



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

Università degli Studi di Firenze

Ingegneria Informatica

Corso di Ingegneria del Software

## SnapShotSpring: Piattaforma per la prenotazione di servizi fotografici

---

**Autore:** Alessandro Capialdi

**Anno accademico:** 2023-2024

### In breve:

Si vogliono applicare le conoscenze acquisite durante il corso di Ingegneria del Software nella creazione di un'applicativo, scritto in Java con l'ausilio del framework Spring Boot, in grado di consentire prenotazione e gestione di servizi fotografici forniti da fotografi freelancer. Il progetto include l'uso di un database, un sistema di registrazione/login e personalizzazione degli utenti.

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Obiettivo . . . . .	3
1.2	Architettura . . . . .	3
<b>2</b>	<b>Analisi e progetto</b>	<b>4</b>
2.1	Casi d'uso . . . . .	4
2.2	Use Case Templates . . . . .	5
2.3	Schema delle classi . . . . .	8
2.4	Entity-Relationship Diagram . . . . .	11
2.5	Pattern architetturale Repository . . . . .	12
<b>3</b>	<b>Implementazione</b>	<b>12</b>
3.1	Package Configuration . . . . .	12
3.1.1	class SecurityConfig . . . . .	12
3.1.2	class CustomAuthenticationSuccessHandler . . . . .	13
3.1.3	class PasswordEncoderConfig . . . . .	15
3.1.4	class MapperConfig . . . . .	15
3.2	Package Domain . . . . .	15
3.2.1	class AppointmentEntity . . . . .	16
3.2.2	class PhotoshootDto . . . . .	18
3.2.3	class LoginDto . . . . .	18
3.3	Package Repositories . . . . .	19
3.3.1	class TimeSlotRepository . . . . .	19
3.4	Package Mappers . . . . .	19
3.4.1	interface Mapper . . . . .	20
3.4.2	Photoshoot Mapper . . . . .	20
3.5	Package Services . . . . .	21
3.5.1	class AbstractCrudService . . . . .	21

3.5.2	class UserService . . . . .	22
3.5.3	class AuthenticationService . . . . .	23
3.5.4	class TimeSlotResetTask . . . . .	24
3.6	Package Controllers . . . . .	25
3.6.1	class AbstractCrudController . . . . .	26
3.6.2	class TimeSlotController . . . . .	28
3.6.3	class AuthController . . . . .	29
3.6.4	class CustomerDashboardController . . . . .	30
3.6.5	class PhotographerDashboardController . . . . .	31
3.7	Package templates . . . . .	31
3.7.1	register.html . . . . .	32
3.7.2	login.html . . . . .	32
3.7.3	photographer-dashboard.html . . . . .	33
3.7.4	manage-photoshoots.html . . . . .	33
3.7.5	book-appointment.html . . . . .	34
<b>4</b>	<b>Test</b>	<b>35</b>
4.1	class UserEntityIntegrationTests . . . . .	35
4.2	class UserServiceTests . . . . .	37
4.3	class AuthenticationServiceTests . . . . .	38
4.4	class UserControllerIntegrationTests . . . . .	38
4.5	Risultati dei test . . . . .	40
<b>5</b>	<b>Conclusioni</b>	<b>41</b>
<b>6</b>	<b>Librerie di terze parti</b>	<b>41</b>

# 1 Introduzione

## 1.1 Obiettivo

Si vuole realizzare un software in grado di gestire prenotazioni da parte di utenti per servizi fotografici erogati da fotografi registrati alla piattaforma. Consideriamo che i clienti dovranno essere in grado di **creare un account personale**, grazie al quale potranno autenticarsi e gestire i propri appuntamenti. Il software dovrà essere in grado di gestirli, considerando giorni ed orari già prenotati da altri utenti e non permettendo la prenotazione in tale scenario. Oltre ai clienti, si dovranno gestire anche i fotografi, che saranno gli **amministratori** in grado di effettuare modifiche alle date ed orari disponibili e modifiche ai servizi fotografici.

## 1.2 Architettura

Di seguito è mostrato lo schema architetturale del progetto e relative dipendenze.

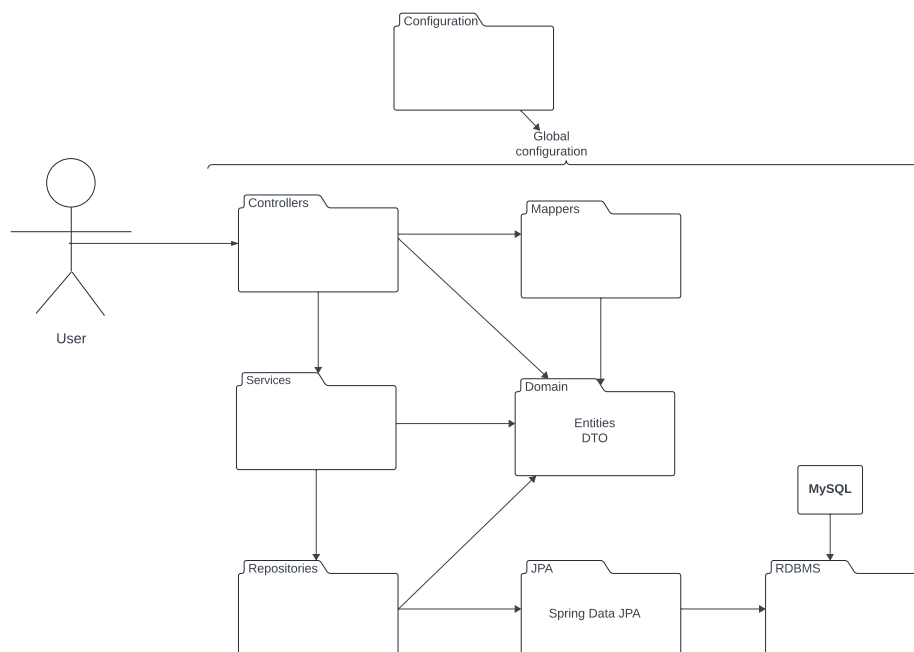


Figure 1: a) Diagramma delle dipendenze

Il software è stato realizzato in Java attraverso l'impiego del framework Spring Boot. Il package *Domain* rappresenta il modello dei dati, con le entità del database e i relativi DTO per il trasferimento dei dati tra i layer; il package *Services* implementa i

servizi dell'applicativo insieme a tutta la business logic necessaria; il package *Repositories* implementa la comunicazione con un database locale MySQL, utilizzando Spring Data JPA (Java Persistence API); infine, abbiamo il package *Controllers* che rappresenta l'application layer, il quale reindirizza a dei template usufruibili dall'utente (fotografo o cliente). Il testing è stato effettuato tramite il framework JUnit. I diagrammi delle classi e dei casi d'uso seguono lo standard UML (Unified Modeling Language) e sono stati realizzati con il software "Lucidchart". Per il database locale è stato utilizzato docker tramite l'immagine "mysql". La configurazione per la connessione è stata inserita in *application.properties* e *docker-compose.yml*. Per la visualizzazione del database ho usato il software *DBeaver*.

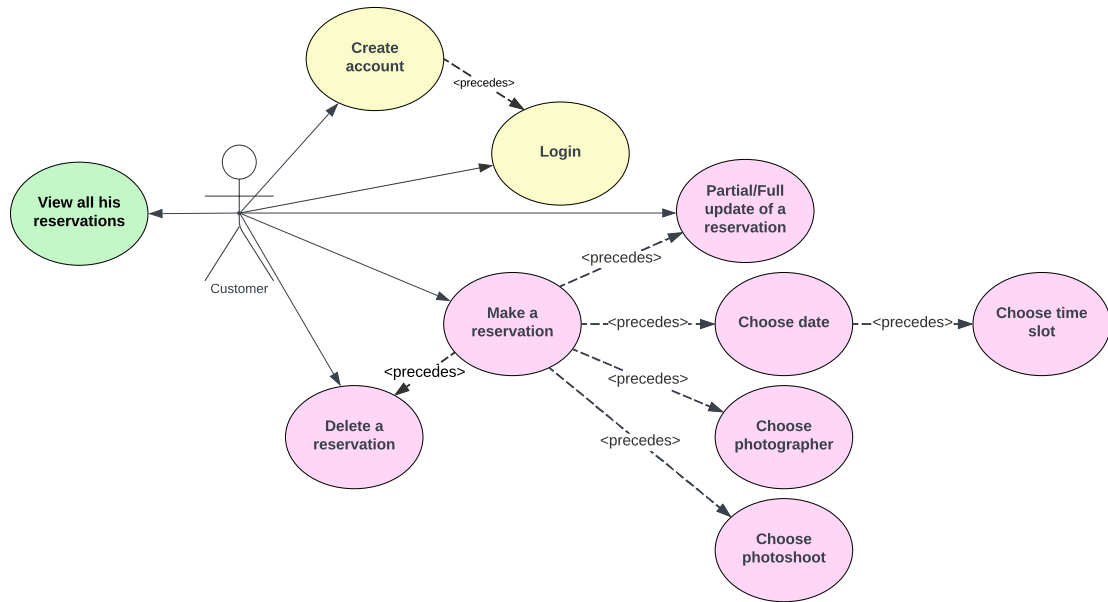
## 2 Analisi e progetto

### 2.1 Casi d'uso

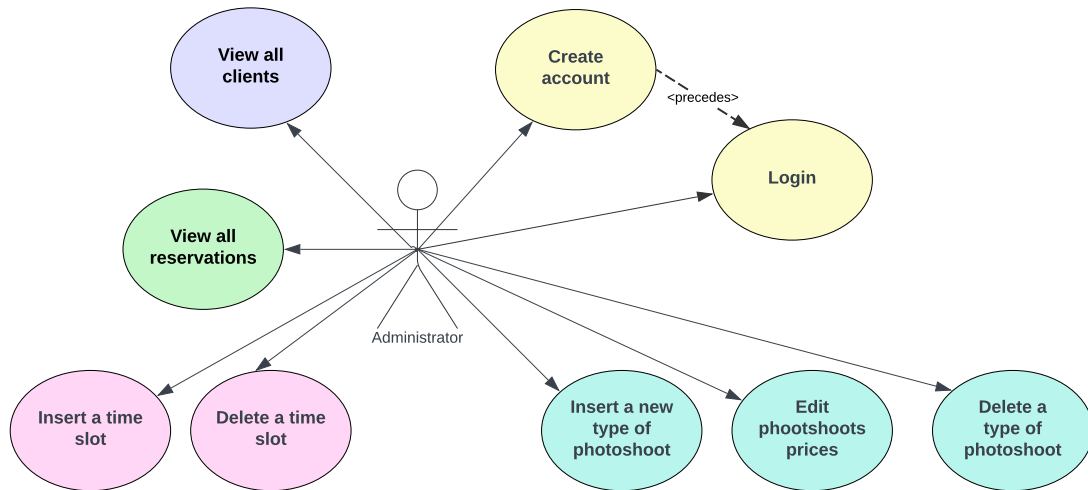
La figura nella pagina successiva mostra il diagramma dei casi d'uso, con i protagonisti coinvolti nel sistema e i modi con cui possono interagirci. Ci sono **2 protagonisti** coinvolti nel sistema:

- **Clienti**, coloro che usufruiscono dei servizi offerti dai fotografi.
- **Fotografi**, i quali possono effettuare alcune operazioni di gestione e visualizzare le prenotazioni dei clienti.

Un *fotografo* ha la possibilità di aggiungere/rimuovere e modificare date e orari prenotabili, aggiungere/rimuovere e modificare servizi fotografici, visualizzare le prenotazioni effettuate dai clienti. Un *cliente* deve avere la possibilità di registrare un account personale e successivamente loggarsi per creare/gestire le proprie prenotazioni. Un utente può anche visualizzare tutti gli appuntamenti da lui creati.



(a) Customer Use Cases



(b) Administrator Use Cases

## 2.2 Use Case Templates

Di seguito sono riportati i template per alcuni dei casi d'uso implementati. Ognuno di questi ha una descrizione dettagliata, i protagonisti coinvolti e il percorso necessario alla realizzazione del caso. Per alcuni di essi, sono riportati percorsi alternativi e pre/post condizioni.

Use Case 1	Nuova prenotazione
Description	Un cliente prenota un servizio fotografico da un certo fotografo
Level	User goal
Actor	User
Basic reservation	<ol style="list-style-type: none"> <li>1. Il cliente sceglie un fotografo</li> <li>2. In base al fotografo scelto, verranno mostrati photoshoot e orari disponibili.</li> <li>3. Si sceglie un orario</li> <li>4. Si sceglie il servizio fotografico desiderato</li> <li>5. Si conferma la prenotazione.</li> </ol>
Alternative reservation 1	Errore di connessione con il server, invio notifica di errore
Pre-conditions	Il cliente deve essere loggato; ci deve essere almeno un fotografo e un servizio fotografico disponibili
Post-conditions	La data e l'orario della prenotazione non devono risultare disponibili ad altri utenti, salvo cancellazione della stessa

a) Use Case Template 1

Use Case 3	Registrazione
Description	Un utente si registra all'applicazione
Level	User/Admin goal
Actor	User/Photographer
Basic registration	<ol style="list-style-type: none"> <li>1. L'utente sceglie se vuole registrarsi da fotografo o da cliente</li> <li>2. L'utente inserisce le proprie credenziali (nome, cognome, email, password e numero di telefono (opzionale)</li> <li>3. Si controlla la validità dell'email</li> <li>4. Si controlla che l'email non sia già presente</li> <li>5. Si esegue l'hash encoding della password</li> <li>6. Si conferma la registrazione e si invia un'email di conferma</li> </ol>
Alternative registration 1	L'email ha un formato sbagliato e si notifica l'errore
Alternative registration 2	L'utente è già registrato, si notifica un errore
Alternative registration 3	C'è stato un errore di connessione al server, si annulla l'operazione e si notifica errore
Post-conditions	L'utente può eseguire il login

b) Use Case Template 3

Use Case 2	Annullamento prenotazione
Description	Un cliente vuole cancellare una prenotazione
Level	User goal
Actor	User
Basic delete reservation	<ol style="list-style-type: none"> <li>1. L'utente visualizza le sue prenotazioni</li> <li>2. Sceglie la prenotazione desiderata</li> <li>3. La prenotazione viene cancellata</li> </ol>
Alternative delete reservation 1	L'utente non ha prenotazioni
Alternative delete reservation 2	Errore di connessione con il server, invio notifica di errore
Pre-conditions	L'utente deve avere almeno una prenotazione attiva

b) Use Case Template 2

Use Case 4	Inserimento servizio fotografico
Description	Un certo fotografo vuole aggiungere un nuovo servizio fotografico
Level	User goal
Actor	Photographer
Basic photoshoot insert	<ol style="list-style-type: none"> <li>1. Il fotografo visualizza i suoi servizi fotografici</li> <li>2. Il fotografo dà un nome al nuovo servizio</li> <li>3. Si sceglie il prezzo</li> <li>4. Si genera un nuovo ID</li> </ol>
Alternative photoshoot insert 1	Errore di connessione con il server, invio notifica di errore
Post-conditions	Un cliente può adesso prenotare il nuovo servizio fotografico

b) Use Case Template 4

Use Case 5	Eliminazione fascia oraria
Description	Un certo fotografo vuole eliminare uno degli orari che ha messo a disposizione
Level	User goal
Actor	Photographer
Basic time slot delete	<ol style="list-style-type: none"> <li>1. Il fotografo visualizza tutte le fasce orarie</li> <li>2. Sceglie la fascia oraria desiderata da eliminare</li> <li>3. Se un utente aveva eseguito delle prenotazioni in quelle fasce orarie, si notifica la cancellazione per email</li> </ol>
Alternative time slot delete 1	Errore di connessione con il server, invio notifica di errore
Post-conditions	Un cliente non può più prenotare in quella fascia oraria di quel fotografo

b) Use Case Template 5

I casi d'uso riportati sono di livello User goal, che rappresentano un'attività elementare che è possibile svolgere tramite il gestionale.

- Lo use case template a) schematizza il percorso base da effettuare per inviare una prenotazione di un servizio fotografico, con percorsi alternativi che riguardano casistiche per cui principalmente la prenotazione non può andare a buon fine.
- Lo use case template b) riguarda l'eliminazione di una prenotazione effettuata da parte di un *User*, applicabile solo per prenotazioni effettuate per almeno il giorno successivo.
- Lo use case template c) riguarda il processo di registrazione di un account per un *User*, permettendo come post-condizione la possibilità di effettuare il login.
- Lo use case template d) riguarda l'aggiunta di un servizio fotografico da parte di un *fotografo*. Questo permette quindi (come post-condizione) di permettere agli utenti di effettuare prenotazioni anche per questo nuovo servizio inserito.
- Lo use case template e) infine riguarda l'eliminazione di una fascia oraria da parte di un fotografo. Questo provoca, come post-condizione, la non possibilità per i clienti di effettuare prenotazioni per qualsiasi servizio fotografico in quel preciso slot orario di quel dato fotografo (ad esempio per variazioni di orari del fotografo stesso).



## 2.3 Schema delle classi

Il diagramma delle classi riportato nella pagina successiva rappresenta le classi implementate e come queste interagiscono tra loro. Il diagramma mostra anche il design pattern Repository, spiegato dettagliatamente nella sezione 2.5. Le classi implementate, sono suddivise in package, ognuno dei quali con uno scopo specifico:

- **Controllers:** contiene tutti i controller relativi all'entità, all'autenticazione e al redirect delle dashboard basate sul ruolo dell'utente.
- **Services:** contiene tutte le classi addette alla gestione dei servizi erogati che si interfacceranno direttamente con il database. Permette di definire la business logic necessaria e gli strumenti per l'autenticazione dell'utente.
- **Domain:** contiene tutte le classi che rappresentano il *modello dei dati*. Queste classi rappresentano la modalità con la quale le sono stati rappresentati i dati e gli oggetti nell'applicativo.
- **Repositories:** contiene tutte le classi atte ad implementare il pattern architetturale Repository. Ognuna di esse estende JpaRepository, che fornisce operazioni di CRUD di default e ulteriori operazioni per la paginazione. Tramite questo package, possiamo aggiungere o togliere entità in qualunque momento e mantenere uguale tutto il resto del codice.
- **Configuration:** contiene tutte le classi per implementare la sicurezza web nell'applicativo, fornendo le autorizzazioni solo agli utenti corretti e ulteriori classi per configurare i mapper e il redirect alle dashboard dopo che si è eseguito il login.
- **Mappers:** contiene tutte le classi che permettono di mappare un DTO nell'entità associata e viceversa. Utilizzate per separare maggiormente l'*application layer* dal *persistence layer*.

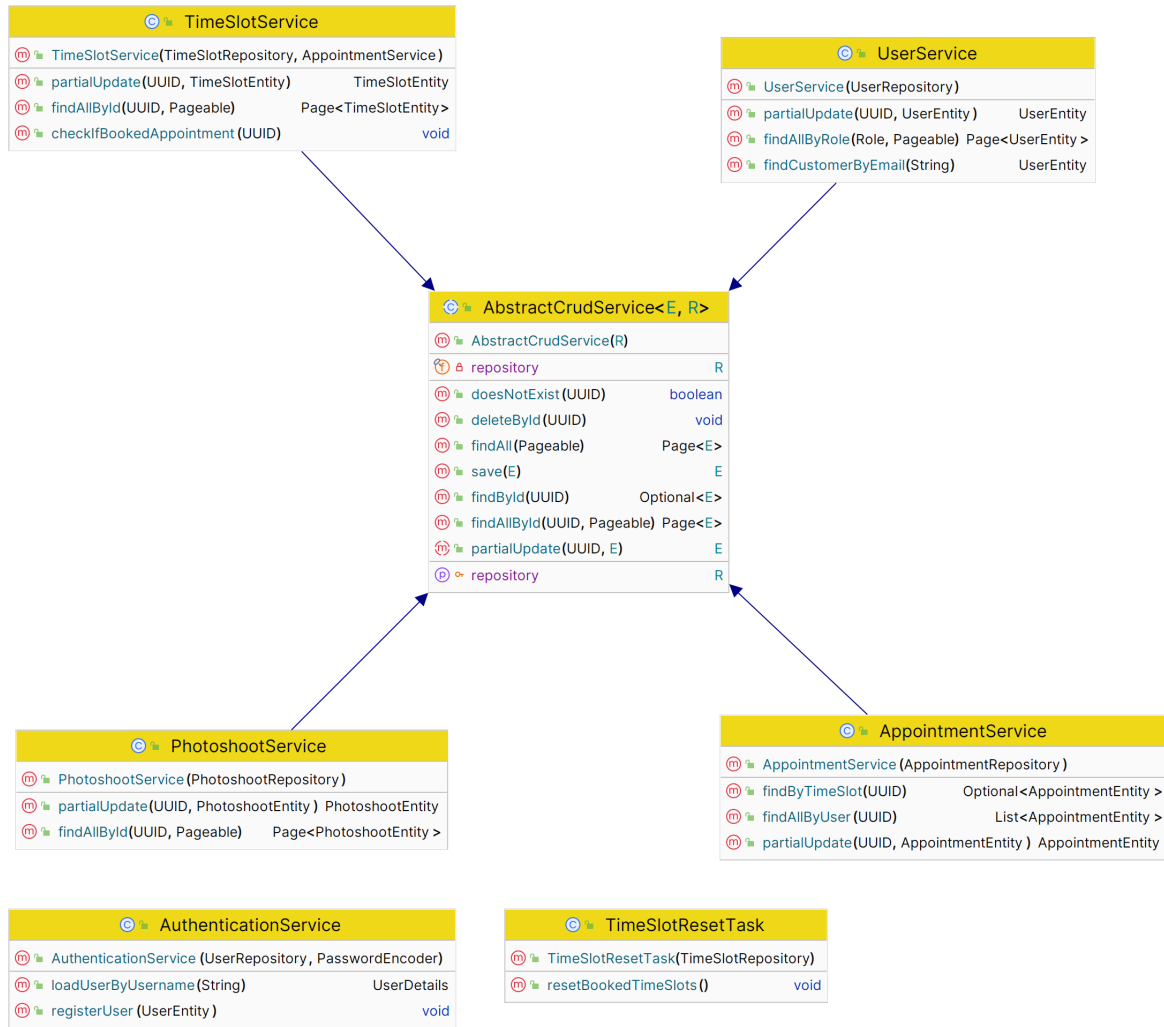


Figure 3: a) UML package services



Figure 4: b) UML package controllers



Figure 5: c) UML package repositories

## 2.4 Entity-Relationship Diagram

Nelle figure successive è presentata la struttura del database utilizzato nell'applicativo. Le *entità principali* coinvolte sono l'utente (**User**), l'appuntamento (**Appointment**), le fasce orarie (**Time Slot**) e un servizio fotografico (**Photoshoot**). Un appuntamento deve includere un riferimento al fotografo a cui è stato richiesto, un servizio fotografico scelto, un giorno e un orario in cui si svolgerà e l'utente che ha eseguito tale prenotazione. Questo è possibile tramite le relazioni uno a molti con tali identità, che si traducono in chiavi esterne. L'entità *User* ha un attributo *role* per permettere la differenziazione dei ruoli e di conseguenza i relativi permessi di accesso. Con questa specifica struttura gli orari disponibili per gli appuntamenti di tutti i fotografi finiscono nell'entità *Time Slot*, riferendosi a un dato fotografo tramite l'impiego di una chiave esterna (relazione uno a molti). Ho stabilito una relazione uno a molti con l'entità *User*, perché ogni fotografo ha la possibilità di gestirsi le proprie fasce orarie. Questo può portare a ridondanze nel caso di condivisione di fasce orarie tra fotografi, ma la gestione del database risulta più semplice e manutenibile. Finché il sistema è limitato a pochi fotografi, la struttura è gestibile; nel momento in cui i fotografi iscritti sono un buon numero, l'operazione migliore da fare sarebbe creare un'istanza del database per ogni singolo fotografo, in modo da separare i dati di ognuno, evitando ridondanze e appesantimenti. Con una struttura del genere, non avremmo più bisogno di utilizzare una chiave esterna per riferirsi al fotografo in questione. Inoltre, ogni fotografo può inserire ed eliminare i propri photoshoot. Se creo un'istanza per ogni fotografo, avrò una singola riga che rappresenta il fotografo in questione e tutti gli altri utenti saranno customer. Per tutte le entità ho utilizzato **UUID** come *Primary Key* perché risulta essere il più efficiente e sicuro nella generazione di valori univoci.

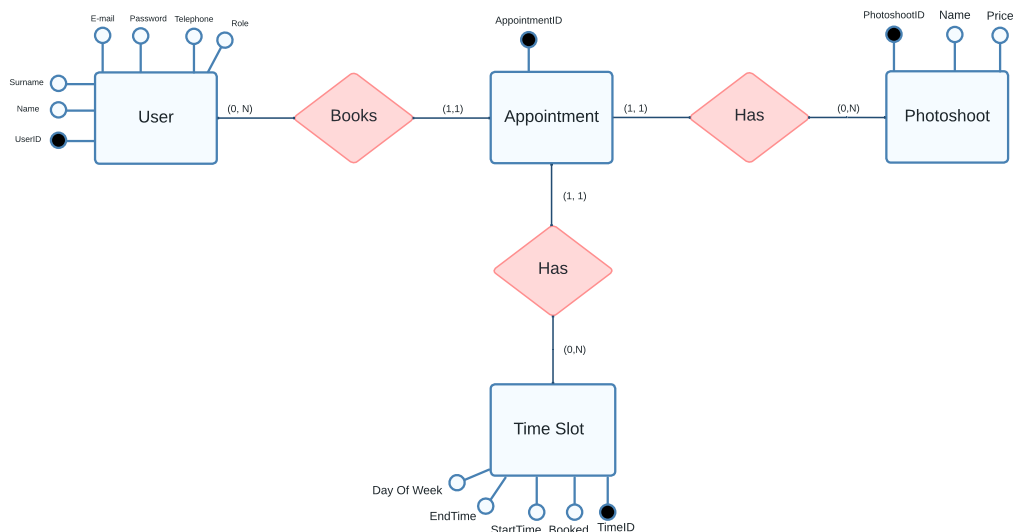


Figure 6: a) Diagramma Entità-Relazione

## 2.5 Pattern architetturale Repository

Il pattern **Repository** è un pattern molto utile per gestire il livello di *persistence*, molto comune quando si impiega Spring Boot. Il database utilizzato è MySQL. Questo ci permette di dividere in maniera netta l'accesso dei dati dalla logica di business. Per questo motivo, segue il *principio di singola responsabilità*, nascondendo i dettagli implementativi della gestione dei dati e potendo agire miratamente sul layer per effettuare modifiche, senza dover cambiare l'intero programma. Nel nostro caso, si creano delle entità all'interno di **com.SWEPhotoshootBookingEntities**, che avranno un tag **@Entity**, con i relativi campi definiti nello schema ER (figura 6) e con i vincoli di chiave stabiliti attraverso l'utilizzo dei tag **@ManyToOne** e **@JoinColumn**. Con tutte le entità create, poi si creano in **com.SWEPhotoshootBookingRepositories**, delle interfacce, una per entità. Grazie all'utilizzo di Spring Boot, ognuna di queste interfacce estende **JpaRepository<Entity, PKType>**, che fornisce tutte le operazioni di CRUD per la data entità fornita e anche operazioni di paginazione, utili in caso di grandi quantità di dati salvati. Grazie a JpaRepository possiamo creare, leggere, modificare e cancellare nuovi record dal database, senza dover programmare da zero tutte le operazioni tramite SQL. JpaRepository offre anche la possibilità di creare metodi personalizzati, che se rispettano una data semantica relativa al nome del metodo, sono in grado di generare l'SQL necessario in automatico. Nel caso di query più complesse, possiamo comunque crearle manualmente attraverso l'utilizzo di **HQL (Hibernate Query Language)**, simile alla sintassi SQL, ma più generica e sicura agli attacchi di tipo SQL Injection.

## 3 Implementazione

Il progetto è stato sviluppato in Java. Ho deciso di utilizzare JetBrains IntelliJ come strumento per la gestione delle librerie e l'organizzazione del progetto. Di seguito descriviamo i contenuti dei file di progetto.

### 3.1 Package Configuration

In questo package sono presenti tutte quelle classi per definire delle operazioni di configurazione utili a gestire la sicurezza, l'autenticazione degli utenti e a definire il PasswordEncoder, oltre che a definire i mapper. Per questo scopo, tutte le classi hanno il tag **@Configuration**.

#### 3.1.1 class SecurityConfig

Questa classe serve a definire gli accessi ai vari template html sulla base del ruolo

dell'utente. Ci sono dei template accessibili globalmente, come ad esempio il form di registrazione, l'homepage e il form di login. Tutti gli altri template che ho creato, come le dashboard in base al ruolo e tutti i relativi template di gestione possono essere utilizzati solo se il login risulta essere eseguito e solo se si ha il permesso basato sul ruolo dell'utente. Infatti le operazioni di gestione dei photoshoot e degli orari sono riservate ai fotografi, ma non agli utenti. Stesso discorso per il template di creazione dell'appuntamento, non accessibile al fotografo. Tramite il tag **@EnableWebSecurity**, si segnala a Spring che si vogliono abilitare operazioni per la sicurezza web. Tramite il metodo `securityFilterChain` si configurano le autorizzazioni HTTP che ci permette di definire quali template hanno accesso incondizionato. Inoltre, quando si esegue il login, in caso di esito positivo si richiama la classe `CustomAuthenticationSuccessHandler` tramite il metodo `successAuth()`, che imposterà l'utente come loggato generando un token relativo alla sessione.

Listing 1: Bean `securityFilterChain(HttpSecurity http)`

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
    http
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers("/", "/index", "/"
                .register", "register/user", "/login").
                permitAll()
            .anyRequest().authenticated()
        )
        .formLogin((form) -> form
            .loginProcessingUrl("/login")
            .successHandler(successAuth())
            .permitAll()
        )
        .logout(LogoutConfigurer::permitAll);

    return http.build();
}
```

### 3.1.2 class `CustomAuthenticationSuccessHandler`

Si occupa di definire l'autenticazione dell'utente una volta che ha inserito correttamente le proprie credenziali di accesso. Implementa l'interfaccia `AuthenticationSuccessHandler` che contiene il metodo `onAuthenticationSuccess`. Ho creato questa classe personalizzata per poter fare il redirect alla dashboard giusta, in base al ruolo dell'utente.

Listing 2: metodo onAuthenticationSuccess

```

@Override
public void onAuthenticationSuccess(HttpServletRequest
request, HttpServletResponse response, Authentication
authentication) throws IOException {
    handle(request, response, authentication);
    clearAuthenticationAttributes(request);
}
}

```

In questo metodo abbiamo 2 ulteriori metodi: `handle` e `clearAuthenticationAttributes`. Il primo permette di determinare il template dal richiamare sulla base del ruolo dell'utente che si è loggato. Infatti `handle` contiene a sua volta un metodo `determineTargetUrl(final Authentication authentication)`, che prende in input l'istanza della autenticazione dal quale ha accesso all'utente che ha inserito le credenziali.

Listing 3: metodo determineTargetUrl

```

protected String determineTargetUrl(final Authentication
authentication) {

    Map<String, String> roleTargetUrlMap = new HashMap<>();
    roleTargetUrlMap.put("ROLE_CUSTOMER", "/customer-
        dashboard");
    roleTargetUrlMap.put("ROLE_PHOTOGRAPHER", "/photographer-
        dashboard");

    final Collection<? extends GrantedAuthority> authorities
        = authentication.getAuthorities();
    for (final GrantedAuthority grantedAuthority :
        authorities) {
        String authorityName = grantedAuthority.getAuthority
            ();
        logger.info("authorityName: " + authorityName);
        if (roleTargetUrlMap.containsKey(authorityName)) {
            return roleTargetUrlMap.get(authorityName);
        }
    }
    throw new IllegalStateException();
}

```

Si crea una mappa con i ruoli e il template relativo associato; poi dall'istanza di `authentication` si accede al ruolo dell'utente che si è appena loggato (`authentication.getAuthorities()`); a questo punto, viene fatto il redirect al template corretto. Il secondo metodo, invece, permette di pulire qualsiasi stato di errore di autenticazione precedente alla sessione HTTP corrente.

### 3.1.3 class PasswordEncoderConfig

Questa classe istanzia un nuovo PasswordEncoder globale, che utilizza l'algoritmo BCrypt per evitare di salvare le password in chiaro sul database, violando le norme di sicurezza.

### 3.1.4 class MapperConfig

Si occupa di istanziare il ModelMapper dalla libreria di terze parti che ho utilizzato per mappare i DTO in entità e viceversa.

## 3.2 Package Domain

Questo package contiene le entità del database nel subpackage **entities** e i dto nel subpackage **dto** utili a trasferire i dati dal browser al database ai fini di differenziare i 3 livelli dell'applicativo (*persistence, services, controllers*).

Nel package entities risiedono tutte le entità che abbiamo definito nel diagramma ER. Prendiamo in esame l'entità Appointment. Grazie all'utilizzo di Spring Boot, possiamo identificarla come tale attraverso il tag **@Entity**. Per definire il nome della tabella associata a tale entità si richiama l'annotazione **@Table = "appointments"**. Inoltre, abbiamo **@Id** per identificare la *Primary Key* della tabella, in questo caso *appointmentID*. Per definire le relazioni tra le entità, si definisce come tipo, l'entità a cui si fa riferimento (che segue la stessa struttura della presente) e poi si definisce il tipo di relazione, che per Appointment risulta essere **@ManyToOne** per rappresentare una relazione di tipo uno a molti. Per definire le chiavi esterne a cui si fa riferimento tramite tale relazione, abbiamo l'annotazione **@JoinColumn(name = "columnName")**, che automaticamente riferenzia la *Primary Key* dell'entità definita nel tipo dell'attributo. Inoltre, definiamo per sicurezza un metodo **generateUUID** con il tag **@PrePersist**, per assicurarsi di avere una primary key impostata prima del salvataggio dei dati sul database, per evitare errori. Per concludere, attraverso l'utilizzo di **Lombok** e delle annotazioni ad esso associate, come ad esempio **@Data**, possiamo generare i getter e i setter in automatico per ogni attributo dell'entità; invece, attraverso **@Builder**, si può creare un oggetto complesso di tipo AppointmentEntity automaticamente richiamando **builder()**, sull'oggetto, definendo tutti i campi e poi costruirlo tramite **build()** (utile per il testing).



### 3.2.1 class AppointmentEntity

Listing 4: Classe AppointmentEntity

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Entity
@Table(name = "appointments")
public class AppointmentEntity {

    @Id
    private UUID appointmentID;

    @ManyToOne
    @JoinColumn(name = "customerID")
    private UserEntity customer;

    @ManyToOne
    @JoinColumn(name = "photoshootID")
    private PhotoshootEntity photoshootEntity;

    @ManyToOne
    @JoinColumn(name = "timeslotID")
    private TimeSlotEntity timeslot;

    @ManyToOne
    @JoinColumn(name = "photographerID")
    private UserEntity photographer;

    @PrePersist
    public void generateUUID() {
        if (appointmentID == null) {
            appointmentID = UUID.randomUUID();
        }
    }
}
```

Listing 5: Utilizzo di builder() e build()

```
UserEntity.builder()  
.userID(UUID.randomUUID())  
.name("Jack")  
.surname("Sparrow")  
.email("jack.sparrow@example.com")  
.password("12345678")  
.telephone("+393980878782").build();
```

Lo stessa struttura viene seguita per `PhotoshootEntity`, `TimeSlotEntity` e `UserEntity`. Nel caso di quest'ultima ho creato una classe *Role* con enum per avere come unici ruoli **CUSTOMER** e **PHOTOGRAPHER**.

Listing 6: Attributo role in UserEntity

```
@Enumerated(EnumType.STRING)  
private Role role;
```

Listing 7: Classe Role

```
public enum Role {  
    CUSTOMER,  
    PHOTOGRAPHER  
}
```

In `TimeSlotEntity`, per definire gli slot orari ho utilizzato la classe di base `LocalTime`.

Inoltre, questo package contiene tutte le classi DTO relative alle entità precedentemente definite. Un **DTO (Data Transfer Object)** è un modello di dati utilizzato per trasferire dati tra diversi strati di un'applicazione software. Un DTO rappresenta un oggetto semplice contenente solo dati e non contiene alcuna logica o comportamento associato. Lo scopo principale di un DTO è quello di semplificare il trasferimento dei dati tra i diversi componenti del sistema, come in questo caso dal frontend al backend. È utile ai fini della divisione del livello di *application* dal livello di *persistence*, evitando di usare direttamente le entità nel passaggio dei dati. Sono una copia delle classi entità, ma senza nessun tag relativo al database e nel caso di chiavi esterne, invece di usare l'intera entità, consideriamo solo l'ID, per alleggerire il trasferimento. Inoltre ho creato un'interfaccia `IdentifiableDto` per avere un metodo `setId` comune a tutti i dto, utile per settare l'ID in modo generico all'interno dei controller (che di base estendono una classe astratta).

### 3.2.2 class PhotoshootDto

Listing 8: Esempio DTO: classe PhotoshootDto

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PhotoshootDto implements IdentifiableDto {

    private UUID photoshootID;

    private String name;

    private Integer price;

    private UUID photographer;

    @Override
    public void setId(UUID id) {
        photoshootID = id;
    }
}
```

### 3.2.3 class LoginDto

Ho creato anche un LoginDto, utile per trasferire i dati di email e password durante il processo di login sul browser, evitando di trasferire altri dati dell'utente che risulterebbero superflui e ridondanti per lo scopo.

Listing 9: LoginDto

```
@Data
public class LoginDto {

    private String email;
    private String password;

}
```

### 3.3 Package Repositories

All'interno di questo package sono presenti tutte le classi che permettono alle entità definite di eseguire operazioni di CRUD sul database (creazione, lettura di uno o molti, aggiornamento parziale/totale e cancellazione). Ognuna di queste classi è un'interfaccia con un tag **@Repository**, che estende **JpaRepository<Entity, PKType>**, la quale fornisce in automatico tutti i metodi di CRUD, senza doverli implementare manualmente. Ci fornisce anche metodi sulla paginazione.

#### 3.3.1 class TimeSlotRepository

Listing 10: Esempio Repository: TimeSlotRepository

```
@Repository
public interface TimeSlotRepository extends JpaRepository<
    TimeSlotEntity, UUID> {

    Page<TimeSlotEntity> findAllByPhotographer_userID(UUID
        photographerId, Pageable pageable);

    Page<TimeSlotEntity> findByDayOfWeek(int dayOfWeek, Pageable
        pageable);
}
```

Come possiamo vedere, semplicemente estendendo la classe si ha accesso a tutti i metodi CRUD di default. I metodi aggiunti sono metodi che eseguono query personalizzate e in questo caso, avendo definito la semantica del nome del metodo in base ai campi dell'entità **TimeSlot**, Spring Boot è in grado di ritornare in automatico i record dal database sulla base del nome, senza dover implementare il metodo da nessuna altra parte. Attraverso **Page** e **Pageable**, si possono ritornare pagine di record del database di una certa lunghezza fissa, evitando di usare liste, che potrebbero risultare pesanti nel caso in cui il metodo ritorni molti dati. Questa struttura è la solita per tutte le altre entità.

### 3.4 Package Mappers

In questo package sono contenute tutte le classi che permettono la traduzione da un oggetto di tipo DTO che trasporta i dati dal browser a un oggetto di tipo Entity per poter permettere le effettive operazioni sul database. Per fare ciò ho utilizzato una libreria di terze parti: **ModelMapper**.

Listing 11: Dependency ModelMapper

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>3.0.0</version>
</dependency>
```

### 3.4.1 interface Mapper

Ho creato un'interfaccia di base che permette di eseguire 2 operazioni: **mapTo** e **mapFrom**, per permettere la traduzione da/a oggetti DTO/Entity.

Listing 12: Interfaccia ModelMapper

```
public interface Mapper<A, B> {
    B mapTo(A a);
    A mapFrom(B b);
}
```

### 3.4.2 Photoshoot Mapper

Ogni mapper concreto implementa i 2 metodi dell'interfaccia. Dato che nei DTO, per le chiavi esterne considero solo l'UUID e non l'intera entità, ho adattato i mapper che hanno tali chiavi esterne per recuperare l'entità relativa allo UUID passato. Tutte le classi del package hanno il tag **@Component**, per permettere a Spring Boot la dependency injection dove necessario.

Listing 13: Photoshoot Mapper

```
@Override
public PhotoshootDto mapTo(PhotoshootEntity photoshootEntity)
{
    return modelMapper.map(photoshootEntity, PhotoshootDto.
        class);
}

@Override
public PhotoshootEntity mapFrom(PhotoshootDto photoshootDto)
{
    PhotoshootEntity photoshootEntity = modelMapper.map(
        photoshootDto, PhotoshootEntity.class);
}
```

```

        UserEntity photographer = userService.findById(
            photoshootDto.getPhotographer())
            .orElseThrow(() -> new RuntimeException("User not
                found"));

        photoshootEntity.setPhotographer(photographer);

        return photoshootEntity;
    }
}

```

## 3.5 Package Services

In questo package sono contenute tutte le classi che vanno a formare il livello di *service*, il quale definisce un livello intermedio tra l'*application layer* e il *persistence layer*, utile a definire tutta la business logic di cui necessitiamo per eseguire operazioni sul database e fornire accesso dei metodi ai controller senza che quest'ultimi si debbano interfacciare direttamente con i metodi CRUD del persistence layer, garantendo una divisione coerente dei livelli. Inoltre, definiamo ulteriori classi per l'autenticazione dell'utente e per il reset delle fasce orarie una volta che gli appuntamenti si sono svolti.

### 3.5.1 class AbstractCrudService

Il mio approccio è stato quello di definire un servizio generico, che rappresenta una classe astratta con una serie di operazioni comuni a tutti i servizi delle entità.

Listing 14: Alcuni metodi di AbstractCrudService

```

@Transactional
public E save(E entity) {
    return repository.save(entity);
}

@Transactional(readonly = true)
public Optional<E> findById(UUID id) {
    return repository.findById(id);
}

@Transactional(readonly = true)
public Page<E> findAll(Pageable pageable) {
    return repository.findAll(pageable);
}

```

```

    @Transactional
    public abstract E partialUpdate(UUID id, E entity);

    @Transactional
    public void deleteById(UUID id) {
        repository.deleteById(id);
    }
}

```

Sostanzialmente, tutti questi metodi fanno riferimento ai metodi CRUD. Grazie a questo livello intermedio, nel caso in cui si decidesse di cambiare la struttura o il tipo di database utilizzato, l'application layer non vedrebbe alcuna differenza, perché fa sempre riferimento a questo livello e mai a quello del database. In questo modo possiamo agire su ogni componente del sistema in modo isolato e mantenere la solita interfaccia, senza dover ristrutturare il codice nella sua interezza, rispettando il *principio di singola responsabilità*. Inoltre, permette una riusabilità di tale codice in più zone del programma, rispettando il **DRY (Don't Repeat Yourself)** e una maggiore facilità nel testing. Notiamo come ho inserito il tag **@Transactional** e **@Transactional (readOnly = true)**, che implementa il concetto di transazione di un database. In questo modo, ci assicuriamo che tutte le operazioni di scrittura/lettura preservino uno stato di coerenza dei dati nel caso di incidenti di qualunque natura che potrebbero portare ad eseguire solo parte delle istruzioni del metodo, rischiando un'inconsistenza dei dati. Il metodo **partialUpdate** l'ho definito puramente astratto, in modo che ogni entità ne facesse l'override e aggiornasse i propri campi, che ovviamente, sono diversi tra le varie tabelle.

### 3.5.2 class UserService

Estende AbstractCrudService<UserEntity, UserRepository>

Listing 15: metodi specifici di UserService

```

@Override
public UserEntity partialUpdate(UUID id, UserEntity entity) {
    entity.setUserID(id);
    return getRepository().findById(id).map(existingCustomer
-> {
        // Check if any of the fields are null and then
        setting them
        Optional.ofNullable(entity.getName()).ifPresent(
            existingCustomer::setName);
        Optional.ofNullable(entity.getSurname()).ifPresent(
            existingCustomer::setSurname);
        Optional.ofNullable(entity.getEmail()).ifPresent(
            existingCustomer::setEmail);
    });
}

```

```

        Optional.ofNullable(entity.getPassword()).ifPresent(
            existingCustomer::setPassword);
        Optional.ofNullable(entity.getTelephone()).ifPresent(
            existingCustomer::setTelephone);
        return getRepository().save(existingCustomer);
    }).orElseThrow(() -> new RuntimeException("User not
        exists"));
}

public UserEntity findCustomerByEmail(String email) {
    return getRepository().findByEmail(email).orElseThrow();
}

public Page<UserEntity> findAllByRole(Role role, Pageable
    pageable) {
    return getRepository().findAllByRole(role, pageable);
}

```

Per definire un service concreto, si utilizza il tag **@Service**, che segnala a Spring Boot che la classe è un **@Bean**, utile per il concetto di **Dependency Injection**. Infatti tramite questo tag, Spring Boot è in grado di iniettare questi servizi nel livello dell'application layer all'interno dei controller, grazie al tag **@Autowired**. Lo stesso concetto vale per le repository, che come in questo caso, viene iniettata nel costruttore del service concreto, permettendo l'accesso ai relativi metodi.

### 3.5.3 class AuthenticationService

Ho definito anche questa classe, che permette di far registrare/loggare l'utente all'applicativo.

Listing 16: metodi registerUser e loadUserByUsername

```

public void registerUser(UserEntity user) {
    logger.info("registerCustomer called with email: {}",
        user.getEmail());
    String encodedPassword = passwordEncoder.encode(user.
        getPassword()); // Encoding the user's password
    user.setPassword(encodedPassword);
    userRepository.save(user); // Saving the user to the
        database
    logger.info("Customer registration successful for email:
        {}", user.getEmail());
}

@Override
public UserDetails loadUserByUsername(String email) {

```



```

        logger.info("loginUser called with email: {}", email);
        UserEntity user = userRepository.findByEmail(email).
            orElseThrow(() -> new RuntimeException("User not found
            "));; // Check if the user is present
        return User.withUsername(user.getEmail())
            .password(user.getPassword())
            .roles(user.getRole().name()) // Building the
            entire user if password matches
            .build();
    }
}

```

Questa classe fornisce un metodo **registerUser**, che riceve una `UserEntity`, la quale contiene tutti i dati che l'utente ha inserito in un form sul browser, i cui dati vengono passati al controller che a sua volta richiama questo servizio. Per una questione di sicurezza, ho utilizzato un **PasswordEncoder**, che utilizza l'algoritmo **BCrypt**, un algoritmo di hashing crittografico progettato per archiviare in modo sicuro le password. È progettato per essere resiliente contro attacchi di forza bruta e di dizionario. Sostanzialmente, per generare l'hash di una password, viene generato casualmente un "salt". Il salt è un valore casuale e unico che viene concatenato alla password prima dell'hashing. Questo salt rende ogni hash unico, anche se due password sono identiche. Abbiamo anche il metodo **loadByUsername**, dell'interfaccia **UserDetailsService**, che si occupa del login dell'utente tramite l'email che quest'ultimo fornisce. Si richiama il metodo **findByEmail** e se l'utente è presente, Spring Boot lo autentica generando un token, altrimenti notifica una non presenza dell'utente o un inserimento della password sbagliato.

### 3.5.4 class TimeSlotResetTask

Per come ho definito gli slot orari, ogni utente può prenotare un servizio fotografico al massimo entro la settimana successiva. Nel momento in cui si effettua una prenotazione con un certo fotografo, la fascia oraria viene considerata prenotata e nessun utente può riprenotarla fino a che la prenotazione non risulta essere stata completata. Siccome ho utilizzato l'attributo **dayOfWeek**, che rappresenta un numero intero relativo al giorno considerato, ogni qual volta questo giorno risulta essere passato rispetto al giorno corrente, lo si rende nuovamente disponibile settando il campo **booked** a **false**. Con questa classe, ogni giorno a 00:00 (tramite il tag **@Scheduled**) viene controllato che ci siano o meno prenotazioni eseguite il giorno precedente e in caso positivo si rendono nuovamente disponibili, permettendo la prenotazione di quel giorno nella settimana successiva.

Listing 17: TimeSlotResetTask

```

@Scheduled(cron = "0 0 0 * * ?") // Task executed everyday at
    midnight
public void resetBookedTimeSlots() {

```

```

    int yesterdayDayOfWeek = LocalDate.now().minusDays(1).
        getDayOfWeek().getValue(); // Taking yesterday
    Pageable pageable = PageRequest.of(0, 1000);
    Page<TimeSlotEntity> timeSlots = timeSlotRepository.
        findByDayOfWeek(yesterdayDayOfWeek, pageable); // Find
        all the timeslots scheduled yesterday
    for (TimeSlotEntity timeSlot : timeSlots) {
        timeSlot.setBooked(false); // All the timeslots can
        be rebooked again for the next week
        timeSlotRepository.save(timeSlot);
    }
}

```

### 3.6 Package Controllers

In questo package implementiamo l'*application layer*. I controller sono uno strumento utile per ritornare le viste che l'utente andrà ad utilizzare e per permettere l'effettivo passaggio dei dati dal browser all'applicativo attraverso l'utilizzo dei DTO e di invio dei dati sottoforma di JSON. L'insieme di questi metodi formano una **RESTful API (Representational State Transfer API)**, un meccanismo che consente a un'applicazione o servizio di accedere a una risorsa all'interno di un'altra applicazione o servizio. L'applicazione o il servizio che accede alle risorse è il client e l'applicazione o il servizio che contiene la risorsa è il server. Le RESTful API permettono di eseguire operazioni di CRUD attraverso delle richieste che possono essere:

- **POST:** Il metodo POST viene utilizzato per inviare dati al server per elaborazione o aggiornamento. Questo metodo viene spesso utilizzato per creare nuove risorse sul server o per inviare dati a un endpoint che elaborerà l'input e restituirà una risposta. Le richieste POST possono modificare lo stato del server e non sono idempotenti, il che significa che inviare la stessa richiesta più volte può avere effetti diversi.
- **GET:** Il metodo GET viene utilizzato per richiedere dati da una risorsa specificata. Quando un client invia una richiesta GET a un server, il server restituisce la rappresentazione della risorsa richiesta. Le richieste GET sono generalmente considerate "sicure" e "idempotenti", il che significa che non modificano lo stato del server e possono essere eseguite più volte senza effetti collaterali.
- **PUT:** Il metodo PUT viene utilizzato per aggiornare o sostituire completamente una risorsa esistente sul server con i dati forniti nella richiesta. In altre parole, PUT viene utilizzato per creare o sovrascrivere completamente la rappresentazione di una risorsa con una nuova versione. Le richieste PUT dovrebbero essere idempotenti, il che significa che inviare la stessa richiesta più volte dovrebbe produrre lo stesso risultato.

- **PATCH:** Il metodo PATCH viene utilizzato per applicare modifiche parziali a una risorsa esistente. A differenza di PUT, che sostituisce completamente la risorsa, PATCH viene utilizzato per applicare modifiche specifiche senza sostituire l'intera risorsa. Questo metodo è utile quando si desidera aggiornare solo determinate parti di una risorsa senza dover inviare tutti i dati.
- **DELETE:** Il metodo DELETE viene utilizzato per rimuovere una risorsa specificata dal server. Quando un client invia una richiesta DELETE a un server, la risorsa corrispondente viene eliminata. Le richieste DELETE dovrebbero essere idempotenti, il che significa che inviare la stessa richiesta più volte non dovrebbe avere effetti diversi.

### 3.6.1 class AbstractCrudController

Come per i services, ho creato una classe astratta con le operazioni di CRUD comuni a tutte le entità, in modo da evitare codice duplicato.

Listing 18: Metodi di AbstractCrudController

```
@PostMapping
public ResponseEntity<D> create(@RequestBody D dto) {
    E entity = mapper.mapFrom(dto);
    E savedEntity = service.save(entity);
    return new ResponseEntity<>(mapper.mapTo(savedEntity),
        HttpStatus.CREATED);
}

@GetMapping
public Page<D> listRecords(Pageable pageable) {
    Page<E> listAll = service.findAll(pageable);
    return listAll.map(mapper::mapTo);
}

@GetMapping("/{id}")
public ResponseEntity<D> getRecord(@PathVariable("id") UUID
id) {
    Optional<E> foundEntity = service.findById(id);
    return foundEntity.map(E -> {
        D dto = mapper.mapTo(E);
        return new ResponseEntity<>(dto, HttpStatus.OK);
    }).orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}

@PutMapping("/{id}")
```

```

    public ResponseEntity<D> fullUpdateRecord(@PathVariable("id")
    UUID id, @RequestBody D dto) {
        if (service.doesNotExist(id)) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        dto.setId(id);
        E entity = mapper.mapFrom(dto);
        E savedEntity = service.save(entity);
        return new ResponseEntity<>(mapper.mapTo(savedEntity),
            HttpStatus.OK);
    }

    @PatchMapping("/{id}")
    public ResponseEntity<D> partialUpdateRecord(@PathVariable("
id") UUID id, @RequestBody D dto) {
        if (service.doesNotExist(id)) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }

        E entity = mapper.mapFrom(dto);
        E updatedEntity = service.partialUpdate(id, entity);
        return new ResponseEntity<>(mapper.mapTo(updatedEntity),
            HttpStatus.OK);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteRecord(@PathVariable("id")
    UUID id) {
        service.deleteById(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}

```

Quello che succede è che ognuno di questi metodi può prendere sia l'UUID che il DTO, in base al tipo di operazioni volute. I dati contenuti nel DTO provengono dagli effettivi input dell'utente nelle pagine html che vanno a formare il frontend. Dato che i DTO non sono le reali entità, vado ad utilizzare il relativo mapper associato al DTO passato per poterne permettere la conversione e dunque poi eseguire le operazioni sul database. Ogni metodo ritorna una **ResponseEntity**, che rappresenta un insieme di codici di stato utili a definire l'esito di ognuno dei metodi. I codici di stato utilizzati a questo scopo sono i seguenti:

- **200 OK**: risposta standard per le richieste HTTP andate a buon fine. Utile per operazioni di lettura.
- **201 Created**: quando viene creato un nuovo record sul database con successo. Utile per operazioni di creazione.

- **204 No Content** : Il server ha processato con successo la richiesta e non restituirà nessun contenuto. Utile per operazioni di cancellazione.
- **404 Not Found**: La risorsa richiesta non è stata trovata ma in futuro potrebbe essere disponibile.

L'implementazione di questa RESTful API è stata testata tramite **Postman Agent**.

### 3.6.2 class TimeSlotController

Prendiamo in esempio un controller concreto. Esso estende `AbstractController<TimeSlotEntity, TimeSlotDto, TimeSlotRepository>`. Oltre a tutte le operazioni ereditate da `AbstractCrudController`, ho creato metodi specifici per poter permettere la visualizzazione di tutti gli orari di un determinato fotografo e per eliminare degli appuntamenti che contengono una fascia oraria che viene eliminata dal fotografo.

Listing 19: Metodi specifici di TimeSlotController

```
@GetMapping("/photographer/{photographerId}")
public Page<TimeSlotDto> listTimeSlotsByPhotographer(@
PathVariable UUID photographerId, Pageable pageable) {
    logger.info("Listing time slots for photographer with id:
        " + photographerId);
    Page<TimeSlotEntity> timeSlots = timeSlotService.
        findAllById(photographerId, pageable);
    return timeSlots.map(mapper::mapTo);
}

@Override
public ResponseEntity<Void> deleteRecord(@PathVariable("id")
UUID id) {
    timeSlotService.checkIfBookedAppointment(id); // Check if
        there are any appointments that refers to the
        timeslot id that the photographer intend to delete. If
        true, the appointment has been cancelled
    timeSlotService.deleteById(id);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
}
```

Per definire la classe come parte della RESTful API si utilizza il tag **@RestController** e anche **@RequestMapping**, che identifica il percorso da richiamare per eseguire le operazioni correttamente.

### 3.6.3 class AuthController

Grazie a questa classe, si richiamano i template di homepage, register e di login nel frontend.

Listing 20: Metodi di AuthController

```
//Showing homepage
@GetMapping("/index")
public String home() {
}

//Showing registration template
@GetMapping("/register")
public String showRegistrationForm(Model model) {
    model.addAttribute("user", new UserDto());
    return "/register";
}

//Showing login template
@GetMapping("/login")
public String showLoginForm(Model model) {
    model.addAttribute("login", new LoginDto());
    return "/login";
}

@PostMapping("/register/user")
public String registerUser(@Valid @ModelAttribute("user")
UserDto userDto,
                           BindingResult result,
                           Model model) {
    UserEntity existingCustomer = userService.
        findCustomerByEmail(userMapper.mapFrom(userDto).
            getEmail());

    if (existingCustomer != null && existingCustomer.getEmail()
        != null && !existingCustomer.getEmail().isEmpty())
    {
        result.rejectValue("email", null,
            "There is already an account registered with
            the same email");
    }

    if (result.hasErrors()) {
        model.addAttribute("user", userDto);
        return "/register";
    }
}
```

```

        log.info("Registering user with email: {}", userDto.getEmail());
        authenticationService.registerUser(userMapper.mapFrom(
            userDto));
        log.info("User registered successfully");
        return "redirect:/register?success";
    }
}

```

Quando il browser fa richiesta di una pagina html, lo fa tramite `@GetMapping("/template-name")`, che ritorna la pagina in modo dinamico. Questo è possibile grazie all'utilizzo di **Thymeleaf**. Il metodo `registerUser` prende i dati sotto forma di JSON del form compilato dall'utente e lo ricerca tramite email. Se l'utente è già presente si rifiuta la richiesta e non si crea nuovamente l'utente, notificando l'errore e venendo reindirizzati nuovamente alla pagina di registrazione; se l'utente non è presente si richiama il metodo `registerUser` di `AuthenticationService`, che farà l'encoding della password e il push sul database dell'utente.

### 3.6.4 class CustomerDashboardController

In base all'utente che si logga possiamo avere una dashboard per il cliente o una dashboard per il fotografo, che ha privilegi di admin. Nel caso del Customer, esso può creare un nuovo appuntamento oppure visualizzarli con la possibilità di modificarli o eliminarli.

Listing 21: Metodi di CustomerDashboardController

```

    @GetMapping("/customer-dashboard")
    public String showUserDashboard(Model model, Principal
principal) {
        UserEntity customer = userService.findCustomerByEmail(
            principal.getName());
        model.addAttribute("customerName", customer.getName());
        return "customer-dashboard";
    }

    @GetMapping("/book-appointment")
    public String showBookAppointmentPage(Model model, Principal
principal) {
        AppointmentDto appointment = new AppointmentDto();
        UserEntity customer = userService.findCustomerByEmail(
            principal.getName());
        appointment.setCustomer(customer.getUserID()); // Setting
            the user that is going to book the appointment
        model.addAttribute("appointment", appointment);
    }
}

```

```

        Pageable firstPageWithTenElements = PageRequest.of(0, 10)
        ;
        model.addAttribute("photographers", userService.
            findAllByRole(Role.PHOTOGRAPHER,
                firstPageWithTenElements)); // Sending to the frontend
            all the available photographers
        return "book-appointment";
    }

    @GetMapping("/view-appointments")
    public String showViewAppointmentsPage(Model model, Principal
principal) {
        UserEntity customer = userService.findCustomerByEmail(
            principal.getName());
        model.addAttribute("appointments", appointmentService.
            findAllByUser(customer.getUserID()));
        return "view-appointments";
    }
}

```

Quando si richiama il template di creazione di un appuntamento si istanzia un nuovo `AppointmentDto`, che conterrà tutti i dati che l'utente sceglierà e che poi verranno passati al metodo `create` di `AbstractCrudController`, per permettere l'effettiva creazione. Inoltre, dato che abbiamo accesso all'utente loggato, setto nel backend l'UUID dell'utente che sta eseguendo la prenotazione, in modo da passarlo durante la `create`, evitando che sia `null`. Inoltre la visualizzazione di tutti i fotografi sfrutta la paginazione tramite `Pageable` e `PageRequest`.

### 3.6.5 class `PhotographerDashboardController`

In questo controller, definiamo la visualizzazione dei template relativi alla gestione dei photoshoot, dei timeslots e alla visualizzazione delle prenotazioni dei clienti. La struttura è la medesima di quella del `Customer`.

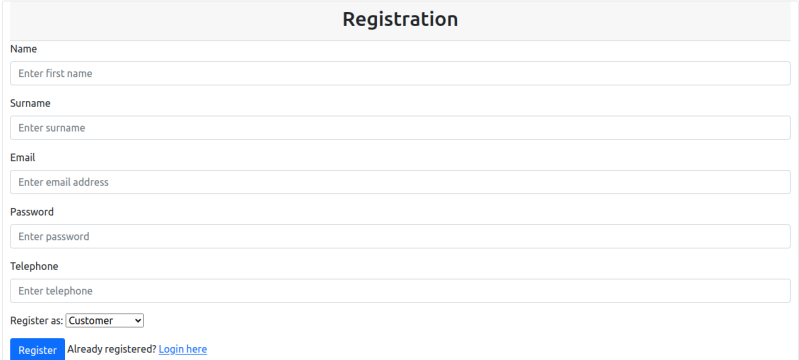
## 3.7 Package templates

Contiene tutti i template HTML utilizzati nell'applicativo. Per caricare le pagine in modo dinamico ho utilizzato **Thymeleaf**. Per il passaggio dei dati in modo sicuro ho utilizzato un **CSRF (Cross-Site Request Forgery) Token**, una misura di sicurezza utilizzata per proteggere le applicazioni web da attacchi CSRF. Un attacco CSRF si verifica quando un aggressore riesce a far eseguire azioni non autorizzate da parte di un utente autenticato all'interno di un'applicazione web, sfruttando la sua sessione autenticata. Il CSRF token



viene utilizzato per mitigare questo tipo di attacco introducendo un valore casuale e univoco all'interno di ogni richiesta inviata dall'utente autenticato. Questo token l'ho incluso come un campo nascosto nel codice.

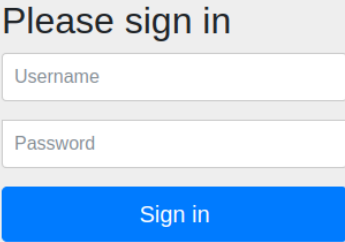
### 3.7.1 register.html



The image shows a web page for a registration system. At the top, there is a dark header bar with the text "Registration and Login System" and a "Login" link. Below the header, the main content area contains a "Registration" form. The form has a title "Registration" and several input fields: "Name" (with a sub-label "Enter first name"), "Surname" (with a sub-label "Enter surname"), "Email" (with a sub-label "Enter email address"), "Password" (with a sub-label "Enter password"), and "Telephone" (with a sub-label "Enter telephone"). There is also a dropdown menu for "Register as:" with "Customer" selected. At the bottom of the form, there is a blue "Register" button, a link "Already registered? Login here", and a "Login" link.

Figure 7: Register template

### 3.7.2 login.html



The image shows a web page for a login system. The background is a light gray. In the center, there is a "Please sign in" form. The form has a title "Please sign in" and two input fields: "Username" and "Password". Below the input fields, there is a blue "Sign in" button.

Figure 8: Login template

### 3.7.3 photographer-dashboard.html

Welcome, Alessandro!

Manage Photoshoots

Create, update or delete photoshoots.

[Go to Photoshoots](#)

Manage Time Slots

Manage your available time slots.

[Go to Time Slots](#)

View Clients

View your clients and their bookings.

[Go to Clients](#)

Figure 9: Photographer Dashboard template

### 3.7.4 manage-photoshoots.html

Photoshoots Management			Dashboard	Logout
Photoshoots				
Name	Price			
Ritratto	70	<a href="#">+</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
Matrimonio	500	<a href="#">+</a>	<a href="#">Edit</a>	<a href="#">Delete</a>

Figure 10: Manage Photoshoots template

Per poter inviare i dati ai REST Controller, tutti i campi devono essere convertiti in JSON. Inoltre si deve definire il tipo di richiesta in base all'operazione che stiamo svolgendo. Per questo scopo, ho usato **AJAX**, che permette di eseguire uno **stringify** dei campi, il redirect al controller corretto e il tipo di richiesta che si vuole richiamare.

Listing 22: Utilizzo di AJAX

```
$.ajax({
  url: '/photoshoots/' + photoshootId,
  type: 'PATCH',
  contentType: 'application/json',
  data: JSON.stringify({
    name: name,
    price: price,
```

```

        photographer: photographerID
    } ),
    beforeSend: function (xhr) {
        xhr.setRequestHeader(header, token);
    },
    success: function (response) {
        $('#exampleModal').modal('hide');
        window.location.href = "/manage-photoshoots";
    },
    error: function (error) {
        console.log(error);
    }
});

```

### 3.7.5 book-appointment.html

Figure 11: Book Appointment template

In questa pagina, si sceglie il fotografo desiderato e grazie ad AJAX, mi ricavo tutti gli slot orari disponibili del dato fotografo, insieme a tutti i photoshoot da lui inseriti. Quando la prenotazione viene effettuata, si imposta il campo `booked` del time slot scelto a `true`, in modo che nessun altro utente possa prenotare quella fascia oraria. Tornerà nuovamente disponibile quando la prenotazione sarà stata effettuata, grazie al servizio **TimeSlotResetTask**, definito precedentemente.

Listing 23: Come vengono recuperati i dati sulla base del fotografo scelto dall'utente

```

$.ajax({
    url: '/timeslots/photographer/' + photographerId,
    type: 'GET',
    beforeSend: function (xhr) {
        xhr.setRequestHeader(header, token);
    },
    success: function (data) {
        var timeSlotSelect = $('#timeSlotSelect');

```

```

timeSlotSelect.empty();
$.each(data.content, function (index, timeSlot) {
    if (timeSlot === undefined || timeSlot.booked === true) {
        return true;
        // If time slot is booked or undefined don't show it
    }
    var diff = timeSlot.dayOfWeek - today.getDay();
    // If already passed, add 7 days
    if (diff < 0) {
        diff += 7;
    }
    var futureDate = new Date();
    // Calculating the next date base on the day of the timeslot
    futureDate.setDate(today.getDate() + diff);
    var dayOfWeek = daysOfWeek[futureDate.getDay()];
    console.log(timeSlot.startTime + '-' + timeSlot.endTime);
    var optionText = '(' + dayOfWeek + ')';
    optionText += timeSlot.startTime + '-' + timeSlot.endTime;
    timeSlotSelect.append($('

```

## 4 Test

Con lo scopo di controllare la correttezza delle funzionalità implementate, sono stati realizzati dei test **unitari** e di **integrazione (integration tests)**, con l'aiuto della libreria java JUnit 5 . In particolare, sono state verificate tutte le operazioni di CRUD che vengono fornite da JpaRepository, tutte le operazioni definite nei RESTful Controllers per verificare che i codici di stato ritornati fossero corretti in base all'operazione svolta, tutti i metodi dei servizi (test unitari) e tutti i metodi legati all'autenticazione dell'utente. Per eseguire i testing ho utilizzato il **database H2**, progettato per essere leggero, veloce e facilmente integrabile, specialmente per il testing.

### 4.1 class UserEntityIntegrationTests

Questa classe permette di verificare che tutte le operazioni di CRUD fornite da JpaRepository siano funzionanti nel contesto di UserEntity.

```

@Test
public void testThatUserCanBeCreated(){
    UserEntity userEntity = TestDataUtil.createTestUserA();
    underTest.save(userEntity);

    Optional<UserEntity> result = underTest.findById(
        userEntity.getUserID());
    assertThat(result).isPresent();
    assertThat(result.get()).isEqualTo(userEntity);
}

```

Si genera una nuova UserEntity tramite createTestUserA() e la si salva sul database. Dopodiché la si ricerca tramite il metodo findById(userEntity.getUserID()) e si asserisce che sia uguale per verificarne la veridicità.

Listing 25: Testing metodo di aggiornamento di JpaRepository

```

@Test
public void testThatUserCanBeUpdated(){

    UserEntity userEntityA = TestDataUtil.createTestUserA();
    underTest.save(userEntityA);
    userEntityA.setName("Matthew");
    underTest.save(userEntityA);
    Optional<UserEntity> result = underTest.findById(
        userEntityA.getUserID());
    assertThat(result).isPresent();
    assertThat(result.get()).isEqualTo(userEntityA);
}

```

L'aggiornamento viene eseguito richiamando nuovamente il metodo save.

TestDataUtil è una classe che contiene dei metodi per la creazione di entità tramite l'utilizzo di builder() e build().

## 4.2 class UserServiceTests

Questa classe testa unitariamente i metodi dei servizi dell'entità User.

Listing 26: Testing metodo findById di UserService

```
@Test
public void testFindById() {
    UserEntity userEntity = new UserEntity();
    UUID id = UUID.randomUUID();
    userEntity.setUserID(id);
    when(repository.findById(id)).thenReturn(Optional.of(
        userEntity));

    Optional<UserEntity> result = service.findById(id);

    assertEquals(Optional.of(userEntity), result);
}
```

Si istanzia una nuova entità di tipo User e si genera un ID. A questo punto tramite `when(repository.findById(id)).thenReturn(Optional.of(userEntity))`; Mockito resituisce un Optional di tipo UserEntity con l'ID generato. A quel punto si ricerca l'utente e si asserisce che siano uguali.

Listing 27: Testing metodo deleteById di UserService

```
@Test
public void testDelete() {
    UserEntity userEntity = new UserEntity();
    UUID id = UUID.randomUUID();
    userEntity.setUserID(id);
    when(repository.findById(id)).thenReturn(Optional.of(
        userEntity));

    service.deleteById(id);

    verify(repository, times(1)).deleteById(id);
}
```

Stessa struttura del test precedente.

## 4.3 class AuthenticationServiceTests

Questa classe permette di testare che le operazioni di autenticazione vadano a buon fine.

Listing 28: Testing registrazione utente

```
@Test
public void testCustomerRegistration() {
    String email = "testCustomer@test.com";
    String password = "testPassword";
    String encodedPassword = "encodedTestPassword";

    UserEntity userEntity = new UserEntity();
    userEntity.setEmail(email);
    userEntity.setPassword(password);
    userEntity.setRole(Role.CUSTOMER);

    when(passwordEncoder.encode(password)).thenReturn(
        encodedPassword);

    authenticationService.registerUser(userEntity);

    verify(passwordEncoder, times(1)).encode(password);
    verify(userRepository, times(1)).save(any(UserEntity.
        class));
}
```

## 4.4 class UserControllerIntegrationTests

Questa classe permette di testare i controller dell'applicativo, i quali comunicano con il livello di service e per questo si parla di test di integrazione e non unitari.

Listing 29: Testing metodo create di UserController

```
@Test
public void testThatCreateUserSuccessfullyReturn201() throws
Exception {
    UserEntity testCustomerA = TestDataUtil.createTestUserA()
        ;
    testCustomerA.setUserID(null);
    String customerJSON = objectMapper.writeValueAsString(
        testCustomerA);
    mockMvc.perform(
        MockMvcRequestBuilders.post("/users")
            .contentType(MediaType.APPLICATION_JSON)
```

```

        .with(csrf()).content(customerJSON)).andExpect(
            MockMvcResultMatchers.status().isCreated());
    }

```

In questo test, si genera un nuovo utente e si setta un ID nullo. Dopodiché da un JAVA Object, lo trasformiamo in un oggetto JSON per permettere la creazione. A questo punto viene richiamata una richiesta di tipo post sul metodo create del controller che ha **@RequestMapping("/users")**. Il metodo mapperà l'oggetto JSON in un nuovo oggetto UserEntity, grazie al quale si farà il save sul database. A questo punto, se l'operazione è andata a buon fine il metodo ritornerà il codice di stato 201 (`.andExpect(MockMvcResultMatchers.status().isCreated());`).

Listing 30: Testing metodo delete di UserController

```

@Test
public void
testThatDeleteUserReturnsHttpStatus204ForExistingUser() throws
Exception {
    UserEntity testUserEntityA = TestDataUtil.createTestUserA
        ();
    UserEntity savedCustomer = userService.save(
        testUserEntityA);
    mockMvc.perform(MockMvcRequestBuilders.delete("/users/" +
        savedCustomer.getUserID())
        .contentType(MediaType.APPLICATION_JSON)
        .with(csrf())).andExpect(MockMvcResultMatchers.
        status().isNoContent());
}

```

Come per il test precedente, si crea una nuova entità e si salva sul database. Poi si fa una richiesta di delete passando il percorso corretto, che questa volta ha anche l'ID, perché vogliamo eliminare uno specifico utente. Se l'operazione va a buon fine ci si aspetta un codice di stato 204. Anche se l'utente non è presente, cancella un qualcosa che non esiste, che però non genera errore e ritorna comunque un codice 204.

Il metodo crea una nuova UserEntity e gli associa un email, una password e un ruolo (Customer). A questo punto si esegue un encoding della password e si richiama il metodo registerUser. Per verificare la correttezza, si richiamano i metodi **verify**, che controllano se il metodo encode e il metodo save siano effettivamente chiamati.

Listing 31: Testing login utente

```

@Test
public void testPhotographerLogin() {
    String username = "testPhotographer";
    String password = "testPassword";

    UserEntity userEntity = new UserEntity();
}

```



}

all'entità precedentemente simulata allora il login è avvenuto con successo.

## 4.5 Risultati dei test

Di seguito i risultati dei test effettuati con JUnit 5.

Test Name	Duration	Status
✓ UserEntityRepositoryIntegrationTests (com.SWE_photoshoot_booking.repositories)	573 ms	✓ Tests passed: 4 of 4 tests – 573 ms
✓ testThatUserCanBeUpdated()	458 ms	/usr/lib/jvm/jdk-21-oracle-x64/bin/java
✓ testThatMultipleUsersCanBeCreated()	75 ms	21:27:35.913 [main] INFO org.springframework
✓ testThatUserCanBeCreated()	21 ms	21:27:36.007 [main] INFO org.springframework
✓ testThatUserCanBeDeleted()	19 ms	

Figure 12: Output User Entity Integration Tests

Test Name	Duration	Status
UserServiceTests (com.SWE_photoshoot_booking.services)	147 ms	Passed
testFindAll()	91 ms	Passed
testSave()	10 ms	Passed
testFindById()	12 ms	Passed
testDelete()	20 ms	Passed
testUpdate()	14 ms	Passed

Tests passed: 5 of 5 tests - 147 ms

Figure 13: Output User Service Tests

✓ AuthenticationServiceTests (com.SWE_photoshoot_booking.services)	81 ms	✓ Tests passed: 4 of 4 tests – 81 ms
✓ testPhotographerLogin()	55 ms	/usr/lib/jvm/jdk-21-oracle-x64/bin/java ...
✓ testPhotographerRegistration()	13 ms	21:25:09.312 [main] INFO org.springframework
✓ testCustomerLogin()	6 ms	21:25:09.423 [main] INFO org.springframework
✓ testCustomerRegistration()	7 ms	

Figure 14: Output Authentication Service Tests

✓ UserControllerIntegrationTests (com.SWE_photoshoot_booking.controllers)	1 sec 425 ms	✓ Tests passed: 13 of 13 tests – 1 sec 425 ms
✓ testThatListUsersReturnsHttpStatus200()	684 ms	/usr/lib/jvm/jdk-21-oracle-x64/bin/java ...
✓ testThatDeleteUserReturnsHttpStatus204ForExistingUser()	92 ms	21:22:58.476 [main] INFO org.springframework
✓ testThatDeleteUserReturnsHttpStatus204ForNonExistingUser()	22 ms	21:22:58.581 [main] INFO org.springframework
✓ testThatCreateUserSuccessfullyReturn201()	67 ms	
✓ testThatGetUserReturnsHttpStatus200WhenUserExist()	27 ms	
✓ testThatCreateUserSuccessfullyReturnUser()	94 ms	
✓ testThatPartialUpdateExistingUserReturnsHttpStatus200Ok()	283 ms	
✓ testThatFullUpdateUserReturnsHttpStatus404WhenUserNotExists()	32 ms	
✓ testThatPartialUpdateExistingUserReturnsUpdateUser()	38 ms	
✓ testThatFullUpdateUserReturnsHttpStatus200WhenUserExists()	24 ms	:: Spring Boot :: (v3.2.3)
✓ testThatGetUserReturnsHttpStatus404WhenNoUserExist()	15 ms	2024-05-03T21:22:59.009+02:00 INFO 2964
✓ testThatFullUpdateUpdatesExistingUser()	26 ms	2024-05-03T21:22:59.010+02:00 INFO 2964:
✓ testThatListUsersReturnsListOfUsers()	21 ms	2024-05-03T21:22:59.694+02:00 INFO 2964

Figure 15: Output User Controller Integration Tests

## 5 Conclusioni

Nel complesso, quest'applicativo al momento permette la gestione dei photoshoot, degli orari e della creazione degli appuntamenti. Non ho implementato la possibilità di visualizzare gli appuntamenti all'utente con la possibilità di modificarli o eliminarli. Per tutte le altre sezioni invece è stato fatto, si tratterebbe solo di replicare questo processo anche per questo. Anche il fotografo al momento non può visualizzare gli appuntamenti che sono stati presi con lui, ma comunque, come prima, si tratta di eseguire le stesse operazioni. Non ho implementato un sistema di notifica nel caso di cancellazione di appuntamenti, implementabile attraverso `spring-boot-starter-mail`. Per la connessione al database ho utilizzato docker, configurandolo nel file `docker-compose.yml`.

## 6 Librerie di terze parti

**JUnit** - per i test

<https://junit.org/junit5/>

**Spring Data JPA** - per la gestione e il collegamento al database MySQL.

<https://spring.io/projects/spring-data-jpa>

**Spring Security** - per la sicurezza web.

<https://spring.io/projects/spring-security>

**UML Diagrams** - per la generazione di diagramma di classe UML, poi adattato manualmente. Integrato in IntelliJ IDEA Ultimate.

**ModelMapper** - per la conversione da/a DTO-Entità.

<https://modelmapper.org/>