliminazione, elenchi) e controlli di coerenza come l'unicità dei nomi e la corretta associazione tra entità. Sono utilizzati in modo trasversale per compilare filtri, form, itinerari e viste riassuntive.

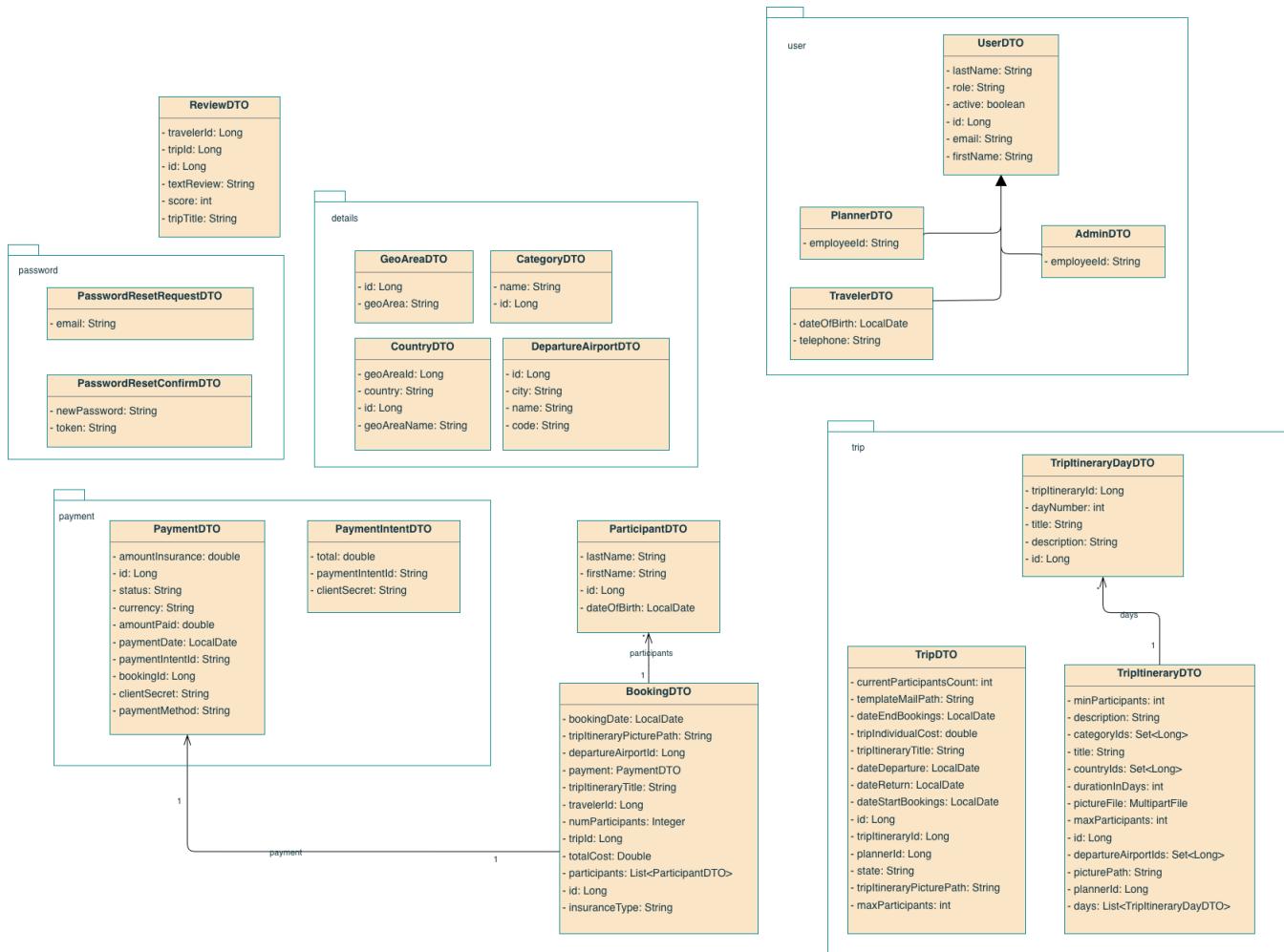
## **Servizi trasversali (parte del package principale)**

Oltre ai sotto-package principali, il package include alcuni servizi fondamentali per il funzionamento complessivo della piattaforma:

- UserService — gestione completa di utenti, ruoli, registrazione e reset password;
- CustomerUserDetailsService — caricamento utenti per Spring Security;
- DataValidationService — validazione centralizzata di età, password, date, partecipanti;
- EmailService — invio di email HTML basate su template Thymeleaf;
- ReviewService — creazione e gestione delle recensioni con calcoli e promemoria.

Questi servizi fungono da blocchi trasversali a più processi applicativi, fornendo componenti riutilizzabili e indispensabili.

## Class Diagram – package DTO



Il package **dto** contiene tutti gli oggetti di trasferimento dati utilizzati per comunicare in modo sicuro ed efficiente tra il livello di servizio e il front-end della piattaforma *Adventure Together*.

A differenza delle entità JPA presenti nel package model, i DTO non rappresentano direttamente la struttura del database, ma una versione semplificata, sicura e orientata alla presentazione dei dati.

La struttura del package è suddivisa in sotto-package tematici — **user**, **trip**, **details**, **payment**, **password** — che riprendono l’organizzazione logica del dominio.

## Allineamento alle entità, ma con struttura semplificata

La maggior parte dei DTO ricalca la corrispondente entità del package model, mantenendo solo i campi significativi per la logica applicativa e la presentazione dei dati. Sono volutamente esclusi:

- campi sensibili (es. password),
- relazioni complesse o cicliche,
- metadati tecnici di JPA.

## Presenza di campi aggiuntivi per il front-end

Alcuni DTO includono campi non presenti nell'entità corrispondente, aggiunti per agevolare la visualizzazione nelle pagine o per ridurre il numero di chiamate necessarie.

Esempio: TripDTO contiene oltre all'id del Triptinerary anche:

- triptineraryTitle
- triptineraryPicturePath
- currentParticipantsCount
- maxParticipants

o, nel caso di ReviewDTO:

- tripTitle (utile per mostrare direttamente il nome del viaggio recensito).

Questi campi derivati evitano logica aggiuntiva nel front-end e ottimizzano il flusso di dati.

## Validazioni integrate tramite annotazioni

Molti DTO includono annotazioni come: @NotBlank, @NotNull, @Positive, @Email, @Past, @Min / @Max. etc.

Queste garantiscono l'integrità dei dati in ingresso già a livello di controller.

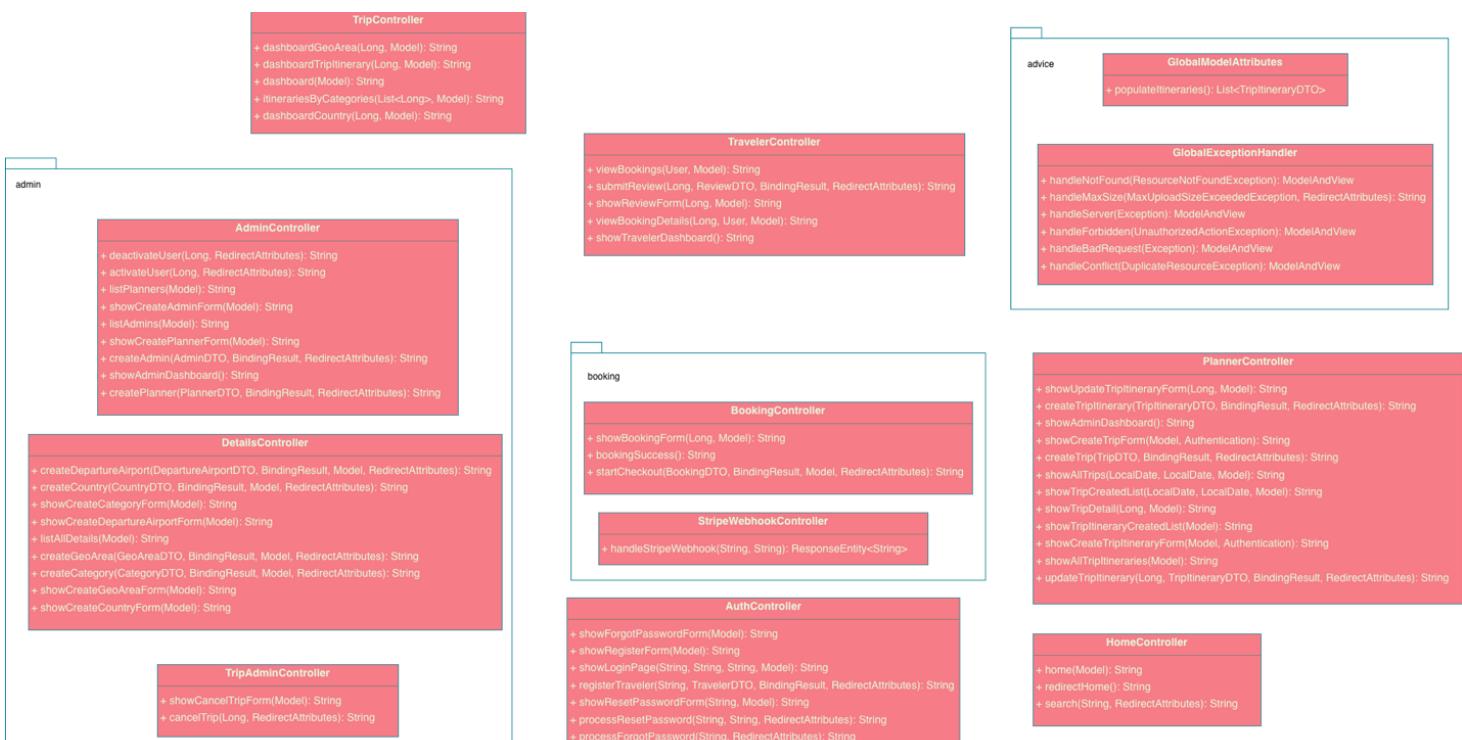
## Conversione entità ↔ DTO

Dove necessario, i DTO espongono metodi di conversione chiari e statici:

- fromEntity(Entity e)
- toEntity() (solo per i DTO che devono ricostruire un'entità)

La creazione degli oggetti avviene prevalentemente tramite via annotazione **@Builder**, così da assicurare immutabilità e leggibilità nella costruzione.

## Class Diagram – package Controller



Il package **controller** contiene tutti i controller responsabili della gestione delle richieste HTTP e dell'interazione tra il front-end e il livello di servizio dell'applicazione *Adventure Together*. A differenza del package model, la loro struttura non riflette direttamente il dominio persistito, ma è modellata sulle esigenze dell'interfaccia utente e dei flussi operativi. Ogni controller espone endpoint specifici, validati e protetti tramite la **security filter chain** definita nella configurazione di sicurezza, che regola l'accesso ai percorsi in base al ruolo dell'utente.

La struttura del package è suddivisa in alcuni sotto-package tematici:

- **controller.admin** — funzioni amministrative (AdminController, DetailsController, TripAdminController)
- **controller.advice** — componenti trasversali, come il gestore globale delle eccezioni e gli attributi condivisi (GlobalExceptionHandler, GlobalModelAttributes)
- **controller.booking** — gestione del flusso di prenotazione e dei webhook Stripe (BookingController, StripeWebhookController)

Gli altri controller principali si trovano alla radice del package (auth, planner, traveler, trip, home), ognuno dedicato a un contesto applicativo specifico.

## Descrizione dei controller

Di seguito una descrizione sintetica di ogni controller presente nel diagramma.

### Sotto-package admin

#### AdminController

Gestisce tutte le operazioni amministrative relative agli utenti di tipo Planner e Admin: creazione, attivazione, disattivazione e visualizzazione. Fornisce le schermate dedicate al pannello di amministrazione.

#### DetailsController

Gestisce la creazione, modifica e visualizzazione delle entità di dettaglio necessarie alla configurazione dei viaggi: GeoArea, Country, Category e DepartureAirport. Espone form di gestione utilizzati dai planner e dagli admin.

#### TripAdminController

Controller dedicato alle operazioni amministrative sui Trip. Permette agli Admin di visualizzare il form di cancellazione e confermare l'eliminazione di un viaggio tramite l'interfaccia web.

### Sotto-package advice

#### GlobalExceptionHandler

Gestore globale delle eccezioni dell'applicazione. Cattura errori noti (risorse non trovate, errori di validazione, eccezioni applicative) e fornisce risposte uniformi alle viste, prevenendo crash non gestiti.

#### GlobalModelAttributes

Controller advice utilizzato per aggiungere attributi condivisi a tutti i modelli delle viste. In particolare, popola la lista degli itinerari per la barra di ricerca globale.

### Sotto-package booking

#### BookingController

Gestisce il flusso di prenotazione lato traveler: mostra il form di booking, avvia il processo di checkout con Stripe (creazione del PaymentIntent) e visualizza la pagina di successo. Le fasi successive del processo (post-pagamento) sono gestite nei servizi e nei webhook.

#### StripeWebhookController

Riceve gli eventi webhook da Stripe, verifica la firma e delega la gestione al servizio di orchestrazione degli eventi. Garantisce che i pagamenti vengano correttamente sincronizzati con lo stato dei booking.

## Altri controller principali

### AuthController

Gestisce l'autenticazione e la registrazione degli utenti Traveler. Include le pagine di login, registrazione, recupero password e reset password.

### HomeController

Gestisce la visualizzazione della home page, il redirect iniziale e la logica di ricerca dei viaggi tramite parola chiave.

### TripController

Offre una dashboard pubblica e completa per la consultazione dei viaggi disponibili, filtrabili per categorie, aree geografiche, itinerari o paesi. Ogni elemento porta alla pagina di dettaglio dell'itinerario.

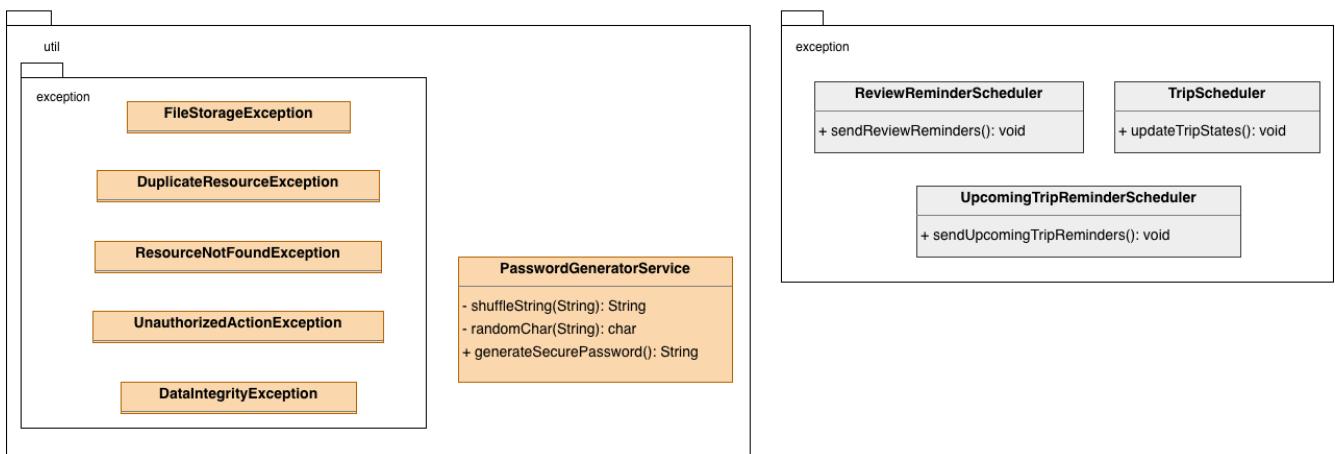
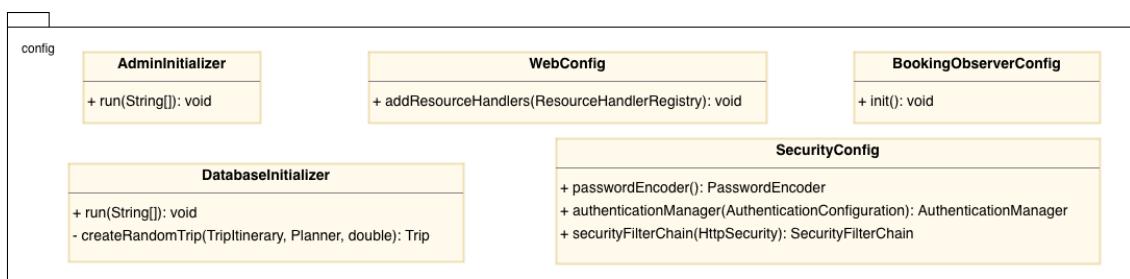
### PlannerController

Dedicato alle funzionalità degli utenti Planner (e Admin). Permette di creare e aggiornare Triptinerary, creare Trip, visualizzare le loro liste e accedere alla dashboard del planner.

### TravelerController

Gestisce le funzionalità destinate ai Traveler: dashboard personale, visualizzazione delle prenotazioni, dettagli dei viaggi prenotati e creazione delle recensioni.

## Class Diagram – packages Util, Scheduler, Config



Questa sezione raccoglie alcuni componenti trasversali dell'applicazione *Adventure Together*: configurazioni, scheduler e utility. Pur non appartenendo direttamente al dominio principale, essi svolgono funzioni fondamentali per l'inizializzazione del sistema, la sicurezza, la generazione di dati di base, la gestione di risorse esterne (come file e password), e l'automazione di processi periodici.

L'insieme è suddiviso logicamente in tre aree:

- **config** — configurazioni di avvio, sicurezza, mapping risorse e inizializzazioni;
- **scheduler** — processi pianificati per aggiornare lo stato dei viaggi e inviare email automatiche;
- **util** — eccezioni applicative e servizi di utilità.

## **Package config**

### **AdminInitializer**

Inizializza un *Admin* di default all'avvio dell'applicazione. Se il database non contiene utenti, crea automaticamente un account amministratore con credenziali predefinite. Viene eseguito tramite CommandLineRunner.

### **DatabaseInitializer**

Inizializzatore che popola il database con dati di base (GeoArea, Country, Category, DepartureAirport) quando le tabelle corrispondenti sono vuote. Include un metodo di supporto per creare Trip casuali quando necessario. Anch'esso eseguito tramite CommandLineRunner.

### **BookingObserverConfig**

Configura il sistema di **Observer** per le prenotazioni, inizializzando i componenti responsabili dell'invio di notifiche email quando cambia lo stato di un Trip associato a un booking.

### **SecurityConfig**

Contiene la configurazione generale di sicurezza dell'applicazione. Definisce il SecurityFilterChain, l'AuthenticationManager, il password encoder e le strategie di autenticazione (*per approfondimenti si rimanda alla sezione "Gestione Sicurezza" del documento*).

### **WebConfig**

Configura la gestione delle risorse statiche e dei file caricati dagli utenti. Mappa il path fisico di upload affinché le immagini degli itinerari e altri asset caricati siano serviti correttamente tramite URL.

### **Package scheduler**

Il package contiene tre scheduler basati su pianificazioni cron, che automatizzano processi ripetitivi di gestione viaggi e comunicazioni ai traveler.

#### **ReviewReminderScheduler**

Invia ogni giorno alle **22:00** email ai viaggiatori che devono ancora lasciare una recensione. Raccoglie i Trip conclusi (circa 3 giorni prima) e notifica i partecipanti tramite il ReviewService.

#### **TripScheduler**

Aggiorna automaticamente lo stato dei Trip ogni giorno alle **02:00**. Controlla le date di apertura/chiusura e passa in stato *Expired* o *Confirmed* i viaggi che soddisfano le condizioni, sincronizzando la logica del pattern State.

#### **UpcomingTripReminderScheduler**

Invia email di promemoria ai viaggiatori con un viaggio in partenza entro **7 giorni**. Lo scheduler gira quotidianamente alle **09:00**.

### **Package util**

Questo package include sia classi di utilità generale sia un insieme di eccezioni applicative personalizzate.

### **Eccezioni personalizzate**

#### **DataIntegrityException**

Segnala incoerenze nei dati forniti (es. età non valida, password troppo debole, date incoerenti). Utilizzata principalmente dai servizi di validazione.

#### **DuplicateResourceException**

Lanciata quando si tenta di creare una risorsa già esistente, come un'email duplicata nella registrazione.

#### **FileStorageException**

Indica errori durante l'upload/archiviazione dei file. Usata dal servizio di gestione file e dalla logica degli itinerari.

## ResourceNotFoundException

Restituisce un errore 404 quando una risorsa richiesta non esiste. Include informazioni dettagliate sul campo e valore della ricerca.

## UnauthorizedActionException

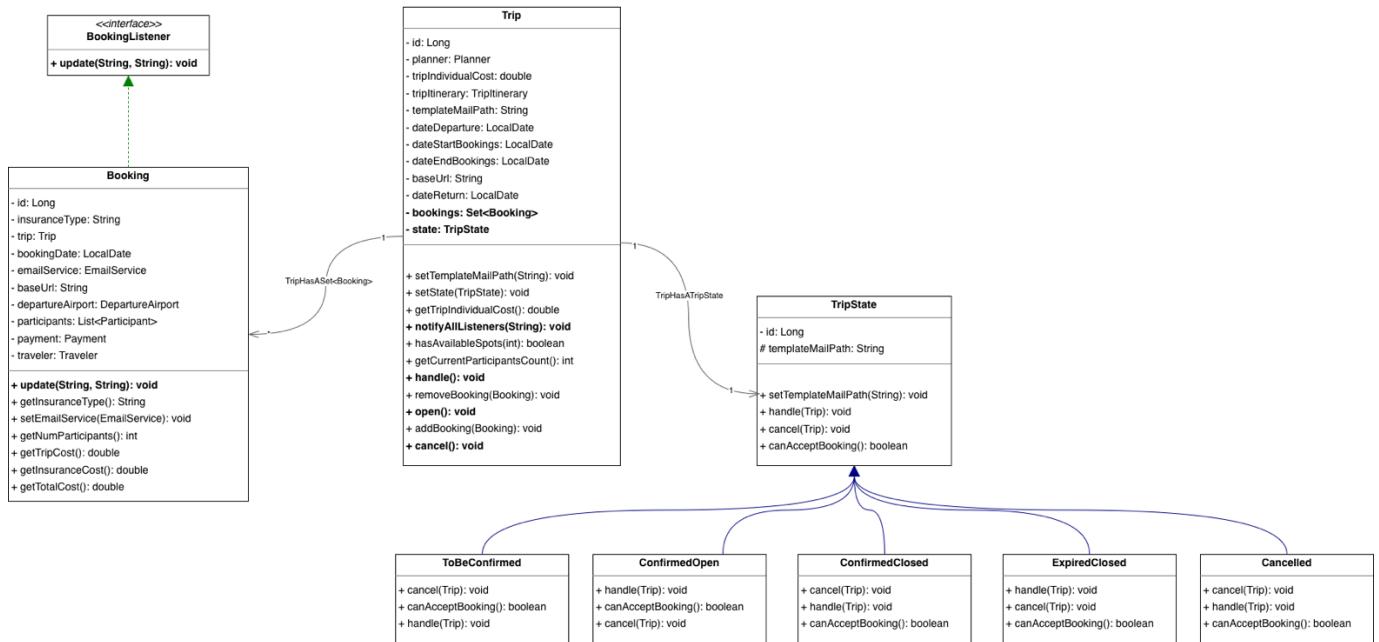
Restituisce un errore 403 quando un utente tenta un'azione non autorizzata (es. un traveler che prova a recensire un viaggio a cui non ha partecipato).

## Utility service

### PasswordGeneratorService

Genera password sicure da utilizzare per nuovi utenti o reset. Le password rispettano criteri stringenti di complessità (maiuscole, minuscole, numeri, caratteri speciali e lunghezza tra 8 e 20). Internamente utilizza metodi di randomizzazione e mescolamento dei caratteri.

## Design Pattern – State Pattern and Observer Pattern



All'interno dell'applicazione *Adventure Together*, la gestione del ciclo di vita di un viaggio (**Trip**) e le notifiche automatiche ai viaggiatori sono implementate tramite l'integrazione di due design pattern complementari:

- **State Pattern** → per modellare i diversi stati operativi del Trip e i relativi comportamenti;
- **Observer Pattern** → per notificare automaticamente tutte le prenotazioni (**Booking**) quando un Trip cambia stato.

I due pattern operano in maniera coordinata: quando il Trip cambia stato (State Pattern), vengono attivati i Booking associati (Observer Pattern), che inviano email ai viaggiatori interessati.

Questa combinazione permette di mantenere il codice modulare, chiaro e scalabile, semplificando la gestione di logiche complesse come transizioni di stato, aggiornamenti automatici e notifiche email.

### 1. State Pattern — Modellazione del ciclo di vita del Trip

#### Trip = Context

La classe Trip rappresenta il **contesto** del pattern. Include:

- le date di apertura/chiusura prenotazioni,
- le date di viaggio,
- il costo individuale,

- i riferimenti all'itinerario e al planner,
- la collezione di Booking associati,
- e soprattutto lo stato corrente:

```
@OneToOne(cascade = CascadeType.ALL)
```

```
private TripState state;
```

Il Trip non conosce la logica dei singoli stati e delega tutto allo stato corrente tramite metodi come:

```
public void handle() { state.handle(this); }
public void cancel() { state.cancel(this); }
public boolean canAcceptBooking() { return state.canAcceptBooking(); }
```

### Transizioni automatiche

La logica della transizione (es. conferma del viaggio, chiusura iscrizioni, scadenza) è completamente incapsulata negli stati concreti.

Il Trip si limita a chiamare handle() in momenti chiave, ad esempio:

- dopo una nuova prenotazione,
- tramite gli scheduler (es. TripScheduler)

### TripState = Superclasse astratta

TripState definisce l'interfaccia comune per tutti gli stati:

```
public abstract void handle(Trip trip);
public abstract void cancel(Trip trip);
public boolean canAcceptBooking() { return false; }
```

Inoltre contiene:

- templateMailPath: il percorso del template email da inviare quando il Trip entra in quello stato.

Grazie all'annotazione **@Inheritance(SINGLE\_TABLE)**, tutti gli stati sono salvati nella stessa tabella *trip\_states*, e vengono distinti tramite una colonna discriminante.

### Stati concreti del Trip

Gli stati implementati sono:

- **ToBeConfirmed** → stato iniziale
- **ConfirmedOpen** → viaggio confermato, iscrizioni aperte
- **ConfirmedClosed** → viaggio confermato, iscrizioni chiuse
- **ExpiredClosed** → chiuso per mancato raggiungimento del minimo partecipanti
- **Cancelled** → cancellato dall'amministratore

Ogni stato:

- implementa la propria logica di transizione (nel metodo handle()),
- definisce se accettare nuove prenotazioni (canAcceptBooking()),
- decide se la cancellazione è permessa,
- assegna il templateMailPath necessario per notificare i viaggiatori.

### Esempi:

- In **ToBeConfirmed**:

- se vengono raggiunti i min. partecipanti → *ConfirmedOpen*
  - se vengono raggiunti i max. partecipanti → *ConfirmedClosed*
  - se scade la data di prenotazione senza minimo → *ExpiredClosed*
- In **ConfirmedOpen**:
    - se vengono raggiunti i max. partecipanti → *ConfirmedClosed*
    - se scade la data prenotazioni → *ConfirmedClosed*
  - In stati conclusivi (**ConfirmedClosed**, **ExpiredClosed**, **Cancelled**): nessuna è permessa nessuna transizione ulteriore.

## 2. Observer Pattern — Notifica automatica dei Booking

Il pattern State viene completato dal **pattern Observer**, utilizzato per informare automaticamente i viaggiatori quando lo stato del Trip cambia.

### **BookingListener = Interfaccia Observer**

```
public interface BookingListener {
    void update(String mailTemplatePath, String baseUrl);
}
```

Qualsiasi oggetto che deve reagire ai cambi di stato implementa questo contratto.

### **Booking = Concrete Observer**

La classe Booking implementa BookingListener:

```
public class Booking implements BookingListener {
    @Override
    public void update(String template, String urlHomePage) {
        emailService.sendHtmlMessage(...);
    }
}
```

Ogni Booking:

- è automaticamente registrato come osservatore del Trip,
- quando riceve la notifica, invia un'email al suo Traveler, usando il template corrispondente allo stato del Trip.

### **Trip = Subject**

Il Trip mantiene una collezione di osservatori (i Booking associati):

```
@OneToMany(mappedBy = "trip")
private Set<Booking> bookings;
```

Quando il Trip cambia stato, viene chiamato il metodo di Trip **notifyAllListeners()**:

```
public void notifyAllListeners(String mailTemplatePath) {
    for (Booking booking : bookings) {
        booking.update(mailTemplatePath, baseUrl + "/home");
    }
}
```

Questo attiva il metodo **update()** su ogni Booking.

```

@Override
public void update(String mailTemplatePath, String urlHomePage) {
    if (emailService == null) {
        System.err.println("[WARN] EmailService non configurato per Booking.update()");
        return;
    }
    emailService.sendHtmlMessage(
        traveler.getEmail(),
        "Aggiornamento sul tuo viaggio " + trip.getTriptinerary().getTitle(),
        mailTemplatePath,
        Map.of("traveler", traveler, "trip", trip, "homepage", urlHomePage)
    );
}

```

I Booking useranno:

- templateMailPath per il tipo di notifica,
- baseUrl per comporre link e pulsanti nelle email,
- i dati del Trip per personalizzare il contenuto.

### **Flusso integrato State + Observer**

Ecco come i due pattern lavorano insieme:

1. Una nuova prenotazione viene creata → booking.addBooking() → potrebbe cambiare il numero di partecipanti.
2. Il servizio richiama trip.handle() → lo stato corrente valuta le condizioni e decide se effettuare una transizione.
3. Il Trip cambia stato → trip.setState(newState) → aggiorna templateMailPath.
4. Il Trip notifica tutti i Booking associati → trip.notifyAllListeners(template).
5. Ogni Booking riceve update() → invia una email al proprio viaggiatore (es. conferma viaggio, chiusura iscrizioni, viaggio scaduto...).

### **Benefici dell'integrazione State + Observer**

#### **Separazione totale delle responsabilità**

- State → gestisce logica e transizioni del Trip
- Observer → gestisce notifiche e side effects

#### **Nessuna logica duplicata**

Il Trip non conosce il contenuto delle email. Gli stati non conoscono i Booking. I Booking non conoscono la logica degli stati.

#### **Manutenibilità**

Aggiungere un nuovo stato (es. *Suspended*) o un nuovo tipo di notifica richiede modifiche minime e localizzate.

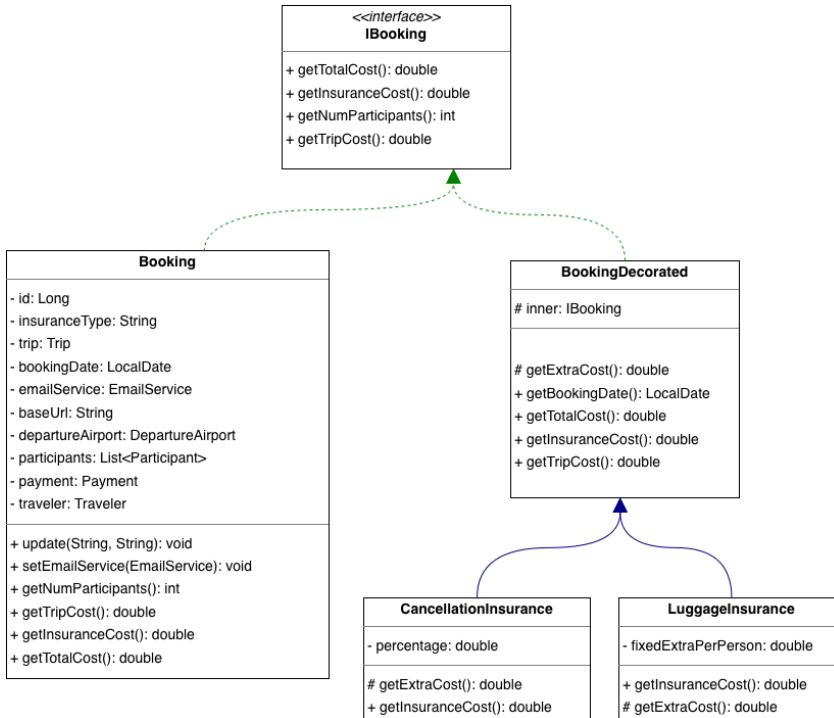
#### **Reattività automatica**

Qualsiasi aggiornamento sul Trip genera notifiche coerenti, senza bisogno di chiamate aggiuntive nei servizi.

#### **Coerenza dell'esperienza utente**

I traveler ricevono sempre comunicazioni aggiornate e pertinenti al loro viaggio.

## Design Pattern – Decorator Pattern



Nel sistema *Adventure Together*, il calcolo del costo totale di una prenotazione (**Booking**) varia in base alle assicurazioni aggiuntive selezionate dal viaggiatore (bagaglio, cancellazione, ecc.). Per mantenere questo meccanismo estendibile, modulare e conforme ai principi **SOLID**, è stato adottato il **Decorator Pattern**.

Questo pattern permette di aggiungere funzionalità alla prenotazione in modo incrementale, senza modificare la classe Booking e consentendo la composizione libera di più decoratori.

### 1. Componente base: Booking

La classe Booking implementa l'interfaccia IBooking e definisce:

#### Costo del viaggio

Dipende dal numero di partecipanti:

```

public double getTripCost() {
    return trip.getTripIndividualCost() * Math.max(1, getNumParticipants());
}
  
```

#### Assicurazione base (10% del costo del viaggio)\*

Questo è il comportamento fondamentale, prima di ogni decorazione:

```

public double getInsuranceCost() {
    return this.getTripCost() * 0.10;
}
  
```

\* la percentuale 10% è stata scelta puramente in maniera arbitraria

#### Costo totale della prenotazione

Di default:

```

public double getTotalCost() {
    return getTripCost() + getInsuranceCost();
}
  
```

```
}
```

Questa implementazione è volutamente semplice, in modo da delegare ai decoratori eventuali estensioni (bagaglio, cancellazione, ecc.).

## 2. Decorator astratto: BookingDecorated

Il decoratore astratto incapsula un'altra istanza di IBooking e delega ad essa i metodi di base:

```
public abstract class BookingDecorated implements IBooking {  
    protected final IBooking inner;  
  
    @Override public double getTripCost() { return inner.getTripCost(); }  
  
    @Override public double getInsuranceCost() { return inner.getInsuranceCost(); }  
  
    protected double getExtraCost() { return 0.0; }  
  
    @Override  
    public double getTotalCost() {  
        return getTripCost() + getInsuranceCost();  
    }  
}
```

Il decoratore:

- non altera il costo del viaggio,
- non altera il numero di partecipanti,
- altera solo la parte assicurativa, aggiungendo costi extra tramite `getExtraCost()`.

## 3. Decoratori concreti

I decoratori estendono BookingDecorated e ridefiniscono il metodo `getInsuranceCost()`, applicando la formula generale:

```
insuranceCostDecorated = insuranceCostBase + costoExtraSpecifico
```

### Luggage (assicurazione bagaglio)

Aggiunge un costo fisso per partecipante.

Metodo chiave:

```
@Override  
public double getInsuranceCost() {  
    return inner.getInsuranceCost() + this.getExtraCost();  
}  
  
@Override  
protected double getExtraCost() {  
    return fixedExtraPerPerson * Math.max(1, inner.getNumParticipants());  
}
```

Quindi, se la prenotazione ha 3 partecipanti e `fixedExtraPerPerson = 20.0`:

`LuggageExtraCost = 20 * 3 = 60€`

`InsuranceTotal = InsuranceBase + 60€`

### Cancellation (assicurazione cancellazione)

Aggiunge una percentuale del costo del viaggio (es. 5%).

Metodo chiave:

```
@Override  
public double getInsuranceCost() {  
    return inner.getInsuranceCost() + this.getExtraCost();  
}  
  
@Override  
protected double getExtraCost() {  
    return inner.getTripCost() * percentage;  
}
```

Se il viaggio costa 1000€, il supplemento è:

CancellationExtraCost =  $1000 * 0.05 = 50\text{€}$

InsuranceTotal = InsuranceBase + 50€

#### 4. Composizione dei decoratori

Il grande vantaggio del Decorator Pattern è che i decoratori si possono combinare liberamente:

Esempio:

Assicurazione base + bagaglio + cancellazione

```
IBooking booking = new Booking(...);  
booking = new LuggageInsurance(booking);  
booking = new CancellationInsurance(booking);  
double total = booking.getTotalCost();
```

Il risultato sarà:

TotalCost = TripCost + InsuranceBase + LuggageExtra + CancellationExtra

Ogni decoratore aggiunge solo la propria parte, senza interferire con gli altri.

#### 5. Benefici della soluzione adottata

- Estendibilità

Nuove assicurazioni → basta creare un nuovo decorator.

- Nessuna modifica alla classe Booking

La classe Booking resta minimal, stabile e chiara.

- Componibilità illimitata

I decoratori possono essere applicati in qualsiasi ordine.

- Uniformità dell'interfaccia

Il controller e i servizi utilizzano sempre IBooking (polimorfismo totale).

#### Casi Rilevanti – Booking process

Il flusso di prenotazione rappresenta uno dei componenti centrali della piattaforma *Adventure Together* e costituisce l'intersezione tra diversi sottosistemi fondamentali: gestione dei viaggi, gestione degli utenti, logiche di stato, transazioni economiche e comunicazioni automatizzate.

Proprio per questa complessità, il processo di booking è un elemento chiave dell'architettura complessiva e merita una descrizione dettagliata.

Il flusso di prenotazione del sistema è strutturato in modo modulare e transazionale per garantire integrità dei dati, affidabilità dei pagamenti e corretta sincronizzazione con il ciclo di vita del viaggio (Trip). Il processo si divide in **due macro-fasi**:

- FASE 1 — Avvio della prenotazione e creazione del PaymentIntent (checkout Stripe)
- FASE 2 — Conferma del pagamento e finalizzazione della prenotazione

FASE 1 — Avvio della prenotazione e creazione del PaymentIntent (checkout Stripe)

#### 1.1 Accesso al form di prenotazione

Endpoint:

GET /bookings/new/{tripId}

BookingController.showBookingForm():

1. Recupera dettagli del Trip (TripDTO) e del suo Itinerary.
2. Ottiene l'ID del traveler autenticato.
3. Prepara un primo BookingDTO contenente:
  - tripId
  - travelerId
  - una lista iniziale con almeno un ParticipantDTO
4. Recupera gli aeroporti di partenza ammessi per l'itinerario.
5. Mostra il form booking/booking-form.html.

#### 1.2 Invio del form e avvio del checkout

Endpoint:

POST /bookings/checkout

BookingController.startCheckout():

- Esegue validazione del BookingDTO via Bean Validation.
- In caso di errori → redirect con binding errors.
- In caso di dati validi:

delega totalmente la fase preparatoria a BookingPreparationServiceImpl.startBookingAndPayment().

#### 1.3 Logica applicativa della fase preliminare

Implementata in:

BookingPreparationServiceImpl.startBookingAndPayment()

Questa è la fase più importante prima del pagamento.

a) Recupero e validazione delle entità correlate

- Trip
- Traveler
- DepartureAirport

b) Verifica disponibilità

- trip.getState().canAcceptBooking()
- trip.hasAvailableSpots(req.getParticipants().size())

Il sistema rifiuta prenotazioni se:

- lo stato del Trip non accetta nuove prenotazioni;
- non ci sono posti disponibili.

c) Conversione dei ParticipantDTO in entity *non ancora persistite*

Servono unicamente per:

- creare un oggetto Booking transitorio,
- calcolare correttamente i costi con il Decorator Pattern.

d) Creazione del Booking transitorio

```
Booking temp = req.toEntity(trip, traveler, airport, participants);
```

Non viene salvato nel database.

e) Calcolo dinamico del costo tramite Decorator Pattern

La piattaforma supporta quattro tipi di assicurazione:

<i>insuranceType</i>	<i>Decorator applicato</i>
basic	nessuno
cancellation	new CancellationInsurance(...)
luggage	new LuggageInsurance(...)
full	entrambi in cascata: new CancellationInsurance(new LuggageInsurance(...))

Il calcolo avviene tramite:

```
IBooking decorated = decorations.get(...).apply(temp);
```

```
double totalCost = decorated.getTotalCost();
```

#### 1.4 Gestione dei "Temporary Participants"

Un punto critico del progetto riguarda i limiti di dimensione dei metadata Stripe:

Stripe accetta metadati solo *di piccole dimensioni* (~500-1.000 caratteri).

Non è possibile serializzare tutta la lista dei partecipanti come JSON nel PaymentIntent.

Soluzione adottata → utilizzo di **TemporaryParticipantList** e **TemporaryParticipant**.

Funzionamento:

1. Si crea una TemporaryParticipantList.
2. Per ogni participant del booking vengono create entità TemporaryParticipant.
3. La lista contiene solo i dati minimi indispensabili (nome, cognome, data di nascita).
4. Nei metadati Stripe viene salvato solo l'ID della lista temporanea:

```
participantsTempListId = {id}
```

In questo modo:

- non si viola il limite di payload dei metadata,
- i dati rimangono nel DB per essere recuperati dai webhook.

#### 1.5 Serializzazione dei metadata per Stripe

I metadata inviati al PaymentIntent contengono:

```
{  
    tripId,  
    travelerId,  
    departureAirportId,  
    insuranceType,  
    numParticipants,  
    participantsTempListId,  
    totalCost  
}
```

Il tutto viene serializzato tramite:

```
bookingSerializerService.serializeBooking(metadataMap)
```

e inviato allo StripeClient.

### 1.6 Creazione del PaymentIntent con Stripe

```
stripeClient.createPaymentIntent(  
    totalCost,  
    "eur",  
    metadataJson  
)
```

Lo StripeClient invia:

- amount (in centesimi)
- currency
- automatic\_payment\_methods
- metadata (incluso JSON con le informazioni della prenotazione)

Ritorna:

```
PaymentIntentDTO(paymentIntentId, clientSecret, totalCost)
```

Il controller reindirizza l'utente alla pagina booking/booking-checkout.html.

## FASE 2 — Conferma del pagamento e finalizzazione della prenotazione

Questa fase è **asincrona** ed è completamente gestita tramite **webhook Stripe**.

### 2.1 Ricezione del webhook

Endpoint:

```
POST /stripe/webhook
```

StripeWebhookController.handleStripeWebhook():

1. Verifica la firma con il webhook secret.
2. Deserializza l'evento tramite Stripe SDK.
3. Delega a:

```
stripeWebhookService.processEvent(event)
```

### 2.2 Processamento dell'evento

StripeWebhookService gestisce due casi:

- payment\_intent.succeeded → chiama BookingFinalizeService.finalizeBooking()
- payment\_intent.payment\_failed → logga e termina

### 2.3 Finalizzazione

Implementata in:

```
BookingFinalizeServiceImpl.finalizeBooking(PaymentIntent intent)
```

STEP 1 — Recupero dei metadati

Dal PaymentIntent si estrae:

```
String metadataJson = intent.getMetadata().get("booking")
Map<String, Object> data = deserializeBookingAsMap(metadataJson)
```

Si recuperano:

- tripId
- travelerId
- departureAirportId
- insuranceType
- numParticipants
- participantsTempListId

STEP 2 — Recupero delle entità reali

Si ricaricano dal database:

- Trip
- Traveler
- DepartureAirport
- TemporaryParticipantList + lista TemporaryParticipant

STEP 3 — Ricostruzione definitiva dei Participant

Ogni TemporaryParticipant diventa un Participant (entity persistente).

STEP 4 — Costruzione del Booking definitivo

```
Booking booking = new Booking();
booking.setTrip(trip);
booking.setTraveler(traveler);
booking.setDepartureAirport(airport);
booking.setParticipants(participants);
booking.setInsuranceType(insuranceType);
```

Viene applicato di nuovo il **decorator pattern** per calcolare correttamente *totalCost* e *insuranceCost*.

STEP 5 — Creazione dell'entità Payment

```
Payment payment = new Payment();
payment.setStatus("PAID");
payment.setPaymentIntentId(intent.getId());
payment.setPaymentDate(LocalDate.now());
payment.setAmountPaid(totalCost);
```

```
payment.setAmountInsurance(insuranceCost);
```

```
payment.setCurrency("eur");
```

Associazione:

```
booking.setPayment(payment);
```

STEP 6 — Persistenza nel DB

```
bookingRepository.save(booking);
```

Grazie alle cascade:

- vengono salvati automaticamente i Participant
- viene salvato Payment

Infine viene eliminata la TemporaryParticipantList (cleanup).

#### 2.4 Aggiornamento dello stato del Trip — State Pattern

Subito dopo il salvataggio della prenotazione:

```
TripState before = trip.getState();  
trip.handle();  
if (!before.getClass().equals(trip.getState().getClass())) {  
    tripRepository.save(trip);  
}
```

La chiamata a handle() permette al Trip di cambiare stato a seconda delle diverse circostanze.

Il sistema salva lo stato solo se è effettivamente cambiato.

#### 2.5 Invio email al traveler

Dopo la finalizzazione:

```
emailService.sendHtmlMessage(..., "/mail/booking-confirmation", ...)
```

Parallelamente - se lo stato cambia a seguito di handle() - lo State Pattern può attivare notifiche a tutti i booking di un trip tramite:

```
trip.notifyAllListeners(templatePath)
```

Template Thymeleaf

L'applicazione utilizza **Thymeleaf** come motore di template, con le viste organizzate nella directory:

```
src/main/resources/templates/
```

La struttura è suddivisa per aree funzionali:

- **/admin** – pagine dedicate ad Admin e Planner, incluse le viste per la gestione dei dati di dettaglio (GeoArea, Category, Country, DepartureAirport) e degli utenti.
- **/auth** – login, registrazione e recupero password.
- **/booking** – form di prenotazione, checkout con Stripe e pagine di successo/fallimento.
- **/error** – pagine collegate al GlobalExceptionHandler per i diversi codici HTTP.
- **/mail** – template HTML per le email (conferme, notifiche di stato, reminder).

Le pagine includono un **fragment comune** (la navbar) tramite:

```
<div th:replace="~{fragments/navbar :: mainNavbar}"></div>
```

Questo garantisce uniformità grafica e riduce la duplicazione di codice.

## Application Properties e Environment Variables

I parametri di configurazione dell'applicazione sono definiti nel file **application.properties**, che raccoglie tutte le impostazioni necessarie al funzionamento del sistema. Oltre alla configurazione standard della connessione al database MySQL e alle proprietà JPA, sono presenti:

- le impostazioni del modulo email, utilizzato per l'invio delle notifiche attraverso SMTP Gmail;
- i parametri custom dell'app, come il percorso base, la configurazione degli upload e la gestione delle immagini;
- le impostazioni per la gestione degli errori e i limiti degli upload multipart.

Le credenziali sensibili (DB, mail, Stripe) non sono definite direttamente nel file properties, ma caricate tramite environment variables esterne (**.env**), garantendo maggiore sicurezza e separazione tra configurazioni locali e di deploy.

## Dipendenze Maven

Le dipendenze utilizzate dal progetto sono definite nel file **pom.xml**, gestito dal plugin Maven, fondamentale per la build, l'esecuzione locale e l'eventuale packaging in un file **jar**. La struttura del POM riflette tutte le esigenze funzionali dell'applicazione.

### Dipendenze principali

- **Spring Boot Starter Web, Thymeleaf, Security e Validation:** costituiscono il nucleo della web application, gestendo routing, rendering delle pagine, autenticazione/authorization e validazione dei dati in input.
- **Spring Boot Starter Data JPA + MySQL Driver:** abilitano la persistenza tramite Hibernate e l'accesso al database MySQL.
- **Spring Boot Starter Mail:** gestisce l'invio delle email (SMTP Gmail).

### Altre dipendenze essenziali

- **Jackson datatype JSR310:** supporto alla serializzazione/deserializzazione delle date (LocalDate, LocalDateTime), indispensabile per il dominio del progetto.
- **Stripe Java SDK:** utilizzato per la creazione e gestione dei PaymentIntent durante il processo di prenotazione.
- **Lombok:** riduce la boilerplate attraverso annotazioni come @Getter, @Setter, @Builder.

### Dipendenze per l'ambiente di test

- **Spring Boot Starter Test e Spring Security Test:** strumenti completi per unit testing, mocking, verifica della sicurezza e test delle API.
- **JaCoCo plugin:** genera report di code coverage in fase di build.

## Piano di Test

Il progetto adotta una strategia di **Unit Testing estensiva**, allineata alle linee guida del corso, con l'obiettivo di validare:

- la logica di business dei servizi,
- la correttezza delle operazioni sui controller,
- la gestione delle eccezioni,
- la coerenza del dominio nei casi limite,
- l'integrazione tra componenti interne (senza coinvolgere database reale o chiamate HTTP esterne).

Ogni test è progettato per isolare con precisione il componente in analisi, simulando le dipendenze tramite mock.

## Strategia di testing

L'intera suite è organizzata in package paralleli alla struttura del codice sorgente:

- **controller** (validazione input, redirect, caricamento dati modello)
- **service** (logica applicativa, manipolazione delle entità, eccezioni)
- **scheduler** (verifica dei task pianificati)

- **config & util** (classi di supporto e inizializzazione)

### **Esclusioni**

Non vengono testati:

- repository JPA (trattandosi di semplici interfacce)
- DTO puramente strutturali
- template Thymeleaf
- chiamate reali a Stripe o Gmail (sempre mockate)
- upload file tramite multipart (validato a livello di service, non controller)

### Strumenti usati

#### *JUnit 5*

Framework principale per l'esecuzione dei test, utilizzato per:

- definizione dei casi (@Test)
- setup e teardown (@BeforeEach)
- asserzioni (assertEquals, assertThrows, ...)

#### *Mockito*

Usato in combinazione con l'estensione JUnit

```
@ExtendWith(MockitoExtension.class)
```

per:

- creare mock delle dipendenze (@Mock)
- iniettare automaticamente i mock nel componente da testare (@InjectMocks)
- simulare comportamenti (when(...).thenReturn(...))
- verificare interazioni (verify(service).create(...))

L'assenza di un database reale è garantita da repository completamente mockati.

### Esempi di Test

#### *Esempio per Service: TripItineraryServiceImplTest*

Questa classe valida una logica articolata che include:

- ricerca, validazione e caricamento di entità correlate (Planner, Country, Airport, Category)
- controllo degli invarianti (min/max partecipanti, durata, congruenza giorni)
- gestione di file upload
- validazione di vincoli custom (duplicati, formato immagine, dimensioni)
- persistenza dell'entità e delle sue relazioni interne

### Caso di successo

```
@Test
void createItinerary_success() {
    TripItineraryDTO dto = baseValidDto();
    mockValidRelations();
    when(itineraryRepository.existsByTitle("Test Trip")).thenReturn(false);
    when(itineraryRepository.save(any())).thenAnswer(i -> {
```

```

Triptinerary t = i.getArgument(0);
t.setId(99L);
return t;
});

TriptineraryDTO result = service.createItinerary(dto);
assertNotNull(result);
assertEquals(99L, result.getId());
verify(itineraryRepository).save(any());
}

```

### Caso di errore (titolo duplicato)

```

@Test
void createItinerary_duplicateTitle_throws() {
    TriptineraryDTO dto = baseValidDto();
    when(itineraryRepository.existsByTitle(dto.getTitle())).thenReturn(true);
    assertThrows(DuplicateResourceException.class,
        () -> service.createItinerary(dto));
}

```

### Validazione business rule (min/max errati)

```

@Test
void createItinerary_invalidMinMax_throws() {
    TriptineraryDTO dto = baseValidDto();
    dto.setMinParticipants(10);
    dto.setMaxParticipants(5);

    assertThrows(DataIntegrityException.class,
        () -> service.createItinerary(dto));
}

```

Questa varietà di scenari verifica completamente la robustezza del servizio.

*Esempio per Controller: PlannerControllerTest*

I test sui controller validano:

- corretto caricamento dei dati nel modello
- gestione della validazione
- corretta selezione della view
- gestione delle eccezioni tramite redirect e messaggi flash

### GET: caricamento pagina

```

@Test
void showCreateTriptineraryForm_loadsData() {
    Model model = new ConcurrentModel();
}

```

```

when(auth.getName()).thenReturn("test@mail.com");
when(userService.getByEmail("test@mail.com"))
    .thenReturn(Optional.of(UserDTO.builder().id(55L).build()));
when(countryService.getAllCountries()).thenReturn(List.of());
when(categoryService.getAllCategories()).thenReturn(List.of());
when(departureAirportService.getAllDepartureAirports()).thenReturn(List.of());
String view = controller.showCreateTriptineraryForm(model, auth);
assertEquals("planner/create-trip-itinerary", view);
assertTrue(model.containsAttribute("triptineraryDTO"));
}

```

#### **POST: validazione fallita**

```

@Test
void createTriptinerary_validationErrors() {
    when(bindingResult.hasErrors()).thenReturn(true);
    RedirectAttributes attrs = new RedirectAttributesModelMap();
    TriptineraryDTO dto = new TriptineraryDTO();
    String result = controller.createTriptinerary(dto, bindingResult, attrs);
    assertEquals("redirect:/planner/create-trip-itinerary", result);
    assertTrue(attrs.getFlashAttributes().containsKey("triptineraryDTO"));
}

```

#### **POST: eccezione dal service**

```

@Test
void createTriptinerary_exceptionHandled() {
    when(bindingResult.hasErrors()).thenReturn(false);
    doThrow(new DuplicateResourceException("ERR"))
        .when(itineraryService).createtinerary(any());
    RedirectAttributes attrs = new RedirectAttributesModelMap();
    String result = controller.createTriptinerary(new TriptineraryDTO(), bindingResult, attrs);
    assertEquals("redirect:/planner/create-trip-itinerary", result);
    assertTrue(attrs.getFlashAttributes().containsKey("errorMessage"));
}

```

Questi test assicurano che il controller si comporti correttamente in tutti i casi previsti.

#### *Strategia Complessiva*

La strategia generale adottata può quindi essere riassunta così:

- Test basati su casi reali (happy path)
- Test su casi limite e edge case: dati mancanti, ID inesistenti, relazioni nulle
- Test sulle eccezioni personalizzate
- Mock approfondito delle dipendenze esterne

- Totale isolamento del business layer
- No dipendenza da database o servizi esterni (Stripe, Gmail)
- Supporto di JaCoCo per misurare la copertura

## Sintesi dei Test e copertura

L'applicazione è stata verificata tramite una suite di **unit test** basata su JUnit e Mockito, con l'obiettivo di controllare il corretto funzionamento della logica nei servizi, controller e componenti più critici (prenotazioni, gestione trip, pagamenti Stripe, validazioni). L'esecuzione ha prodotto **301 test superati** senza errori o failure, coprendo sia scenari di successo sia condizioni di errore.

I report JaCoCo indica una copertura complessiva dell'**83% delle istruzioni** e del **66% dei rami**, con livelli più elevati nei servizi core e nei controller principali. Le parti con copertura inferiore riguardano componenti strutturali (DTO, entity, handler) che presentano logiche minime o prive di comportamenti significativi da testare unitariamente.

Nel complesso, i risultati ottenuti confermano che le funzionalità principali dell'applicazione sono state testate in modo estensivo e che la base del progetto risulta stabile e coerente rispetto ai requisiti implementati.

## adventure-together

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxt	Missed	Lines	Missed	Methods	Missed	Classes
com.lucamoretti.adventure_together.service.trip.impl	79%	52%	44	107	46	267	7	56	0	3	3	3
com.lucamoretti.adventure_together.config	16%	1	100%	12	14	38	50	12	13	3	4	4
com.lucamoretti.adventure_together.model.trip.state	61%	29%	17	34	22	63	6	22	0	6	0	6
com.lucamoretti.adventure_together.controller.advice	11%	n/a	7	9	18	22	7	9	1	2	1	2
com.lucamoretti.adventure_together.service.booking.impl	87%	78%	9	30	8	129	7	23	0	4	0	4
com.lucamoretti.adventure_together.dto.user	66%	n/a	4	11	20	66	4	11	0	4	0	4
com.lucamoretti.adventure_together.dto.trip	82%	39%	18	31	7	81	2	12	0	3	0	3
com.lucamoretti.adventure_together.dto.details	59%	50%	5	10	14	41	3	8	0	4	0	4
com.lucamoretti.adventure_together.model.user	34%	n/a	7	9	12	15	7	9	2	4	0	4
com.lucamoretti.adventure_together.controller.booking	78%	75%	3	7	14	49	2	5	1	2	1	2
com.lucamoretti.adventure_together.controller.admin	91%	87%	5	28	10	108	3	20	1	3	0	3
com.lucamoretti.adventure_together.controller.traveler	87%	80%	2	10	7	51	0	5	0	1	0	1
com.lucamoretti.adventure_together.model.booking.decorator	61%	n/a	7	13	7	17	7	13	1	3	0	3
com.lucamoretti.adventure_together.controller.planner	94%	75%	4	19	4	102	1	13	0	1	0	1
com.lucamoretti.adventure_together.service.customUserDetails.impl	0%	n/a	3	3	4	4	3	3	1	1	1	1
com.lucamoretti.adventure_together.util.exception	74%	n/a	1	6	2	15	1	6	0	5	0	5
com.lucamoretti.adventure_together.dto.review	78%	50%	3	5	0	14	0	2	0	1	0	1
com.lucamoretti.adventure_together.model.trip	91%	72%	5	20	1	33	1	9	1	2	0	2
com.lucamoretti.adventure_together.service.review.impl	96%	80%	4	25	2	68	0	15	0	1	0	1
com.lucamoretti.adventure_together.controller.booking	0%	n/a	2	2	3	3	2	2	1	1	1	1
com.lucamoretti.adventure_together.controller.auth	95%	100%	0	12	2	36	0	7	0	1	0	1
com.lucamoretti.adventure_together.dto.booking	96%	60%	4	7	1	35	0	2	0	1	0	1
com.lucamoretti.adventure_together.model.booking	94%	83%	1	9	2	16	0	6	0	1	0	1
com.lucamoretti.adventure_together.controller.dto.payment	89%	50%	1	2	0	12	0	1	0	1	0	1
com.lucamoretti.adventure_together.service.user.impl	100%	100%	0	22	0	98	0	17	0	1	0	1
com.lucamoretti.adventure_together.service.details.impl	100%	100%	0	22	0	67	0	18	0	4	0	4
com.lucamoretti.adventure_together.service.payment	100%	100%	0	14	0	51	0	9	0	3	0	3
com.lucamoretti.adventure_together.controller.trips	100%	75%	1	7	0	38	0	5	0	1	0	1
com.lucamoretti.adventure_together.scheduler	100%	100%	0	10	0	34	0	7	0	3	0	3
com.lucamoretti.adventure_together.util.passwordGenerator	100%	100%	0	7	0	19	0	5	0	1	0	1
com.lucamoretti.adventure_together.service.validation	100%	100%	0	16	0	22	0	9	0	1	0	1
com.lucamoretti.adventure_together.service.mail.impl	100%	n/a	0	2	0	20	0	2	0	1	0	1
com.lucamoretti.adventure_together.controller.home	100%	83%	1	6	0	15	0	3	0	1	0	1
com.lucamoretti.adventure_together.dto.participant	100%	n/a	0	2	0	12	0	2	0	1	0	1
com.lucamoretti.adventure_together.service.payment.impl	100%	n/a	0	2	0	3	0	2	0	1	0	1
com.lucamoretti.adventure_together.model.auth	100%	n/a	0	1	0	1	0	1	0	1	0	1
Total	1,163 of 6,944	83%	122 of 361	66%	170	534	244	1,677	75	352	12	78