Object Design Document

EasyCoach

Cronologia delle revisioni

Data	Versione	Descrizione	Autori
13/01/2025	0.1	Introduzione, Object design goals, Linee guida per la documentazione dell'interfaccia	NV, FD, SF, RF
16/01/2025	0.2	Packages e diagrammi: Package Easy coach, Package Autenticazione, Package AreaUtente, Package Admin, Package Sessione, Package Prenotazione	NV, SF
19/01/2025	0.3	Rappresentazione Service di ciascun Package dei sottosistemi principali	NV, FD, SF, RF
24/01/2025	0.4	Continuazione della	NV, FD, SF, RF

Data	Versione	Descrizione	Autori
		rappresentazione dei service	
27/01/2025	0.5	Design pattern con grafici	NV, FR
29/01/2025	0.6	Glossario e indice	SF
30/01/2025	1.0	Revisione finale	NV, SF, FD, RF

Membri del Team

Nome	Ruolo del progetto	Acronimo	Informazioni di Contratto
Nello Valentino	Membro del team	NV	n.valentino6@studenti.unisa.it
Simone Fausto	Membro del team	SF	s.fausto5@studenti.unisa.it
Francesco D'Arco	Membro del team	FD	f.darco6@studenti.unisa.it
Roberto Fiorenza	Membro del team	RF	r.fiorenza3@studenti.unisa.it

1.Introduzione

Il presente documento ha lo scopo di descrivere in maniera dettagliata l'architettura e il design dei componenti software del sistema EasyCoach. Il documento fornisce una panoramica completa delle scelte progettuali, illustrando i package e le classi principali, i design patterns adottati e le interfacce esposte dai vari moduli.

1.1 Object design goals

Modularità e Manutenibilità
Il sistema deve essere strutturato in moduli indipendenti, con chiara separazione tra **business logic**, **data access** e **presentation layer**, in modo da facilitare la manutenzione e l'espandibilità del codice

Alta coesione e Basso accoppiamento
Ogni classe e componente deve avere una responsabilità ben definita e ridurre la dipendenza da altre classi, in modo da migliorare la leggibilità e la riusabilità del codice

Esperienza Utente intuitiva e accessibile
L'interfaccia utente deve essere intuitiva e accessibile, supportando una **navigazione fluida**, **responsive design** e fornendo feedback chiari in caso di errori o di operazioni completate con successo

1.2 Linee guida per la documentazione dell'interfaccia

Di seguito, sono rappresentate le linee guida per la documentazione in modo tale da ottenere un codice **coerente** e **formale** durante la scrittura del progetto:

- **Nomi dei package:** i nomi dei package devono essere scritti in minuscolo, senza spazi o caratteri speciali. Per nomi composti da piu' parole, e' necessario utilizzare il formato **snake_case**
- **Classi DAO:** queste classi devono rispettare il formato **PascalCase** e terminare con il suffisso **DAO** per indicarne il ruolo di accesso ai dati
- **Classi Service:** queste classi devono rispettare il formato **PascalCase** e terminare con il suffisso **Service** per indicare le loro funzionalità
- **Classi Servlet:** queste classi devono rispettare il formato **PascalCase** e terminare con il suffisso **Servlet** per indicare le loro funzionalità
- **Classi Test:** queste classi devono rispettare il formato **PascalCase** e terminare con il suffisso **Test**, devono fare riferimento alla classe la quale si intende testare
- **Nomi dei metodi:** i metodi devono rispettare il formato **camelCase** e devono avere nomi descrittivi, che riflettono chiaramente l'operazione eseguita
- **Nomi delle variabili:** i nomi delle variabili possono seguire sia il formato **camelCase** che il formato **snake_case,** questi devono essere descrittivi
- **Nomi delle JSP:** i nomi delle JSP devono seguire il formato **camelCase**, riflettendo chiaramente il contenuto della pagina
- **Nomi dei fogli di stile:** i nomi dei fogli di stile devono essere scritti in **minuscolo** ed indicare la pagina al quale fanno riferimento
- **Nomi dei file JavaScript:** i nomi dei file JS devono seguire il formato **kebab-case** ed indicare chiaramente la funziona principale del file

1.3 Definizioni, acronimi e abbreviazioni

Di seguito sono riportate alcune definizioni presenti nel documento:

- **ODD:** Object Design Document
- **DAO:** Data Access Object
- **DTO:** Detailed Transfer Object
- **JSP:** JavaServer Pages
- **Service:** classi che incapsulano la logica di business, separate dalla logica di accesso ai dati e dalla presentazione
- **Servlet:** componenti Java che gestiscono le richieste web e generano risposte
- **Naming Conventions:**
    - **kebab-case:** formato in cui le parole sono separate da trattini
    - **camelCase:** formato in cui la prima parola inizia con la lettera minuscola e le successive con la maiuscola
    - **snake_case:** formato in cui le parole sono separate da un underscore
    - **PascalCase:** formato in cui la prima lettera di ogni parola e' maiuscola

1.4 Riferimenti

Di seguito è elencata una lista dei documenti utili durante la lettura

- **Statement Of Work**
- **System Design Document**
- **Object Design Document**
- **Test Plan**
- **Manuale Utente**

2. Packages

Il sistema **EasyCoach** è organizzato in package per garantire una chiara separazione delle responsabilità e facilitare la manutenibilità del codice. La suddivisione ricalca l'architettura **ThreeTier** e segue la struttura standard definita da Maven

.model: contiene il Data Access Layer e le classi di Entità

- ***dao:*** Data Access Layer (DAO) per interazione con il Database
- ***entities:*** classi che rappresentano le entità del dominio
- ***connection:*** classi per la connessione al DataBase e l'inizializzazione delle risorse di persistenza
- ***dto:*** Detailed Transfer Objects utilizzato per arricchire le entità del dominio con informazioni aggiuntive

.sottosistemi: contiene i vari moduli di sistema, ciascuno suddiviso in

- ***Admin:***
    - ***controller:*** gestione delle richieste tramite ***Servlet***
    - ***service:*** gestione della logica di Business
- ***AreaUtente:***
    - ***controller:***
    - ***service:***
- ***Autenticazione:***
    - ***controller:***
    - ***service:***
- ***Prenotazione***
    - ***controller:***
    - ***service:***
- ***Sessione:***
    - ***controller:***
    - ***service:***

.utils: contiene utilities generiche per il progetto

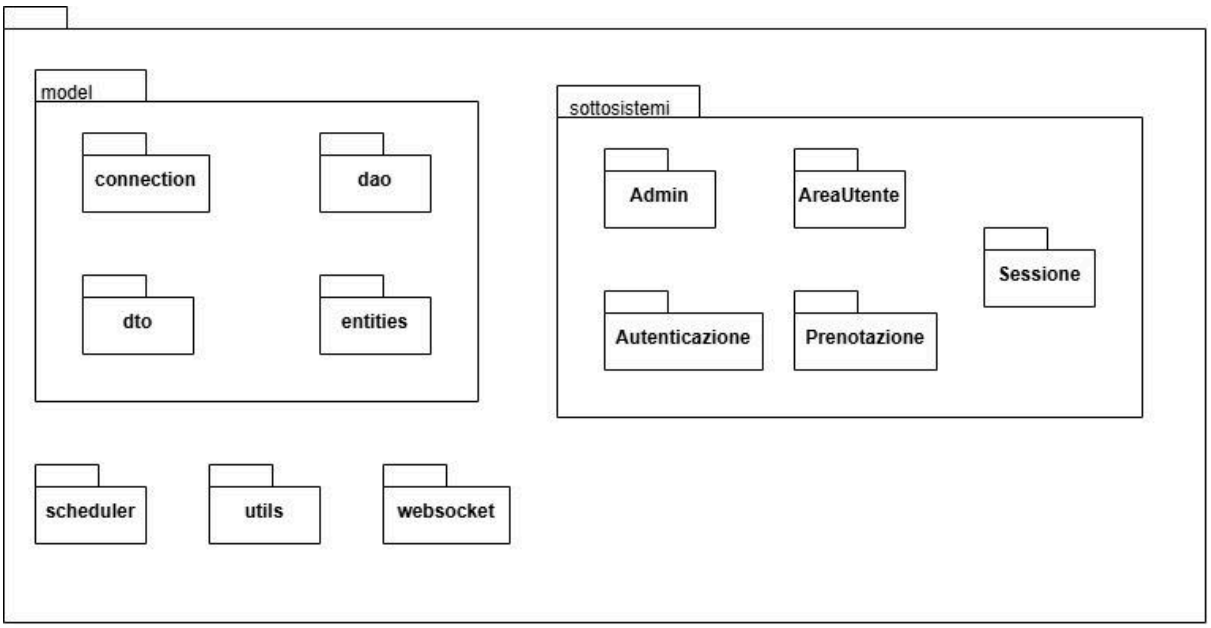
.websocket: implementa un Observer Pattern per la gestione delle notifiche in tempo reale
.scheduler: classi responsabili dell'esecuzione periodica di task automatizzati

Nella presente sezione si mostra la struttura del package principale di **EasyCoach**. La suddivisione generale è stata ottenuta a partire da principali scelte architetturali:

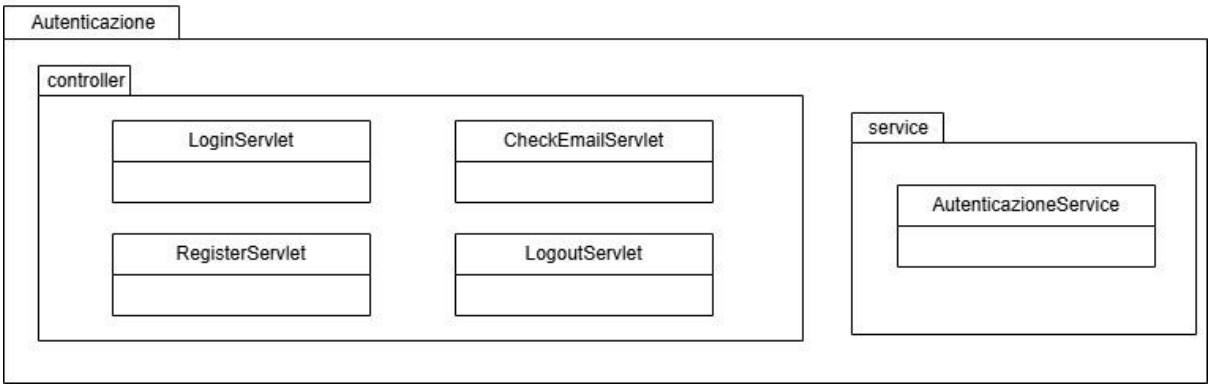
1. **Creare un package separato per ogni sottosistema,** contenente le classi **Service** e **Controller** relative a quel modulo specifico. Questo garantisce una chiara separazione della logica di business e dalle operazioni legate ai diversi aspetti dell'applicazione
2. **Creare un package separato per le classi del model,** contenente le classi **Entity (JavaBeans)** e **DAO** responsabili dell'accesso al Database. Questa scelta è stata adottata per mantenere separata la logica di accesso ai dati e per facilitare la gestione delle relazioni tra le entità del database
3. **Creare un package chiamato *utils,*** in cui sono state inserite eventuali classi di utilità utilizzabili trasversalmente dai vari sottosistemi. Questo package fornisce principalmente funzionalità utili per un corretto indirizzamento utente alla HomePage e per un corretto salvataggio delle immagini
4. **Creare un package *websocket,*** per implementare un **Observer Pattern**, utilizzato per la gestione delle notifiche in tempo reale tra mentor e mentee, migliorando l'interazione dinamica dell'applicazione
5. **Creare un package *scheduler***, dedicato alla gestione delle operazioni pianificate all'interno del sistema. Le classi in questo package si occupano dell'automazione di attività ricorrenti, come la rimozione delle prenotazioni **scadute** (concluse) e l'aggiornamento delle prenotazioni **annullate**

La suddivisione adottata ha portato alla creazione di una relazione diretta tra il package ***model*** e tutti i package del sottosistema, poiché le entità e i DAO forniscono i dati ai **Service.**

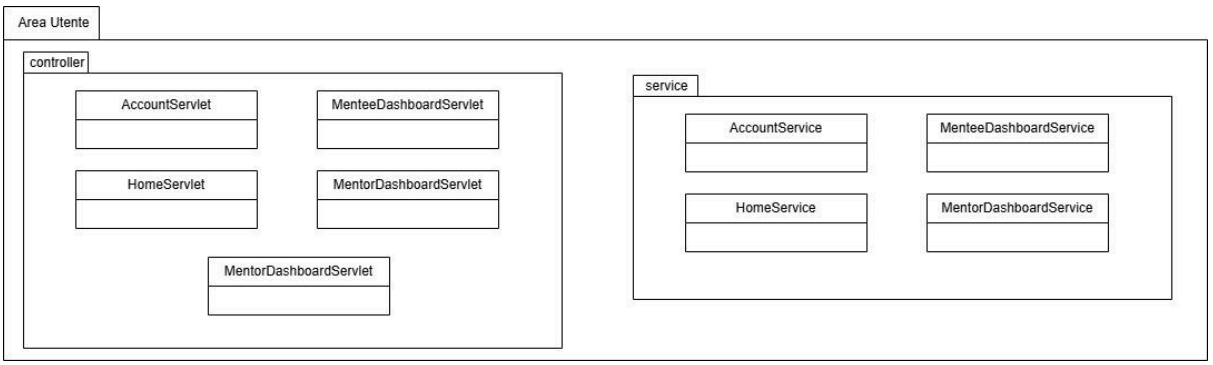
Package EasyCoach



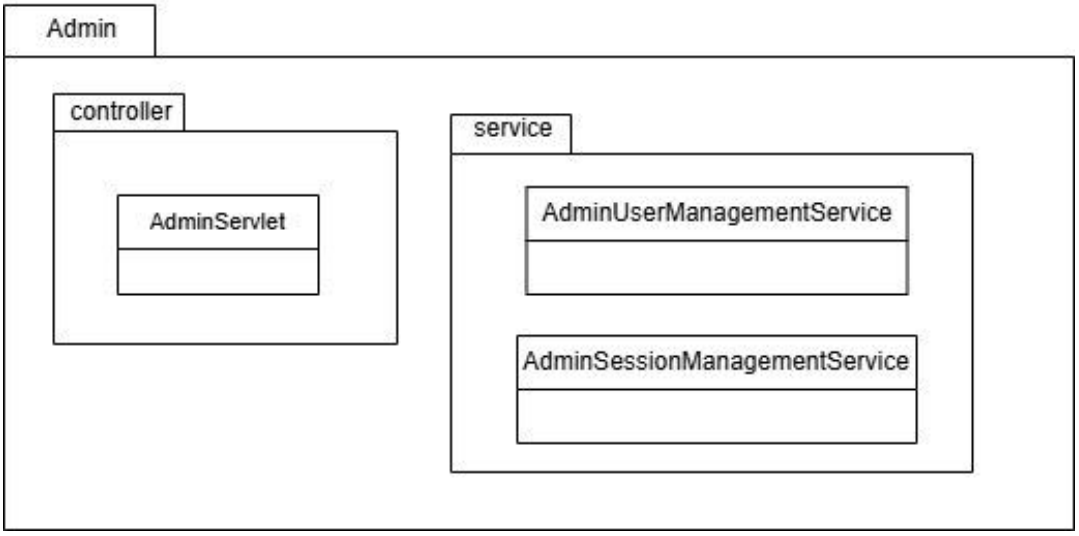
Package Autenticazione



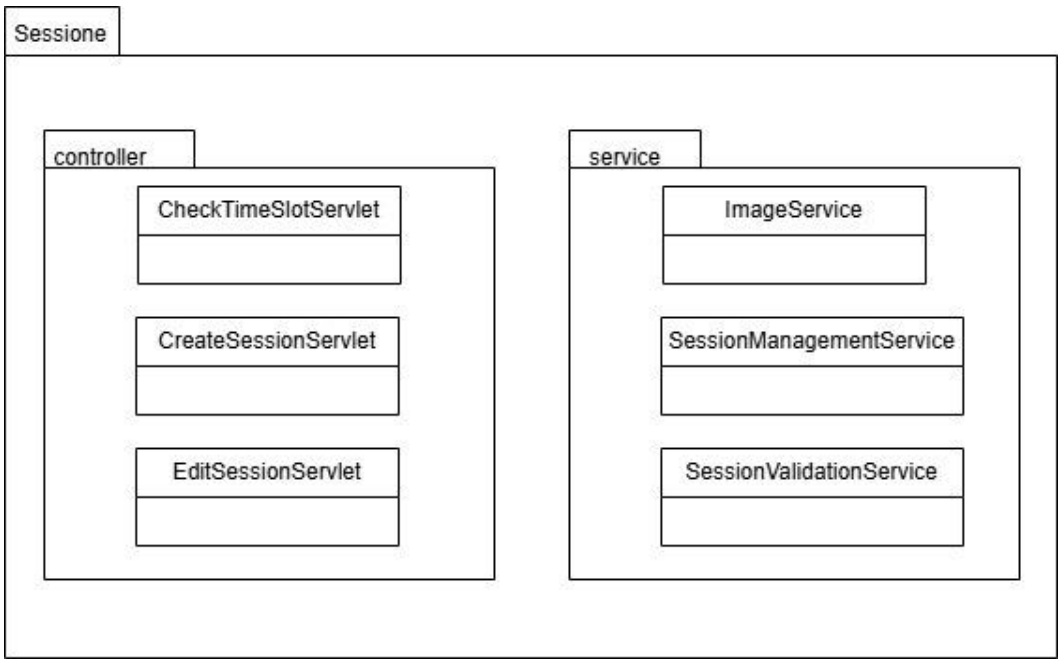
Package AreaUtente



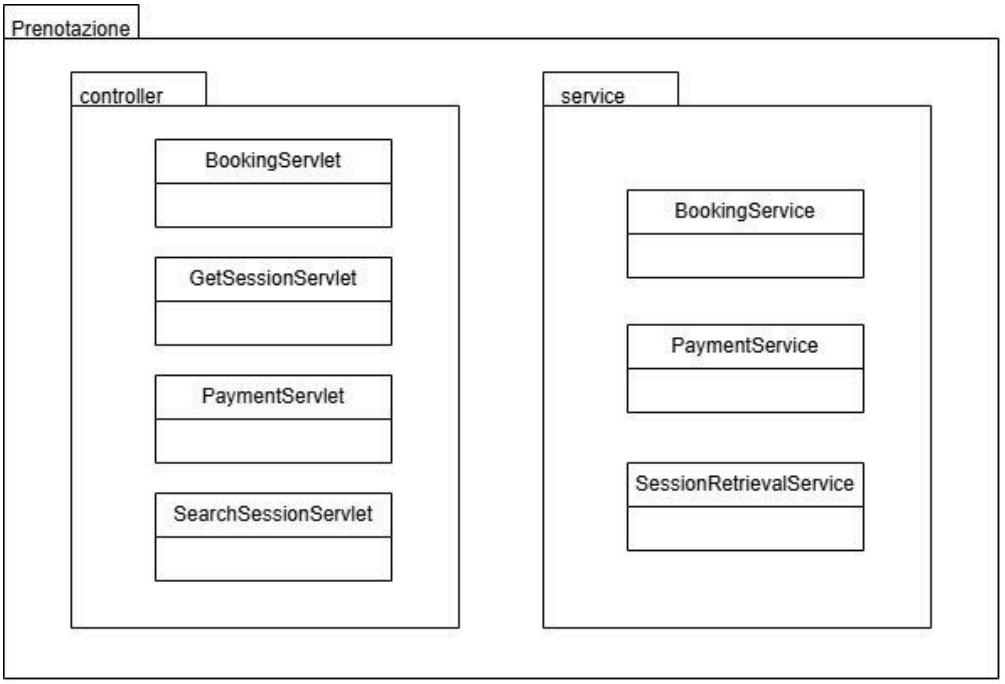
Package Admin



Package Sessione



Package Prenotazione



3. Class Interfaces

Di seguito sono rappresentati Service di ciascun Package dei sottosistemi principali.

Package Admin

Nome classe	**AdminSessionManagementService**
Descrizione	Service dedicato alla gestione delle sessioni e relative operazioni
Metodi	getAllSessionsEnriched() deleteSession(int)
Invariante di classe	/

Nome Metodo	**getAllSessionsEnriched**
Descrizione	Recupera tutte le sessioni dal DB e arricchisce ciascuna con il nome del mentor
Pre-Condizione	/
Post-Condizione	***List<Map<String, Object>>:*** lista di mappe con informazioni arricchite delle sessioni
Nome Metodo	**deleteSession**
Descrizione	Archivia (logicamente) una sessione tramite la chiamata a archiveSession.
Pre-Condizione	***int sessionId:*** ID della sessione da archiviare
Post-Condizione	/

Nome classe	**AdminUserManagementService**

Descrizione	Service dedicato alla gestione degli utenti e relative operazioni
Metodi	getAllUsersSimplified() deleteUser(int)
Invariante di classe	/

Nome Metodo	**getAllUsersSimplified**
Descrizione	Recupera tutti gli utenti dal DB e ne produce una lista di mappe con i campi di interesse
Pre-Condizione	/
Post-Condizione	***List<Map<String, Object>>:*** lista di mappe con informazioni semplificate degli utenti
Nome Metodo	**deleteUser**
Descrizione	Elimina un utente dal DB, archiviando prima tutte le sessioni
Pre-Condizione	***int userId:*** ID dell'utente da eliminare
Post-Condizione	/

Package Area Utente

Nome classe	**AccountService**
Descrizione	Service per la gestione dell'account utente
Metodi	hasActiveBookingsForUser (int) deleteUser (int) updateUserPassword (Utente) hashPassword (String)
Invariante di classe	/

Nome Metodo	**hasActiveBookingForUser**
Descrizione	Verifica se l'utente ha prenotazioni attive
Pre-Condizione	***int userId***: ID dell'utente
Post-Condizione	***true***: se l'utente ha prenotazioni attive ***false***: altrimenti
Nome Metodo	**deleteUser**
Descrizione	Elimina un utente dal DB
Pre-Condizione	***int userId:*** ID dell'utente da eliminare
Post-Condizione	/
Nome Metodo	**updateUserPassword**
Descrizione	Aggiorna la password dell'utente
Pre-Condizione	***Utente utente***: oggetto Utente già caricato

Post-Condizione	/
Nome Metodo	**hashPassword**
Descrizione	Hasha la password (SHA-256) in formato esadecimale
Pre-Condizione	***String password:*** la password già parzialmente hashata
Post-Condizione	***String hashedPassword:*** la password nuovamente hashata

Nome classe	**HomeService**
Descrizione	Service per la gestione della logica della homepage
Metodi	getSessioniInEvidenza() getMentorCasuali()
Invariante di classe	/

Nome Metodo	**getSessioniInEvidenza**
Descrizione	Recupera una lista di tre sessioni casuali
Pre-Condizione	/
Post-Condizione	***List<Sessione> list:*** lista di tre sessioni
Nome Metodo	**getMentorCasuali**
Descrizione	Recupera una lista di mentor casuali di limite 3
Pre-Condizione	/
Post-Condizione	***List<Utente> list:*** lista di tre mentor casuali

Nome classe	**MenteeDashboardService**
Descrizione	Service per la Dashboard Mentee
Metodi	findActiveBookingsForMentee(int) findCompletedBookingsForMentee(int)
Invariante di classe	/

Nome Metodo	**findActiveBookingsForMentee**
Descrizione	Restituisce le prenotazioni attive per uno specifico mentee
Pre-Condizione	***int menteeId:*** id del mentee
Post-Condizione	***List<PrenotazioneDetailsDTO> list:*** lista di prenotazioni attive
Nome Metodo	**findCompletedBookingsForMentee**
Descrizione	Restituisce le prenotazioni conclude per uno specifico mentee
Pre-Condizione	***int menteeId:*** id del mentee
Post-Condizione	***List<PrenotazioneDetailsDTO> list:*** lista di prenotazioni concluse

Nome classe	**MentorDashboardService**
Descrizione	Service per la Dashboard Mentor
Metodi	findSessionsByMentorId(int) findActiveBookingsForMentor(int)
Invariante di classe	/

Nome Metodo	**findSessionsByMentorId**
Descrizione	Restituisce tutte le sessioni ATTIVE relative ad un mentor
Pre-Condizione	***int mentorId:*** id del mentor
Post-Condizione	***List<Sessione> list:*** lista di sessioni attive
Nome Metodo	**findActiveBookingsForMentor**
Descrizione	Restituisce le prenotazioni attive per un mentor
Pre-Condizione	***int mentorId:*** id del mentor
Post-Condizione	***List<PrenotazioneDetailsDTO> list:*** lista di prenotazioni attive per un mentor

Package Autenticazione

Nome classe	**AutenticazioneService**
Descrizione	Servizi utili alla gestione della logica di autenticazione
Metodi	validaInputRegistrazione(String, String, String ,String, String) isEmailRegistrata(String) hashPassword(String) registraNuovoUtente(String, String, String ,String, String) effettuaLogin(String, String) checkEmailExists(String)
Invariante di classe	/

Nome Metodo	**validaInputRegistrazione**
Descrizione	Valida i campi del form di registrazione
Pre-Condizione	***String email, String nome, String cognome,*** ***String hashedPassword, String ruolo:*** campi del form di registrazione
Post-Condizione	***true***: se i campi sono validi ***false***: altrimenti
Nome Metodo	**isEmailRegistrata**
Descrizione	Verifica se esiste già un utente con la email specificata.
Pre-Condizione	***String email:*** email del form di registrazione

Post-Condizione	***true***: se l'email è già presente ***false***: altrimenti
Nome Metodo	**hashPassword**
Descrizione	Esegue la seconda crittografia della password
Pre-Condizione	***String password:*** password già criptata
Post-Condizione	***String hexString:*** la password hashata per la seconda volta
Nome Metodo	**registraNuovoUtente**
Descrizione	Salva un nuovo utente nel database.
Pre-Condizione	***String email, String nome, String cognome,*** ***String doubleHashedPassword, String ruolo***
Post-Condizione	/
Nome Metodo	**effettuaLogin**
Descrizione	Tenta il login di un utente in base a email e password in chiaro.
Pre-Condizione	***String email, String passwordChiara:*** email e password già criptata
Post-Condizione	***true***: l'oggetto Utente se le credenziali sono valide ***false***: altrimenti
Nome Metodo	**checkEmailExists**
Descrizione	Controlla se l'utente (email) esiste nel database.
Pre-Condizione	***String email:*** email dell'utente
Post-Condizione	***true***: se esiste ***false***: altrimenti

Package Prenotazione

Nome classe	**BookingService**
Descrizione	Service per la gestione delle prenotazioni
Metodi	checkAvailability(Map<String, String>) createBooking(Map<String, String>) confirmBooking(Map<String, String>) isTimeslotDisponibile(int, LocalDateTime)
Invariante di classe	/

Nome Metodo	**checkAvailability**
Descrizione	prende la Map di parametri, effettua parse e controlla la disponibilità
Pre-Condizione	***Map<String, String> params***: mappa di parametri
Post-Condizione	***Map<String, Object> availability:*** mappa contenente disponibilità e status
Nome Metodo	**createBooking**
Descrizione	Presa la mappa dei parametri, effettua tutti i controlli, verifica la disponibilità e crea la prenotazione
Pre-Condizione	***Map<String, String> params:*** mappa dei parametri
Post-Condizione	***Prenotazione prenotazione:*** oggetto Prenotazione
Nome Metodo	**confirmBooking**
Descrizione	Conferma la prenotazione (stato "CONCLUSA") partendo dalla

	mappa dei parametri
Pre-Condizione	***Map<String, String> params:*** mappa dei parametri
Post-Condizione	***Prenotazione prenotazione:*** oggetto prenotazione con stato "CONCLUSA"
Nome Metodo	**isTimeslotDisponibile**
Descrizione	Controlla la disponibilità di un timeslot nella data/ora specifica
Pre-Condizione	***int timeslotId, LocalDateTime dateTime:*** ID timeslot e data
Post-Condizione	***true***: se esiste ***false***: altrimenti

Nome classe	**PaymentService**
Descrizione	Service che incapsula la logica di pagamento
Metodi	validateCardPayment(String, String, String, String, String) processPayment(int, String, double, String) sendNotifications(Prenotazione, String)
Invariante di classe	/

Nome Metodo	**validateCardPayment**
Descrizione	Verifica la correttezza dei dati della carta
Pre-Condizione	***String numeroCarta, String scadenzaGGMM,*** ***String scadenzaAnno, String cardHolder, String cvv:*** dati relativi alla carta
Post-Condizione	/

Nome Metodo	**processPayment**
Descrizione	Esegue il processo di pagamento
Pre-Condizione	***int idPrenotazione, String metodoPagamentoParam, double totalePagato, String idUtenteSession:*** dati relativi al completamento di un pagamento
Post-Condizione	***Pagamento pagamento:*** pagamento completato
Nome Metodo	**sendNotifications**
Descrizione	Invia notifiche via WebSocket al mentee e al mentor
Pre-Condizione	***Prenotazione prenotazione, String idUtenteSession:*** prenotazione acquistata e id del Mentor proprietario
Post-Condizione	/

Nome classe	**SessionRetrievalService**
Descrizione	Service che incapsula la logica di recupero Sessioni e relativi timeslot
Metodi	findSessionById(int) findAllSessions() findTimeslotsBySessionId(int) findCorrelatedSessions(int currentSessionId) findSessionsByTitleLike(String)
Invariante di classe	/

Nome Metodo	**findSessionById**
Descrizione	Restituisce la sessione associata ad un determinato Id
Pre-Condizione	***int sessioneId:*** id della sessione
Post-Condizione	***Sessione sessioni:*** sessione restituita
Nome Metodo	**findAllSessions**
Descrizione	Restituisce tutte le sessioni
Pre-Condizione	/
Post-Condizione	***List<Sessione> list:*** lista di tutte le sessioni
Nome Metodo	**findTimeslotsBySessionId**
Descrizione	Restituisce tutti i timeslot associati ad una sessione
Pre-Condizione	***int sessioneId:*** id della sessione
Post-Condizione	***List<Timeslot> list:*** lista di timeslot associati a quella sessione
Nome Metodo	**findCorrelatedSessions**
Descrizione	Restituisce le sessioni "ATTIVA", escludendo la sessione corrente, limitando i risultati a 4
Pre-Condizione	***int currentSessionId:*** id della sessione corrente
Post-Condizione	***List<Sessione> list:*** lista delle sessioni correlate
Nome Metodo	**findSessionsByTitleLike**
Descrizione	Restituisce le sessioni il cui titolo contenga una determinata stringa
Pre-Condizione	***String query:*** input di ricerca
Post-Condizione	***List<Sessione> list:*** lista di sessioni relative alla ricerca

Package Sessione

Nome classe	**ImageService**
Descrizione	Gestisce il caricamento, la validazione e la cancellazione delle immagini.
Metodi	validateImage(Part) processImageUpload(Part, String, String) deleteImage(String, String) getSubmittedFileName(Part)
Invariante di classe	/

Nome Metodo	**validateImage**
Descrizione	Valida la Part (immagine) controllando dimensioni e estensione/mime
Pre-Condizione	***Part filePart:*** immagine da validare
Post-Condizione	***true***: se il file non è vuoto, non supera i 10 mb e ha un'estensione valida ***false***: altrimenti
Nome Metodo	**processImageUpload**
Descrizione	Carica il file in 'permanentUploadPath' e ritorna il path relativo da salvare a DB.
Pre-Condizione	***Part filePart, String permanentUploadPath, String uploadDirectory:*** immagine, path permanente e path di

	salvataggio
Post-Condizione	***String path:*** path relativo
Nome Metodo	**deleteImage**
Descrizione	Elimina il file vecchio, se presente.
Pre-Condizione	***String imagePath, String permanentUploadPath:*** path immagine e path permanente
Post-Condizione	/
Nome Metodo	**getSubmittedFileName**
Descrizione	Restituisce il nome del file originale caricato dall'utente, estraendolo dall'header "content-disposition"
Pre-Condizione	***Part part:*** immagine
Post-Condizione	***String filename:*** nome del file

Nome classe	**SessionManagementService**
Descrizione	Service che gestisce la logica legata a Sessioni e Timeslot
Metodi	getTimeslotsByMentorIdAsMap(int) createSession(Sessione, String[], String[]) findSessionById(int) findTimeslotsBySessionId(int) updateSession(Sessione, String[], String[]) hasActiveBookings(int)

	archiveSession(Sessione)
Invariante di classe	/

Nome Metodo	**getTimeslotsByMentorIdAsMap**
Descrizione	Restituisce una lista di timeslot (giorno/orario) per un determinato mentor
Pre-Condizione	***int mentorId:*** id del mentor
Post-Condizione	***List<Map<String, Integer>> list:*** lista di mappe con campi giorno e orario
Nome Metodo	**createSession**
Descrizione	Crea una nuova sessione e relativi timeslot.
Pre-Condizione	***Sessione session, String[] days, String[] hours:*** sessione, array di giorni, array di ore
Post-Condizione	***int idSessione:*** id della nuova sessione
Nome Metodo	**findSessionById**
Descrizione	Recupera una sessione tramite il suo ID.
Pre-Condizione	***int idSessione:*** id della sessione
Post-Condizione	***Sessione sessione:*** sessione corrispondente
Nome Metodo	**findTimeslotsBySessionId**
Descrizione	Recupera tutti i timeslot associati a una sessione.
Pre-Condizione	***int idSessione:*** id della sessione
Post-Condizione	***List<Timeslot> list:*** lista di timeslot

Nome Metodo	**updateSession**
Descrizione	Aggiorna i campi di una sessione e sostituisce i timeslot con i nuovi.
Pre-Condizione	***Sessione session, String[] days, String[] hours:*** sessione, array di giorni, array di ore
Post-Condizione	/
Nome Metodo	**hasActiveBookings**
Descrizione	Verifica se esistono prenotazioni attive per una sessione.
Pre-Condizione	***int idSessione:*** id della sessione
Post-Condizione	***true***: se ci sono prenotazione attive ***false***: altrimenti
Nome Metodo	**archiveSession**
Descrizione	Archivia una sessione (impostando lo status a "ARCHIVIATA") ed elimina i suoi timeslot.
Pre-Condizione	***Sessione session:*** sessione da archiviare
Post-Condizione	/

Nome classe	**SessionValidationService**
Descrizione	Service per validare i dati di una sessione prima di salvarli nel database.
Metodi	validateTitle(String) validateDescription(String) validatePrice(String)

	validateTimeslots(String[], String[]) validateForm(String, String, String, String[], String[], boolean) validateFormEdit(String, String, String, String[], String[])
Invariante di classe	/

Nome Metodo	**validateTitle**
Descrizione	Valida il titolo (2-25 caratteri).
Pre-Condizione	***String title:*** titolo da validare
Post-Condizione	***true***: se il titolo è accettabile ***false***: altrimenti
Nome Metodo	**validateDescription**
Descrizione	Valida la descrizione (2-250 caratteri).
Pre-Condizione	***String description:*** descrizione da validare
Post-Condizione	***true***: se la descrizione è accettabile ***false***: altrimenti
Nome Metodo	**validatePrice**
Descrizione	Se il prezzo è un numero valido tra 0.01 e 999 con max 2 decimali.
Pre-Condizione	***String priceStr:*** prezzo da validare
Post-Condizione	***true***: se il prezzo è accettabile ***false***: altrimenti
Nome Metodo	**validateTimeslots**

Descrizione	Valida i timeslot. 0 <= Days < 7, 0 <= Hours < 24
Pre-Condizione	***String[] days, String[] hours:*** giorni e ore da validare
Post-Condizione	***true***: Se days hours non sono null, hanno la stessa lunghezza, e i valori sono validi ***false***: altrimenti
Nome Metodo	**validateForm**
Descrizione	Metodo “completo” per validare tutti i campi (CreateSession).
Pre-Condizione	***String title, String description, String priceStr,String[] days, String[] hours, boolean imageProvided:*** tutti i campi da validare di una sessione
Post-Condizione	***Map<String, String> map:*** mappa di messaggi di errore
Nome Metodo	**validateFormEdit**
Descrizione	Variante di validateForm per la modifica(dove l'immagine può essere assente).
Pre-Condizione	***String title, String description,String priceStr, String[] days, String[] hours:*** tutti i campi da validare di una sessione
Post-Condizione	***Map<String, String> map:*** mappa di messaggi di errore

4. Design Patterns

Nella presente sezione si andranno a descrivere e dettagliare i **design patterns** utilizzati nello sviluppo dell'applicativo **EasyCoach**. Per ogni pattern si fornirà:

- Una breve **introduzione teorica**
- **Il problema** che doveva risolvere all'interno del sistema
- Una **spiegazione di come è stato implementato** in EasyCoach
- Una rappresentazione grafica della struttura delle classi che implementano il pattern

Connection Pool (Object Pool Pattern)

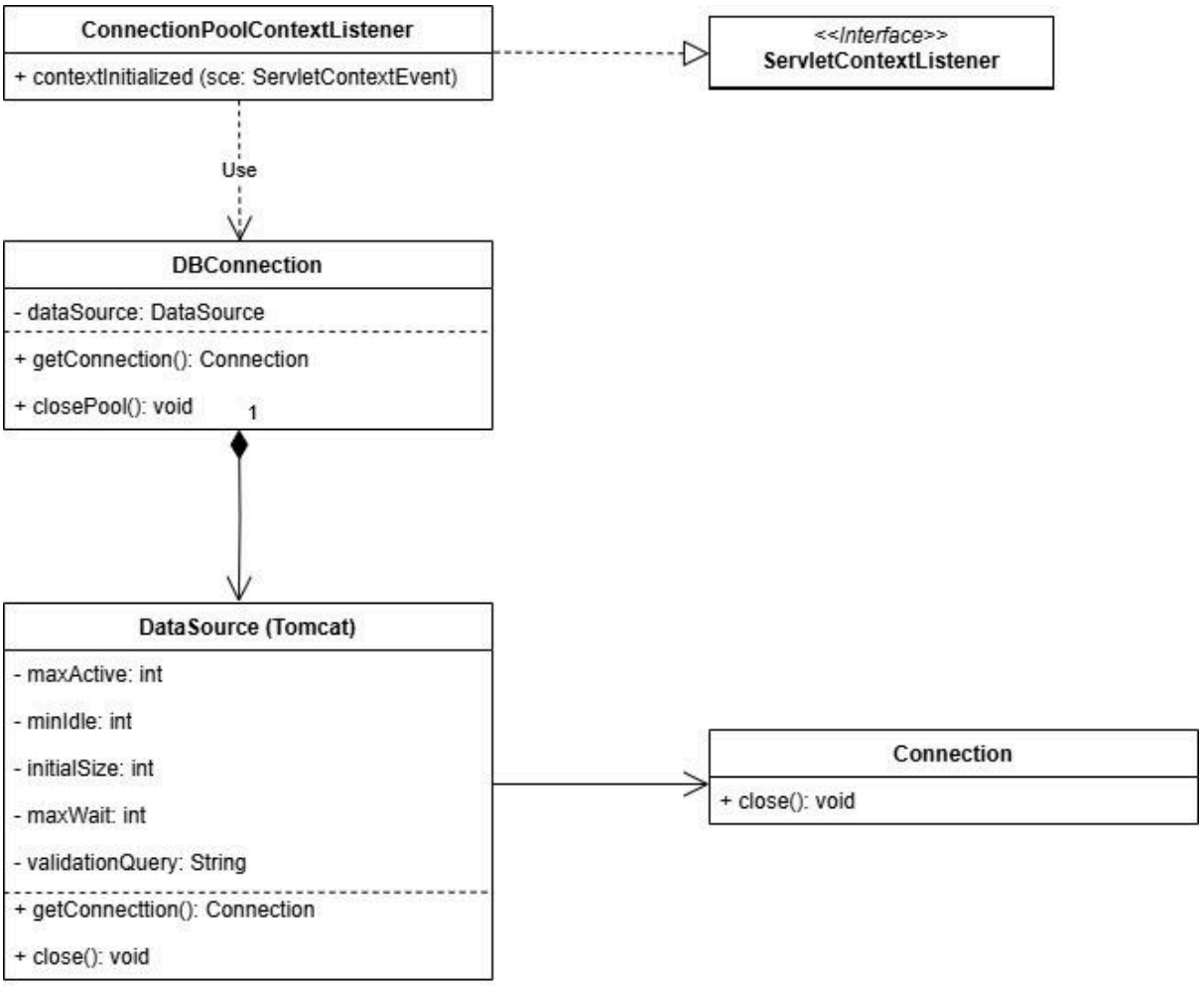
Il **Connection Pooling** è una tecnica di gestione delle connessioni al database che permette di riutilizzare connessioni già esistenti invece di crearne di nuove ogni volta che l'applicazione ha bisogno di interagire con il database. Questo pattern ottimizza le prestazioni riducendo il tempo necessario per stabilire connessioni ripetute e migliorando la gestione delle risorse

L'applicazione **EasyCoach** necessita di frequenti accessi dal database per la gestione di **utenti, prenotazioni** e **sessioni.** Creare una nuova connessione per ogni richiesta sarebbe stato inefficiente e avrebbe causato rallentamenti problematici nonché un uso eccessivo delle risorse

Per risolvere questo problema, è stato implementato un **Connection Pool** nel package ***connection,*** che mantiene un **insieme di connessioni aperte (pool)** e le riutilizza per ogni nuova richiesta. Nel dettaglio il sistema:

- **Recupera una connessione disponibile** dal pool quando necessario
- **Rilascia la connessione** al termine dell'operazione, rendendola disponibile per altri processi
- **Evita di superare il numero massimo di connessioni attive,** ottimizzando le prestazioni. Attualmente questo è impostato a 10

Diagramma Object Pool Pattern



Observer Pattern

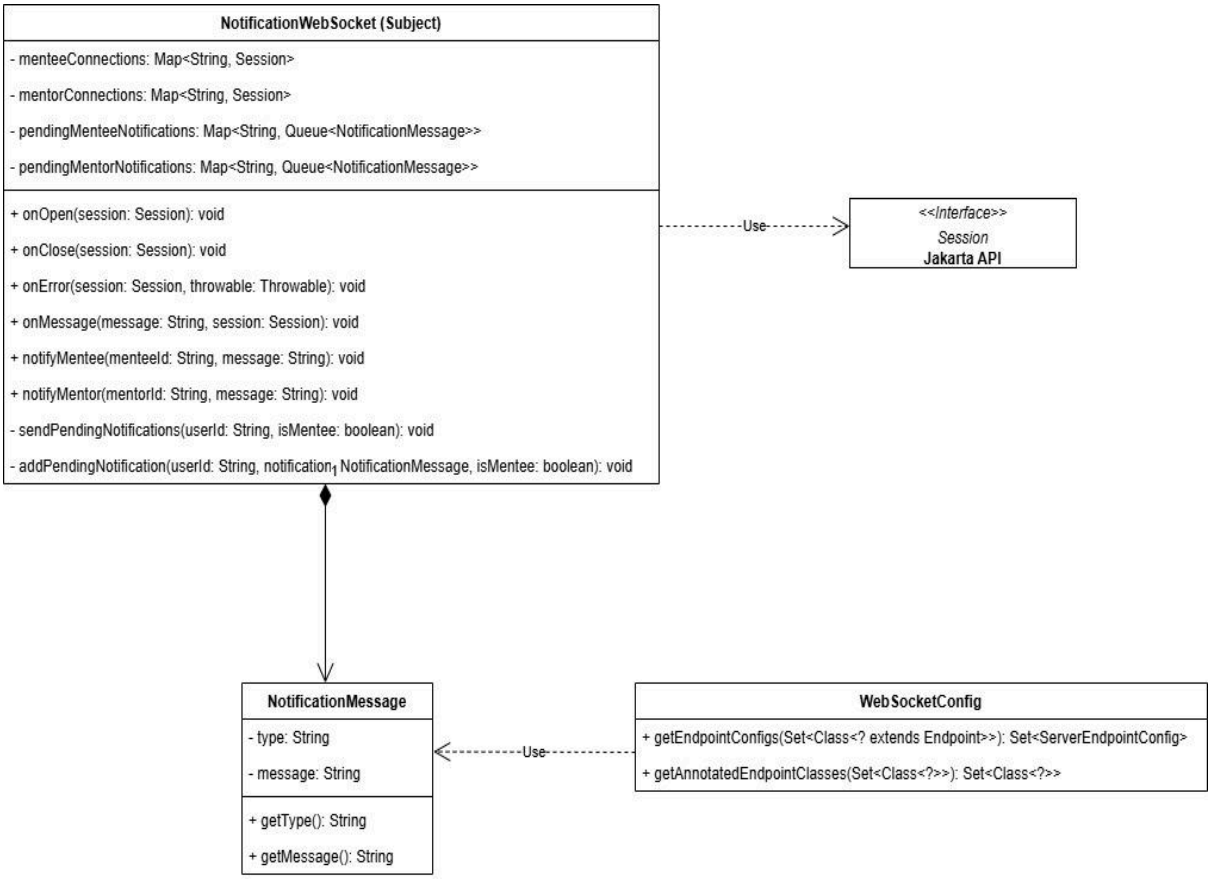
L'**observer pattern** è un **design pattern comportamentale** che permette ad un oggetto (detto **Observable**) di notificare automaticamente più **Observer** registrati quando il suo stato cambia. Questo modello è utile quando diversi componenti devono essere aggiornati in risposta ad un evento

EasyCoach implementa un **sistema di notifiche in tempo reale** per aggiornare mentee e mentor quando una prenotazione viene acquistata con successo. L'invio di notifiche tramite **polling**, ovvero il client che richiede continuamente aggiornamenti, sarebbe stato inefficiente e avrebbe caricato inutilmente il server

Per gestire le notifiche in tempo reale, è stato implementato un **WebSocket Observer** nel package ***websocket*** tramite la classe ***NotificationWebSocket.*** Il sistema funziona come segue:

1. I client **(mentees** e **mentors)** si connettono al WebSocket quando accedono alla piattaforma
2. Il **WebSocket** agisce da **Observable**, mantenendo una lista di sessioni attive, ovvero gli utenti connessi
3. Quando si verifica **l'acquisto di una sessione**, il sistema invia automaticamente una notifica agli osservatori registrati
4. Gli utenti **ricevono la notifica in tempo reale**, senza dover aggiornare manualmente la pagina

Diagramma Observer Pattern



DAO (Data Access Object) Pattern

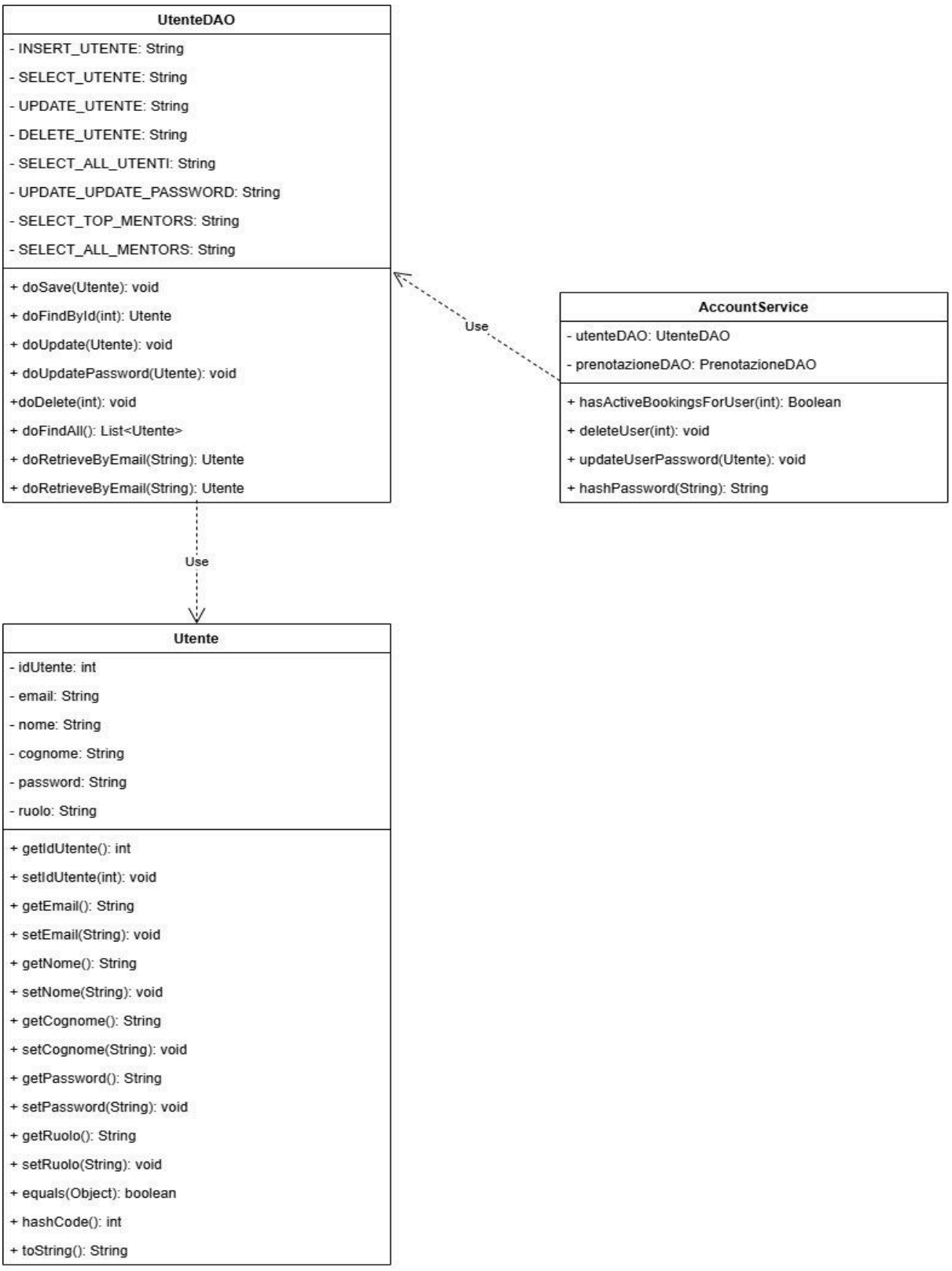
Il **DAO Pattern** è un **design pattern strutturale** che separa la logica di accesso ai dati dal resto dell'applicazione, fornendo un'interfaccia astratta per interagire con il database. Questo modello permette di gestire le operazioni **CRUD** in modo modulare e riutilizzabile

Il sistema EasyCoach gestisce un **database relazionale** con numerose tabelle collegate tra loro. Senza un **DAO separato**, il codice di accesso ai dati sarebbe stato sparso all'interno della logica di business, rendendo difficile la manutenzione e la gestione delle query SQL

Per separare chiaramente la logica di accesso ai dati, è stato creato il package ***dao,*** che contiene classi DAO specifiche per ogni **entità.** Il funzionamento segue questa struttura:

1. I **Service** invocano i **metodi DAO** per recuperare o modificare i dati nel database
2. Le **classi DAO** contengono solo **query SQL** e metodi specifici per accedere al database
3. Le **entità del database** sono **mappate** ai risultati delle query DAO
4. I risultati **vengono passati ai Service**, che applicano la logica di Business prima di inviarli alle **Servlet**

Diagramma DAO Pattern



5. Glossario

Nella presente sezione sono raccolti le sigle o i termini del documento che necessitano di una definizione

Sigla/Termine	Definizione
DAO	Pattern per l'accesso ai dati che separa la logica di business dalla logica di accesso al database
DTO	Oggetto utilizzato per trasportare dati tra i processi
JSP	Tecnologia che aiuta a creare pagine web dinamiche basate su HTML e XML
Service	Classi che incapsulano la logica di business, separate dalla logica di accesso ai dati e dalla presentazione
Servlet	Componenti Java che gestiscono le richieste web e generano risposte
WebSocket	Tecnologia che fornisce un canale di comunicazione bidirezionale full-duplex su una singola connessione TCP
Observable	Pattern di progettazione in cui un oggetto mantiene una lista di dipendenti (observers) e li notifica automaticamente quando cambia stato
Observer	Oggetto che viene notificato quando lo stato dell'Observable cambia
CRUD	Operazioni base per la persistenza dei dati
SQL	Linguaggio standardizzato per database relazionali
Connection Pool	Tecnica di gestione delle connessioni al database che permette di riutilizzare connessioni esistenti
Maven	Strumento di automazione della compilazione utilizzato principalmente per progetti Java
Business Logic	Parte del programma che codifica le regole di business reali e elabora i dati tra il database e l'interfaccia utente
XML	Linguaggio di markup per documenti contenenti informazioni strutturate