DOCUMENTAZIONE SOFTWARE PHARMAZON

Prova finale di Ingegneria del Software

Studente: Andrea Marino

Codice Persona: 10765006

Matricola: 963109

Link Front-End: clicca qui.

Link Back-End: clicca qui.

Indice

1. Introduzione 3
2. Requisiti Tecnici 3
3. Requisiti Funzionali 5
4. Descrizione della progettazione 7
4.1 Use Case Diagram 7
4.2 Package Diagram 8
4.3 Class Diagram 10
4.3.1 Util 10
4.3.2 Configuration 11
4.3.3 Controller 12
4.3.4 Security 14
4.3.5 Dto 16
4.3.6 Model 17
4.3.7 Service – ImplementationClass 19
4.3.9 Repository 23
4.3.10 Exception 25
4.3.11 Services, Chat & E-mail 26
4.4 Class Diagram - Design Pattern 29
4.4.1 State 29
4.4.2 Observer 31
5. Test 34
5.1 Configurazione dei Test 34
5.1 Code Coverage (Copertura del codice) 35
6. Database 36
6.1 Diagramma E-R 36
Prospettive future 37

1. Introduzione

L'applicazione web, chiamata **Pharmazon**, è stata progettata con lo scopo di semplificare l'acceso hai prodotti farmaceutici e servizi, messi a disposizione da una farmacia e la comunicazione tra farmacista e cliente. Nella realizzazione dell'applicativo, sono stati individuati due principali attori: il farmacista che svolge il ruolo di amministratore ed il cliente che svolgere il ruolo di acquirente.

Questa applicazione si presenta come un e-commerce, capace di permettere ai nostri clienti di:

- effettuare degli **acquisti** relativi ai prodotti farmaceutici che richiedono o meno l'utilizzo della ricetta per l'acquisto;
- prenotarsi per alcuni **servizi** messi a disposizione da parte della farmacia, come ad esempio: il test per la celiachia;
- sfruttare il servizio di **chat** per comunicare con il farmacista per determinati dubbi o domande.

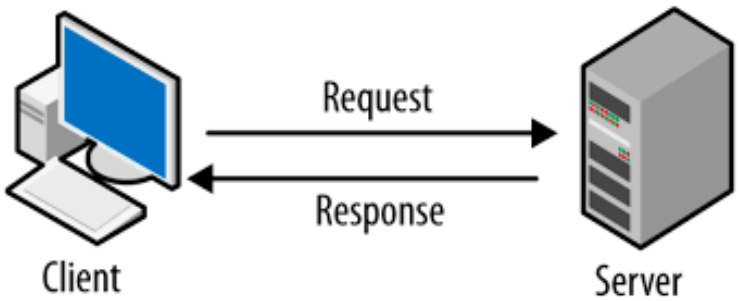
L'idea alla base della realizzazione di questo applicativo non è soltanto permettere al cliente una maggior semplificazione negli acquisti farmaceutici e prenotazione ai servizi, ma rendere più semplice ai clienti impossibilitati, di raggiungere la farmacia, anche questo aspetto così essenziale della vita di tutti i giorni. Perciò l'applicativo è **rivolto a tutti**.

2. Requisiti Tecnici

L'applicazione è basata sul **modello architetturale MVC** (Model-View-Controller), che permette la separazione tra i diversi livelli logici, e una maggior gestione e manutenzione del codice.

Inoltre, è utilizzato il **pattern DTO** (Data Transfer Object) che garantisce un ulteriore livello di astrazione dei dati.

L'applicativo segue il modello Client-Server e la comunicazione tra le parti avviene utilizzando l'**HTTPS** tramite un certificato self-signed (non legato a Certificate Authority).



L'applicazione è costituita da due parti: il Back-End e il Front-End.

Per la parte **Front-End**, si utilizzano le seguenti tecnologie:

- **Angular 17**, è il framework utilizzato per la realizzazione dell'applicativo, che sfrutta la versione 20.8.0 di Node ed utilizza come linguaggio di programmazione TypeScript;
- **Angular Material**, è una libreria che consente di utilizzare alcuni componenti grafici, pronti all'uso per l'implementazione all'interno del codice;
- **STOMP** (Simple Text Oriented Messaging Protocol), è un protocollo di messaggistica testuale, che permette la comunicazione tra il client web e server di messaggistica.

Per la parte **Back-End**, si utilizzano le seguenti tecnologie:

- **Java 17**, capace di introdurre delle semplificazioni per la programmazione e il supporto a nuovi framework;
- **Spring Boot**, nella versione 3.2, è il framework utilizzato, che consente al programmatore di concentrarsi sulla logica di business, piuttosto che sulla gestione delle dipendenze, e di sfruttare le *@Annotation,* che consentono delle configurazioni automatiche;
- **Spring Security**, permette l'autenticazione e autorizzazione degli utenti;
- **Java Persistence API** (JPA), consente di mappare le classi scritte in Java e sulla base di queste realizzare le nostre tabelle nel Database;
- **WebSocket**, protocollo di rete basata sul TCP e viene utilizzato per la realizzazione della chat tra farmacista e cliente;
- **JUnit 5**, è il framework utilizzato per la realizzazione dei test unitari;
- **Java Mail Sender**, libreria Java utilizzata per poter inviare le e-mail, e sfrutta il protocollo **SMTP** (Simple Mail Transfer Protocol).

Per la parte di **archiviazione dei dati** (Database), è stato utilizzato il **MySQL**, un sistema per la gestione di database relazionali, open-source.

È possibile, inoltre, riscontrare i parametri di configurazione relativi al database, all'utilizzo del protocollo HTTPS e SMTP all'interno del file **application.properties**, situato nel path: src\main\resources\application.properties. Per utilizzare il database bisogna, su MySql, creare un database chiamato *pharmazon*.

3. Requisiti Funzionali

Di seguito sono riportati i requisiti funzionali dell'applicativo:

- Utente non registrato nel sistema, può effettuare:
  - la registrazione, che consente di creare una nuova utenza di tipo *cliente.*

- Utente registrato nel sistema, sia cliente o farmacista, può effettuare:
  - il login, che consente di accedere e autenticarsi sull'applicativo come *cliente* o *farmacista* (admin);
  - richiesta di una nuova password, in caso di smarrimento, tramite e-mail.
  - visualizzazione dei prodotti (caso cliente: solo quelli disponibili);
  - visualizzazione dei servizi (caso cliente: solo quelli non scaduti);
  - ricerca dei prodotti per nome o categoria;
  - ricerca dei servizi per nome;
  - visualizzare dettagli di un prodotto o servizio;
  - accedere all'area personale;
  - modifica informazioni personali;
  - utilizzare il servizio di chat;
  - logout.

- Utente registrato come cliente, può effettuare:
  - aggiunta o rimozione di un prodotto dal carrello, dove la rimozione implica l'eliminazione del prodotto dal carrello;
  - aggiunta o rimozione di una prenotazione di un servizio, dove la rimozione implica l'eliminazione;
  - aggiunta, rimozione e modifica carta di credito, dove la rimozione non implica l'eliminazione ma la disattivazione;
  - aggiunta, rimozione e modifica indirizzi, dove la rimozione non implica l'eliminazione ma la disattivazione;
  - visualizzazione carte di credito;
  - visualizzazione indirizzi;
  - visualizzazione storico e stato ordini;
  - visualizzazione carrello;
  - effettuare acquisti con l'aggiunta di prescrizioni (per i prodotti che lo richiedono) o meno, carta di credito e indirizzo di consegna;
  - aggiungere, modificare, eliminare un feedback, dei prodotti consegnati;
  - eliminazione account, con notifica inviata tramite e-mail.

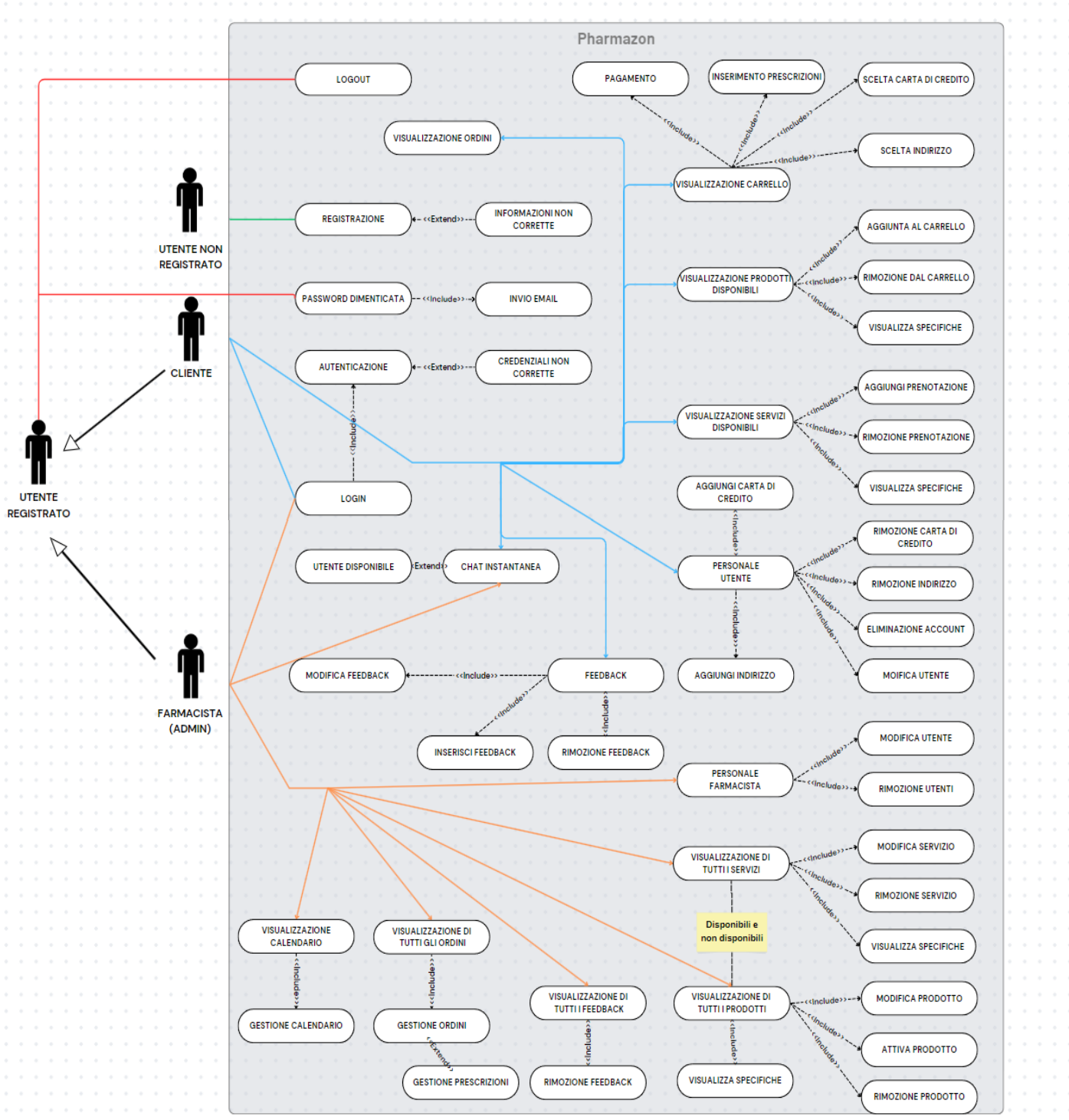
- Utente registrato come farmacista, può effettuare:
  - aggiunta, modifica e rimozione dal catalogo dei prodotti di un prodotto, dove la rimozione non implica l'eliminazione, ma la disattivazione (con possibilità di riattivarlo);
  - aggiunta, modifica e rimozione dal catalogo dei servizi di un servizio, dove la rimozione implica l'eliminazione;
  - approvare o declinare le ricette;
  - aggiunta e modifica categoria;
  - storico degli ordini;
  - modificare lo stato dell'ordine:
    attesa di approvazione, *preparazione*, *in transito*, *consegnato* o *eliminato*
  - accettare o rifiutare le prenotazioni;
  - visualizzare il calendario delle prenotazioni accettate;
  - eliminare gli account dei clienti, con notifica inviata tramite e-mail.

- Aggiornamenti lato cliente:
  - aggiornamento stato ordine, tramite e-mail;
  - aggiornamento rimozione di un prodotto dal carrello se ha subito modifiche o è stato rimosso dal catalogo, tramite e-mail;
  - aggiornamento stato prenotazione a servizio, tramite e-mail;
  - aggiornamento, stato quantità di un prodotto (terminato o prossimo a terminare) che ha acquistato in passato ed è stato consegnato, tramite e-mail.

- Aggiornamenti lato farmacista:
  - aggiornamento, sulla cancellazione di un servizio, che ha creato (e non ha subito modifiche) o che ha modificato per ultimo, tramite e-mail;
  - aggiornamento sulla quantità di un prodotto (terminato o prossimo a terminare), che ha creato (e non ha subito modifiche) o che ha modificato per ultimo, tramite e-mail.

4. Descrizione della progettazione

4.1 Use Case Diagram



Matricola: 963109 Andrea Marino

Il diagramma, qui riportato, prende il nome di **Use Case Diagram** e consente di descrivere i **casi d'uso** (azioni o servizi) che il sistema offre ai nostri **attori**, in questo caso i nostri utenti: *utente non registrato*, *utente registrato*, *farmacista* e *cliente*, dove questi ultimi due sono due tipologie di utenti registrati e che ereditano delle funzionalità e caratteristiche di *utente registrato*.

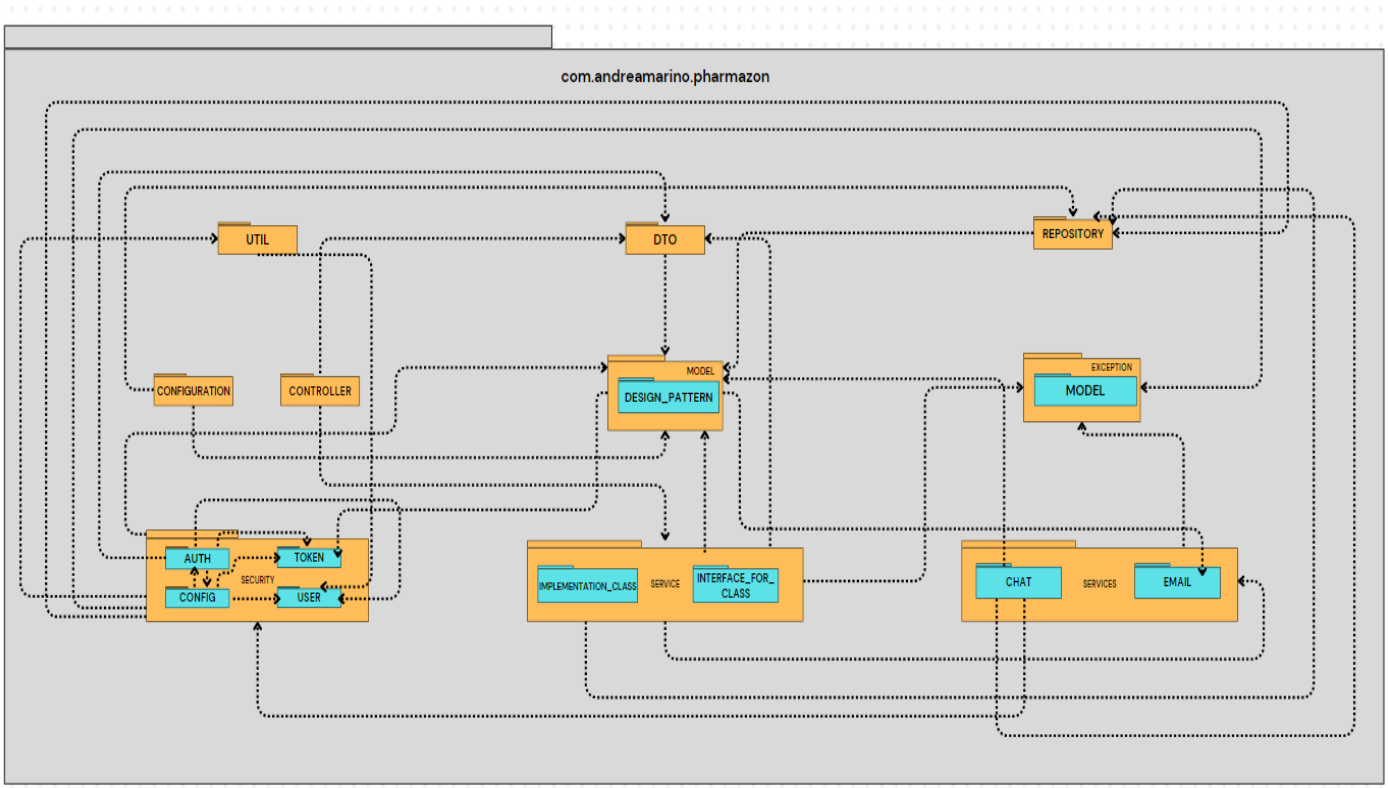
Notare la presenza delle linee che collegano gli attori ai casi d'uso, ed indicano le diverse interazioni appunto tra l'attore ed il caso d'uso e tali collegamenti vengono chiamati **associazioni**. Le linee continue che collegano l'Utente Registrato con il Farmacista e il Cliente, indicano che il Cliente o il Farmacista sono attori che ereditano i casi d'uso dell'attore Utente Registrato.

È possibile anche visionare sul diagramma i concetti di *extends* e *include* utilizzati per rappresentare le relazioni tra i casi d'uso:

- include, indica che un caso d'uso incorpora il comportamento di un altro caso d'uso;
- extends, indica che un caso d'uso può essere esteso con un comportamento aggiuntivo opzionale.

Per riportare un esempio, vediamo che lato Admin, abbiamo la visualizzazione del calendario, ma questo incorpora (include) anche la gestione del calendario stesso. Invece, se l'utente non registrato effettua l'azione "Registrazione", questa estende anche il controllo sulle informazioni inserite, ed eventualmente l'avviso di informazioni non corrette inserite.

4.2 Package Diagram



Ecco la descrizione dei packages:

- **Util**, contiene una classe con tutti i metodi utilizzati per la validazione dei dati;
- **Configuration**, contiene delle classi utili alla creazione dei bean e inizializzazione dei dati;
- **Controller**, contenente i controller dell'applicazione, che ci permettono di gestire le richieste e risposte HTTP;
- **Security**, racchiude alcuni packages utili per la sicurezza dell'applicativo:
    - **Auth**, contiene il controller, il service e il model per gestire l'autenticazione all'applicativo;
    - **Config**, contiene principalmente un filtro per intercettare le richieste HTTP e validare il token JWT dell'header di autorizzazione;
    - **Token**, include il modello dell'entità e il repository associato ad essa;
    - **User**, permette l'ottenimento delle autorità (ruoli) associati all'utente.

- **DTO** (Data Transfer Object), include tutte quelle classi impiegate per definire gli oggetti, utilizzati per il trasferimento dei dati tra i diversi livelli dell'applicativo;
- **Model**, permette di definire quelle classi che vanno a rappresentare il modello di dominio dell'applicativo e racchiude:
    - **Design Pattern**, contenente la definizione delle interfacce e classi come *State*, impiegate nella realizzazione dei design pattern;
- **Service**, all'interno troviamo tutte quelle classi che implementano la *logica di business*, fornendo le necessarie funzioni per l'utilizzo dell'applicativo e inoltre, sono presenti le cartelle:
    - **interfaceForClass**, racchiude tutte le interfacce implementate dalle classi concrete (relative ai servizi);
    - **implementationClass**, racchiude tutte quelle classi che implementano le interfacce citate precedentemente;
- **Repository**, contiene le interfacce che estendono JpaRepository, ottenendo alcuni metodi predefiniti ereditati (utilizzati per operazione CRUD), e contengono anche metodi personalizzati;
- **Exception**, include il gestore delle eccezioni RestExceptionHandler ed alcune classi che estendono *RunTimeExcpetion*, in modo da utilizzare delle eccezioni predefinite, ed è presente la cartella:
    - **Model**, contenete la classe utilizzata dal gestore delle eccezioni;
- **Services**, include i servizi, di chat ed e-mail utilizzati nell'applicativo e rintracciabili nelle sottocartelle:
    - **Chat**, contenente le classi di configurazione per il WebSocket;
    - **Email**, contenente le classi di configurazione per l'utilizzo del servizio di e-mail.

4.3 Class Diagram

4.3.1 Util

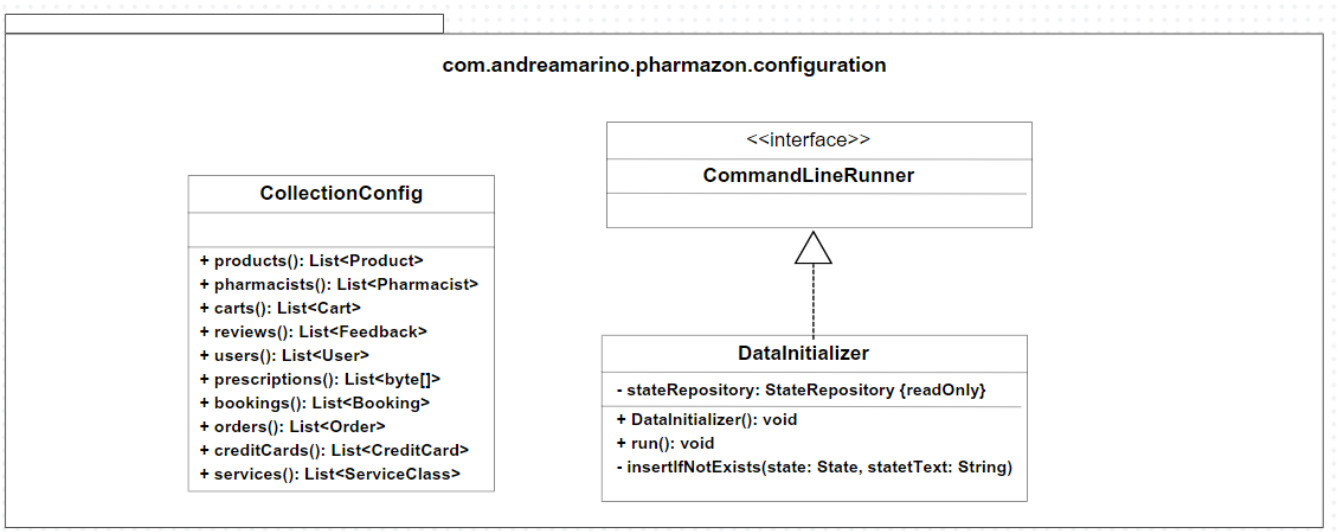
Tale immagine rappresenta il package *com.andreamarino.pharmazon.util*, contenente tutti quei metodi utilizzati all'interno dell'applicativo, per poter validare il contenuto dei dati inviati da parte dell'utente.

Ad esempio, nel caso dell'inserimento del nome e cognome da parte dell'utente, si vorrebbe che questo dato abbia solo lettere ed è qui che viene in aiuto il metodo *containsOnlyLetters*. Oppure nell'ipotesi in cui si volesse verificare la data di nascita dell'utente, per vedere se quest'ultimo è maggiorenne o meno, tale controllo può essere effettuato con il metodo *checkAdult*.

Non sono presenti solo metodi mirati al controllo dei dati personali / anagrafici, ma anche relativi alla validazione dei dati della carta di credito, dove viene verificato il CVC, la lunghezza del numero della carta di credito e la data di scadenza di quest'ultima.

Con questa classe ovviamente, abbiamo il vantaggio di **non** dover riscrivere più volte lo stesso codice, diminuendo così le righe di codice e gli errori provocati dalla stesura del nuovo codice inserito.

4.3.2 Configuration

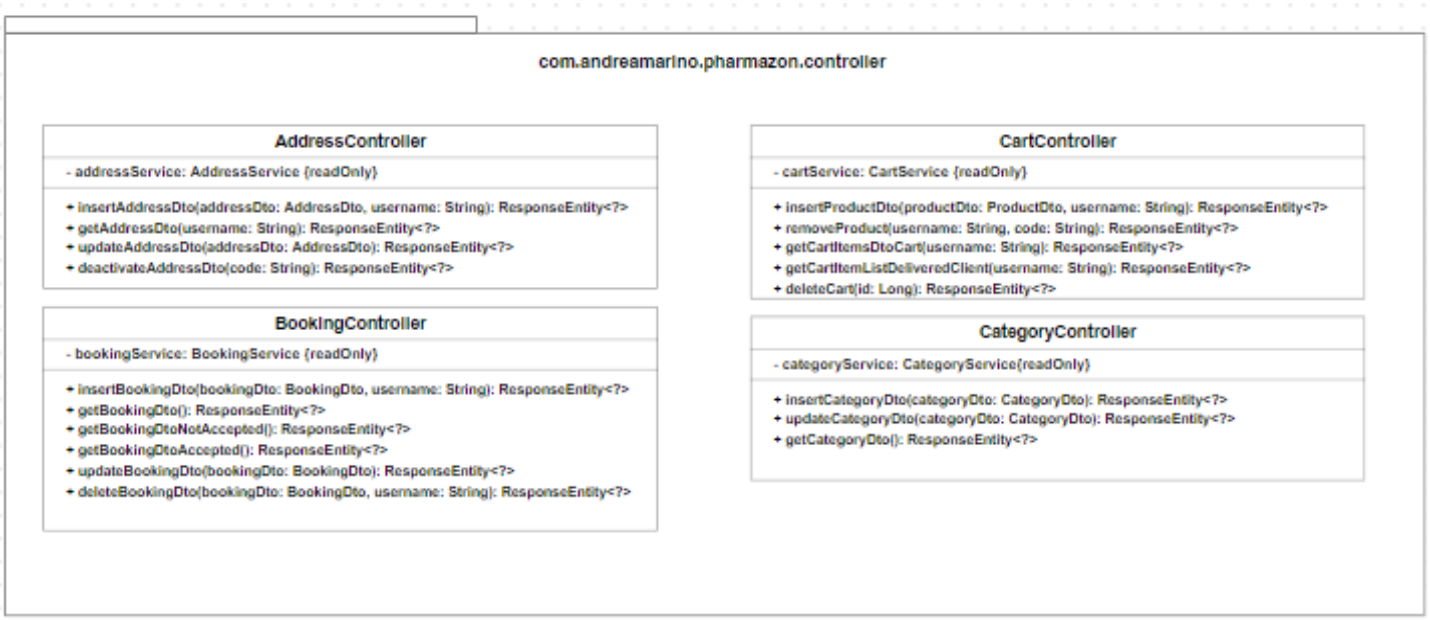


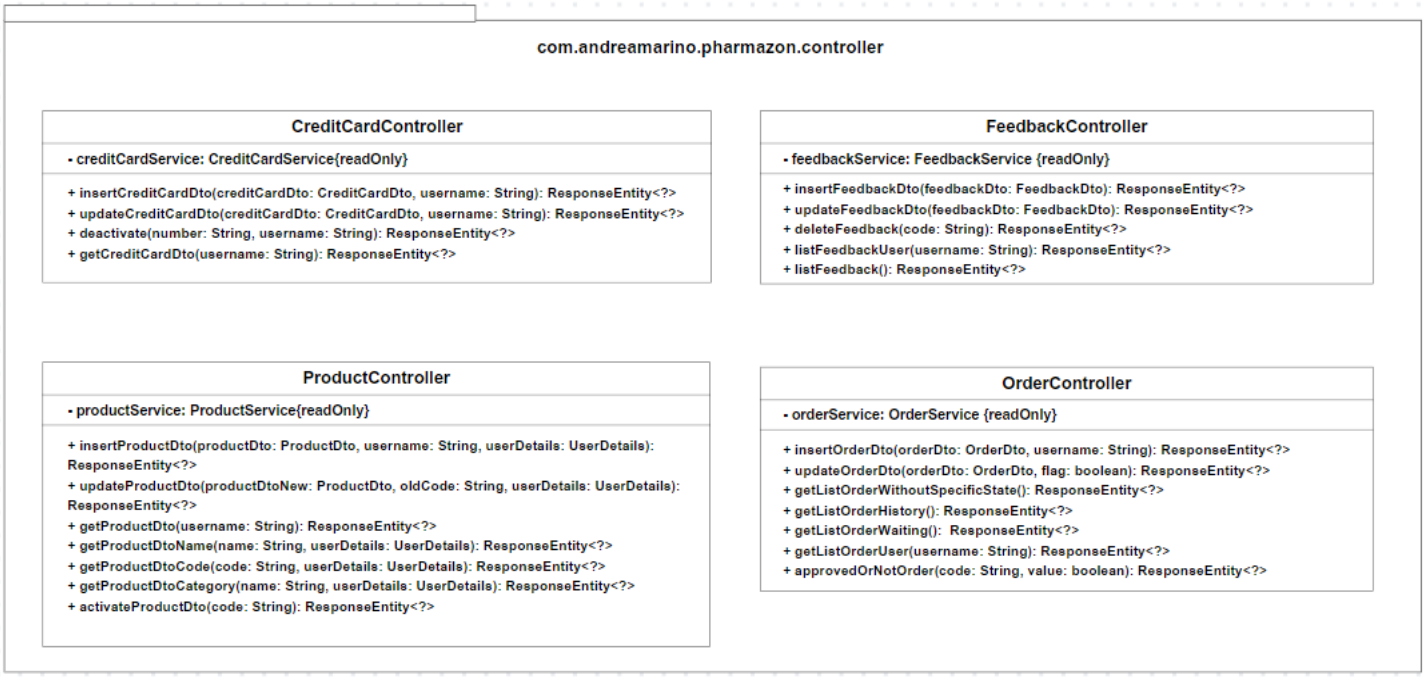
Il package *com.andreamarino.pharmazon.configuration*, riportato in figura, racchiude tutte quelle classi di configurazione utili, per definire i **bean** (oggetto che è stato istanziato) e per inizializzare alcuni dati richiesti per il funzionamento dell'applicativo.

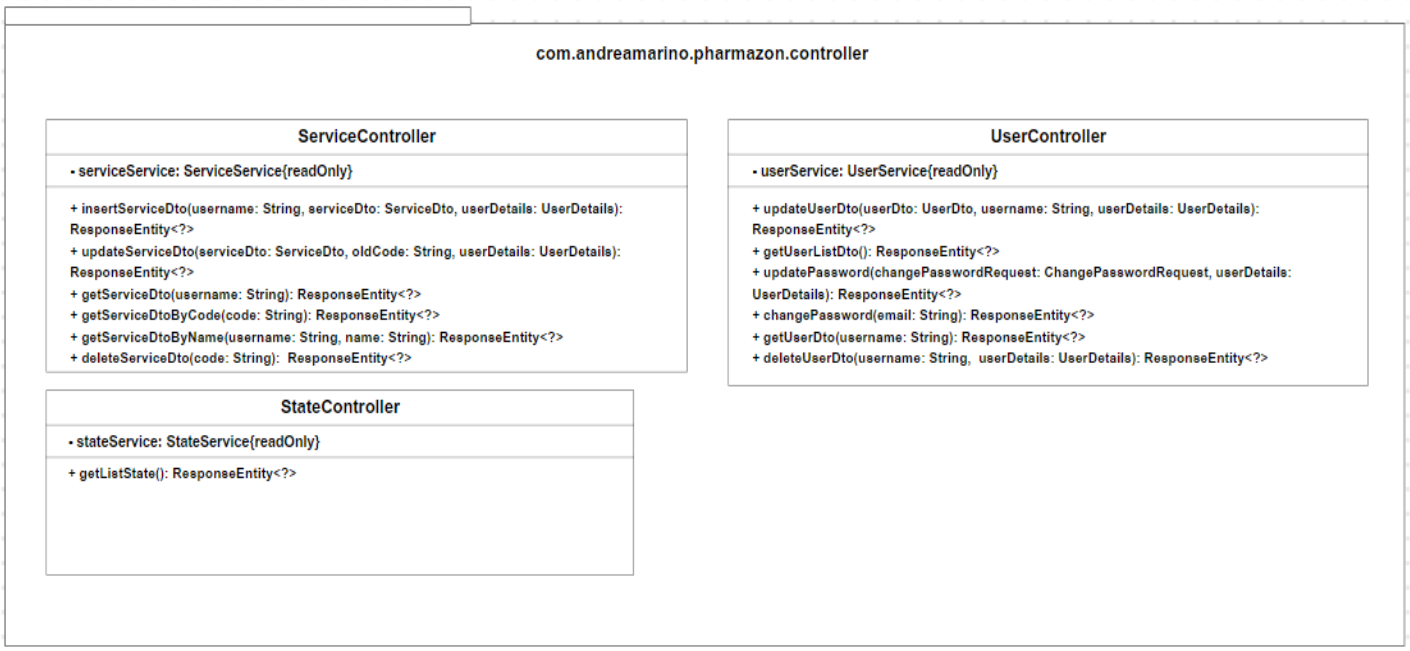
Nello specifico la classe CollectionConfig viene annotata con *@Configuration*, che sta ad indicare che questa classe va a definire i bean che verranno gestiti da Spring. Invece i metodi all'interno di questa classe verranno annotati tramite l'annotation *@Bean*, e ciò significa che il metodo produce un bean, che verrà iniettato nei vari componenti dell'applicativo e gestito da Spring. Da notare anche l'utilizzo dell'annotation *@Qualifier*, che consente di specificare un **qualificatore** per distinguere i diversi bean dello stesso tipo.

Infine, la classe DataInitializer che implementa l'interfaccia CommandLineRunner, consente di eseguire una porzione di codice subito dopo l'avvio dell'applicazione e ciò avviene inserendo la nostra porzione di codice all'interno del metodo *run*.
In questo caso, andiamo a salvare sul database (se non presenti) gli stati che gli ordini potranno avere e necessari per il corretto funzionamento dell'applicativo, infatti questi stati sono predefiniti. Da notare anche l'utilizzo del *@Component*, che indica come tale classe sia un componente di Spring e dell'annotation *@Autowired* che indica l'iniezione automatica delle dipendenze.

4.3.3 Controller







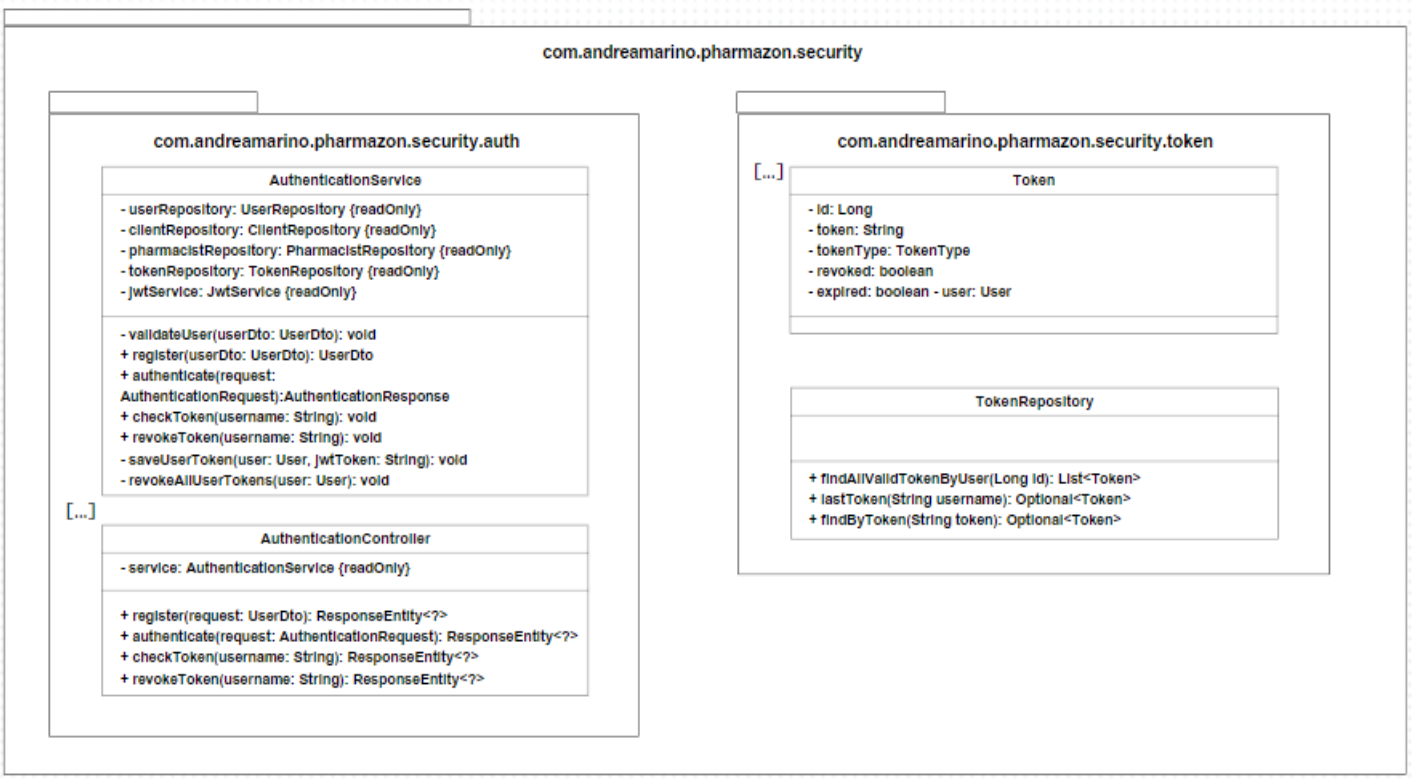
Queste ultime tre figure (per questione di leggibilità sono state separate), fanno riferimento ad un unico package cioè il *com.andreamarino.pharmazon.controller* contenente tutti i controller dell'applicazione. Il suo obiettivo principale è quello di intercettare le richieste HTTP, tramite gli endpoint REST. Gli endpoint, infatti, sono **punti di accesso** a un'applicazione, associati ad un URL. Ogni metodo del controller restituirà un oggetto di tipo ResponseEntity, consentendo l'utilizzo di una risposta HTTP personalizzabile.

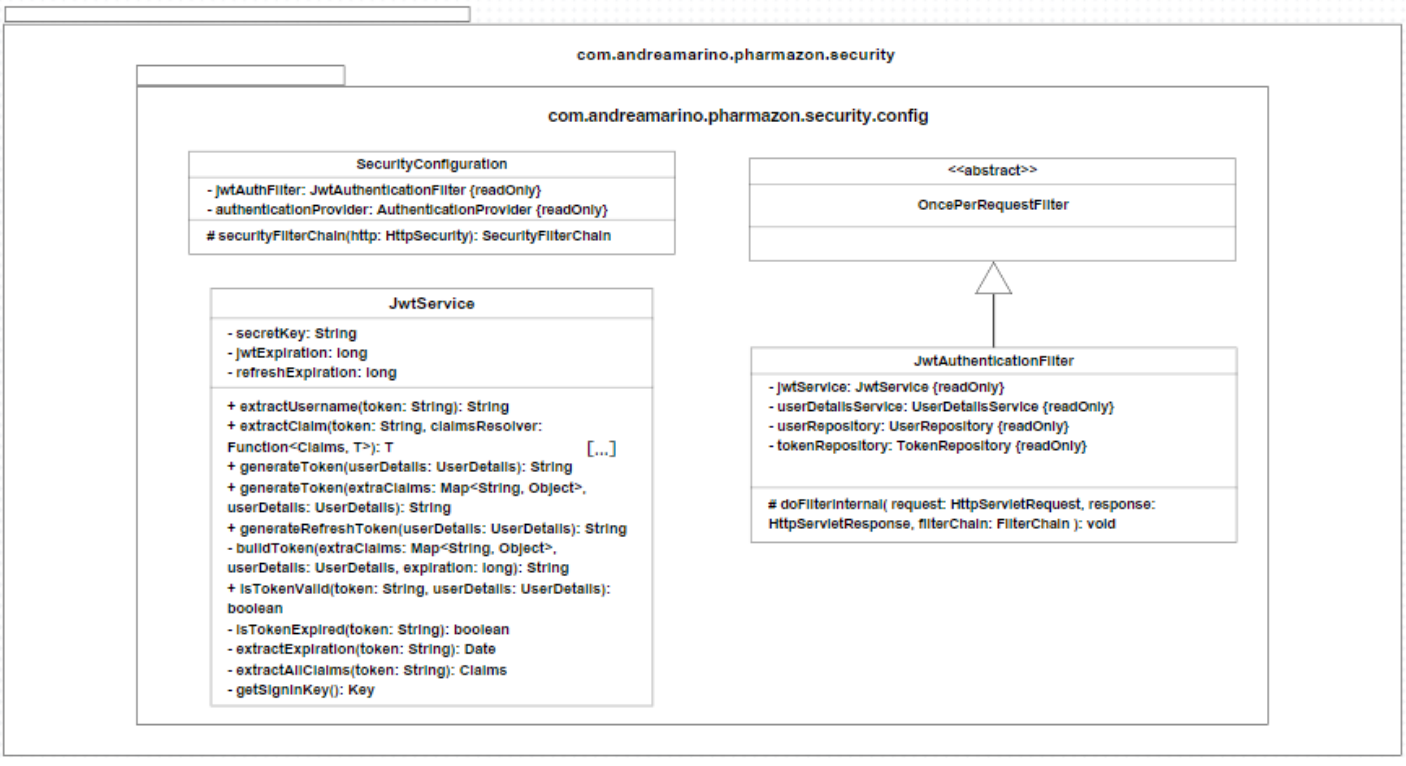
All'interno dei controller possiamo trovare delle annotation:

- @RestController, una combinazione di annotation, @Controller e @ResponseBody, che indicano come la classe sia un controller in un'applicazione Spring MVC e che restituisca i dati in formato JSON;
- @RequiredArgsConstructor, fa parte della libreria Lombok, e permette di generare automaticamente un costruttore, consentendo di iniettare le dipendenze, e conviene utilizzare tale libreria piuttosto che @Autowired, perché così facendo si evita la scrittura esplicita del costruttore;
- @RequestMapping, annotazione utilizzata per mappare le richieste HTTP;
- @CrossOrigin, specifica le origini autorizzate ad effettuare le richieste cross-origin (CORS), e quindi quali origini possono accedere alla risorsa dell'applicazione;
- @PostMapping, indica che il metodo verrà chiamato quando viene effettuata una richiesta HTTP POST al relativo endpoint (per semplicità abbiamo riportato POST, ma è anche presente l'annotation per la GET, DELETE e PUT);
- @RequestParam e @RequestBody, consentono di estrarre dei dati complessi dalla richiesta, dove il primo estrarrà parametri semplici, il secondo il corpo.

Inoltre, in ogni classe con @RestController, è presente l'opportuna documentazione ai metodi.

4.3.4 Security





Queste due ultime immagini, mostrano il package: *com.andreamarino.pharmazon.security*.

Per la gestione della sicurezza, l'applicativo sfrutta le funzionalità concesse dal framework **Spring Security**, insieme al **JWT** (JSON Web Token).

Nello specifico il token è così costituito:

- Header, contenente il tipo di token e l'algoritmo di firma;
- Payload, dichiarazioni (chiamate **claims**) sfruttati per trasmettere delle informazioni che riguardano il client e il server;
- Signature, usato per verificare che il token non sia modificato.

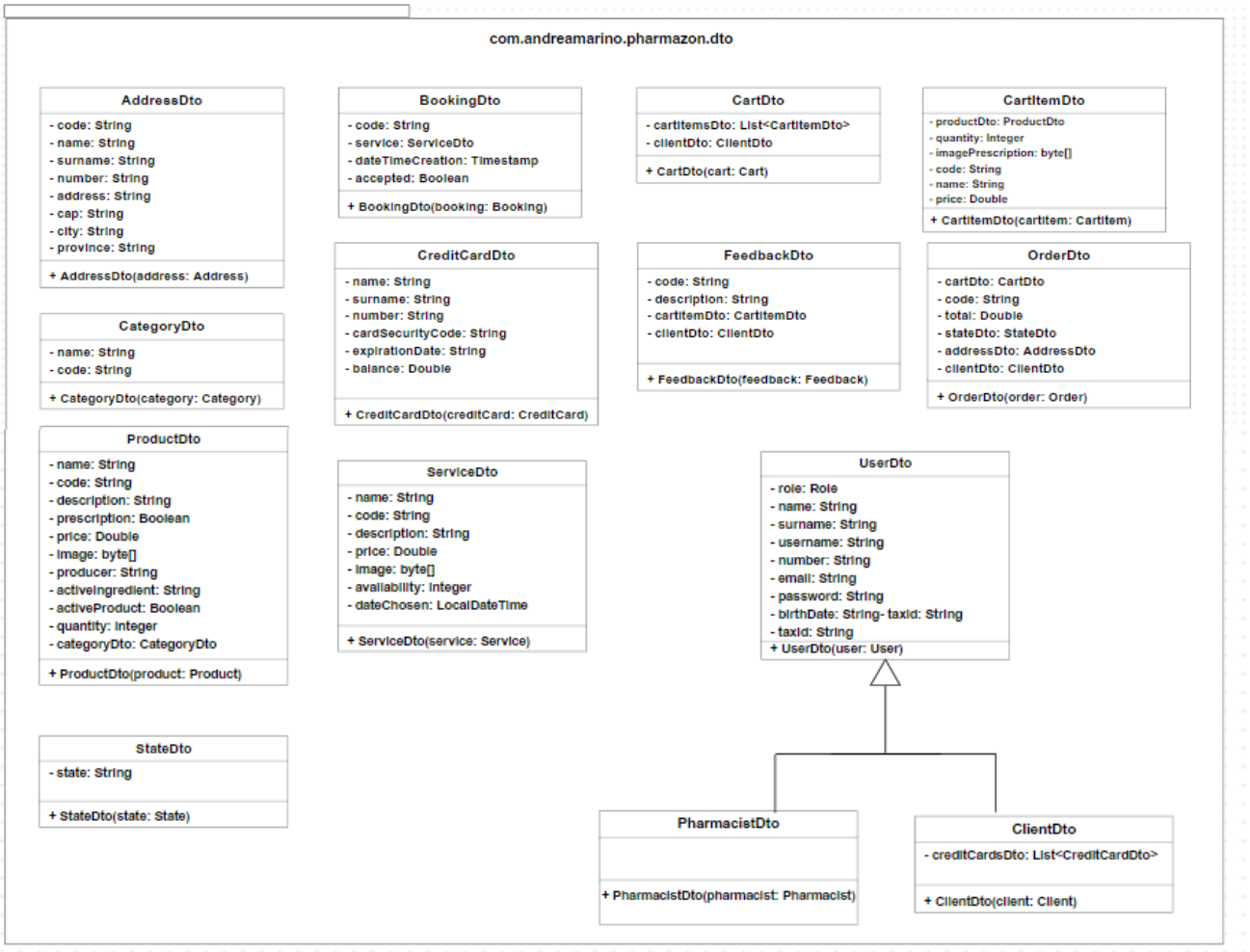
Nel package è possibile trovare le seguenti classi:

- AuthenticationService, per la gestione delle richieste di autenticazione e registrazione degli utenti;
- JwtAuthenticationFilter, filtro di sicurezza che intercetta le richieste HTTP e verifica la presenza e validità di un token JWT;
- JwtService, servizio per la gestione dei token JWT, ed include metodi per generare, firmare, validare e analizzare i token;
- SecurityConfiguration, definisce una *catena* di filtri di sicurezza, la SecuriyFilterChain, che specifica come gestire le richieste HTTP in base ai ruoli degli utenti (ADMIN e CLIENT).

Analizzando nello specifico SecurityConfiguration è possibile trovare alcune componenti fondamentali che lo caratterizzano:

- WHITE_LIST_URL, url accessibili senza autenticazione;
- jwtAuthFilter, filtro di autenticazione JWT, utilizzato per gestire l'autenticazione basata su token JWT;
- authenticationProvider, provider di autenticazione per autenticare gli utenti;
- securityFilterChain, metodo che configura le impostazioni di sicurezza.

4.3.5 Dto



In questa ultima figura, vediamo la rappresentazione sottoforma di class diagram, del package *com.andreamarino.pharmazon.dto*. Dove l'obiettivo è quello di sfruttare il pattern DTO (Data Transfer Object), che ci consente di trasferire i nostri dati tra diversi componenti del sistema, come appunto il Back-End ed il Front-End.

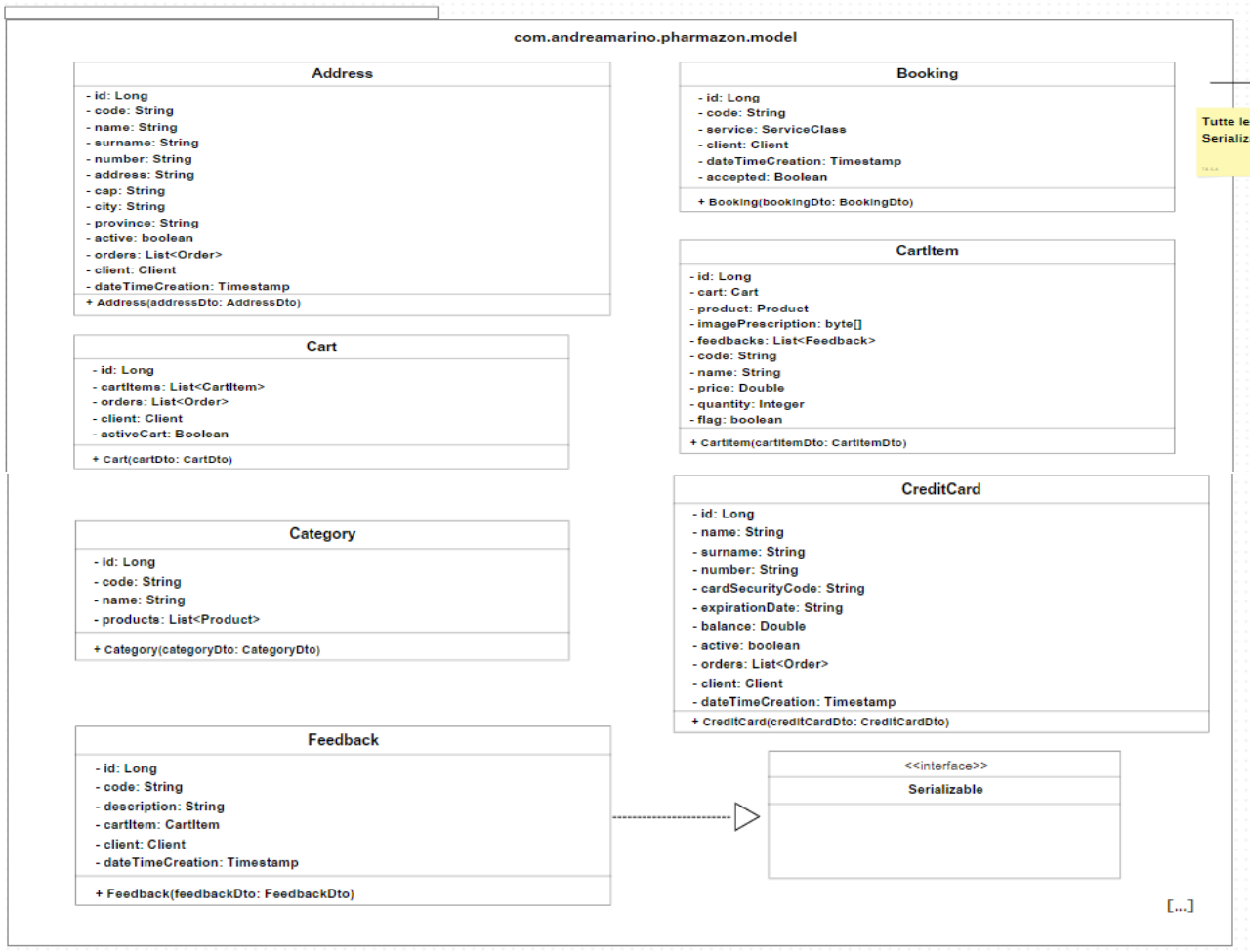
I DTO, infatti, vengono utilizzati proprio per incapsulare i dati ed inseguito trasferirli, consentendo una **modellazione** delle informazioni che verranno scambiate, ed evitando di esporre dati non utili, migliorando così anche la sicurezza dell'applicativo.

Inoltre, nel diagramma è possibile notare una ereditarietà (freccia continua) tra le due classi PharmacistDto e ClientDto, nei confronti di UserDto.

In queste classi sarà comune trovare annotation:

- @Data, funzionalità di Lombok e consente di generare metodi getter e setter, toString, equals e hashCode;
- @NoArgsConstructor e @AllArgsConstructor, dove il primo permette di generare il costruttore senza argomenti, mentre il secondo un costruttore che accetta tutti gli attributi della classe come argomenti.

4.3.6 Model

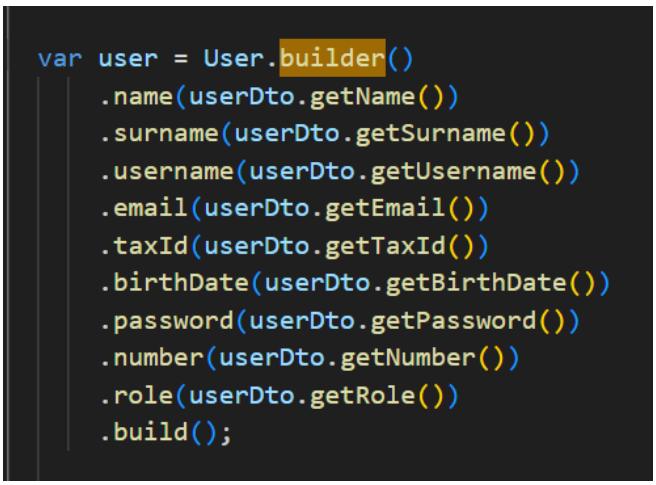


In figura viene riportato il package *com.andreamarino.pharmazon.model* sottoforma di class diagram, che riporta i modelli associati alle entità di dominio applicativo. Queste entità possono contenere delle logiche di business, come dei comportamenti e tutte le classi implementano l'interfaccia *Serializable*, che consente la serializzazione e deserializzazione degli oggetti, consentendo di essere trasferiti tramite rete.

Le classi inoltre riportano le seguenti annotation:

- @Entity che indica che la classe è una entità JPA, che va a rappresentare una tabella nel database;
- @Id, indica l'attributo che rappresenterà la chiave primaria nella tabella del database;
- @GeneratedValue, specifica che il valore della chiava primaria sarà generato **automaticamente** nel database;
- @Column, il nome della colonna con cui l'attributo è mappato nel database;
- @ManyToOne, @OneToMany, @ManyToMany, definiscono le relazioni tra le nostre entità;

- @Inheritance, specifica che tipo di strategia di ereditarietà verrà utilizzata dalla nostra entità;
- @JsonIdentifyInfo, permette di gestire le relazioni bidirezionali nelle serializzazioni JSON;
- @CreationTimestamp, automaticamente permette di popolare l'attributo con il valore corrente di data e ora;
- @Scope("prototype"), indica che ogni richiesta per questo bean produrrà una nuova istanza;
- @Builder, genera il builder pattern per la classe, consentendo di creare i nostri oggetti in modo "fluido", ecco un esempio di cosa ci permette di fare, all'interno dell'applicativo:



```
var user = User.builder()
    .name(userDto.getName())
    .surname(userDto.getSurname())
    .username(userDto.getUsername())
    .email(userDto.getEmail())
    .taxId(userDto.getTaxId())
    .birthDate(userDto.getBirthDate())
    .password(userDto.getPassword())
    .number(userDto.getNumber())
    .role(userDto.getRole())
    .build();
```

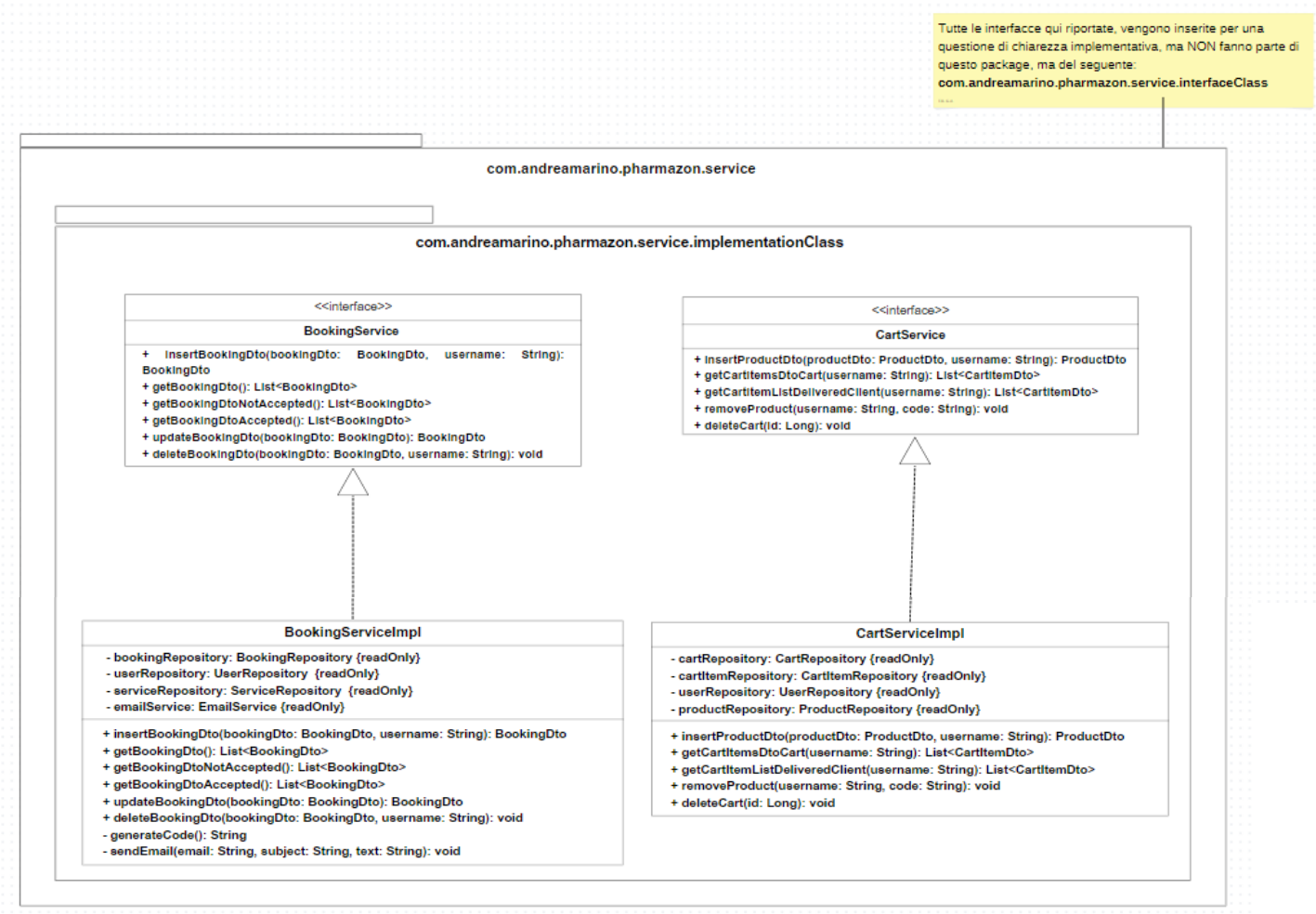
Sono presenti anche @Data, @NoArgsConstructor e @AllArgsConstructor. Alcune classi non sono state inserite all'interno della rappresentazione tramite immagine, perché verranno trattate in seguito.

4.3.7 Service – ImplementationClass

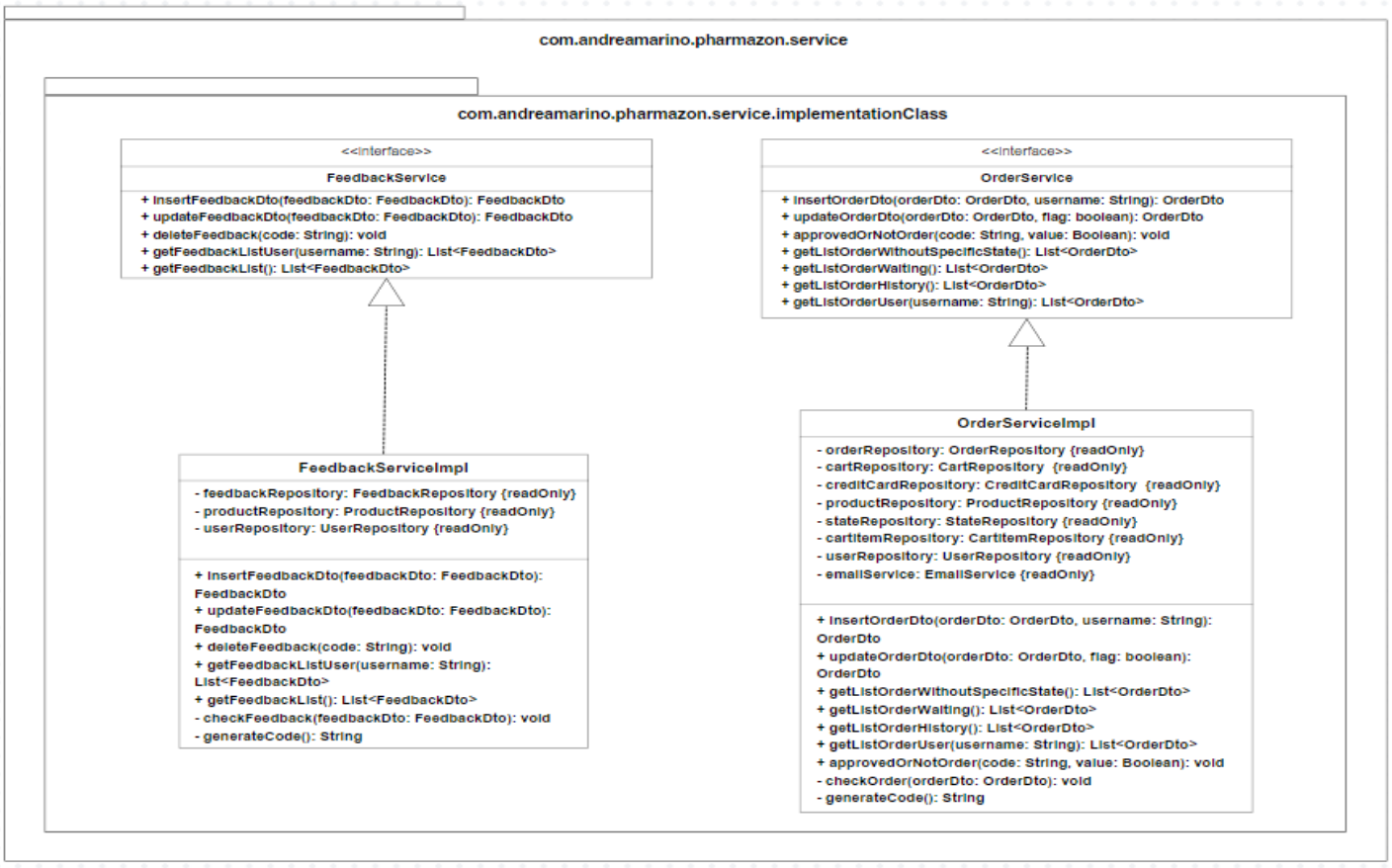
Queste immagini che seguiranno, presentate in questo paragrafo, ci permettono di rappresentare un package fondamentale per l'utilizzo e creazione dell'applicativo, il *com.andreamarino.service.implementationClass*.

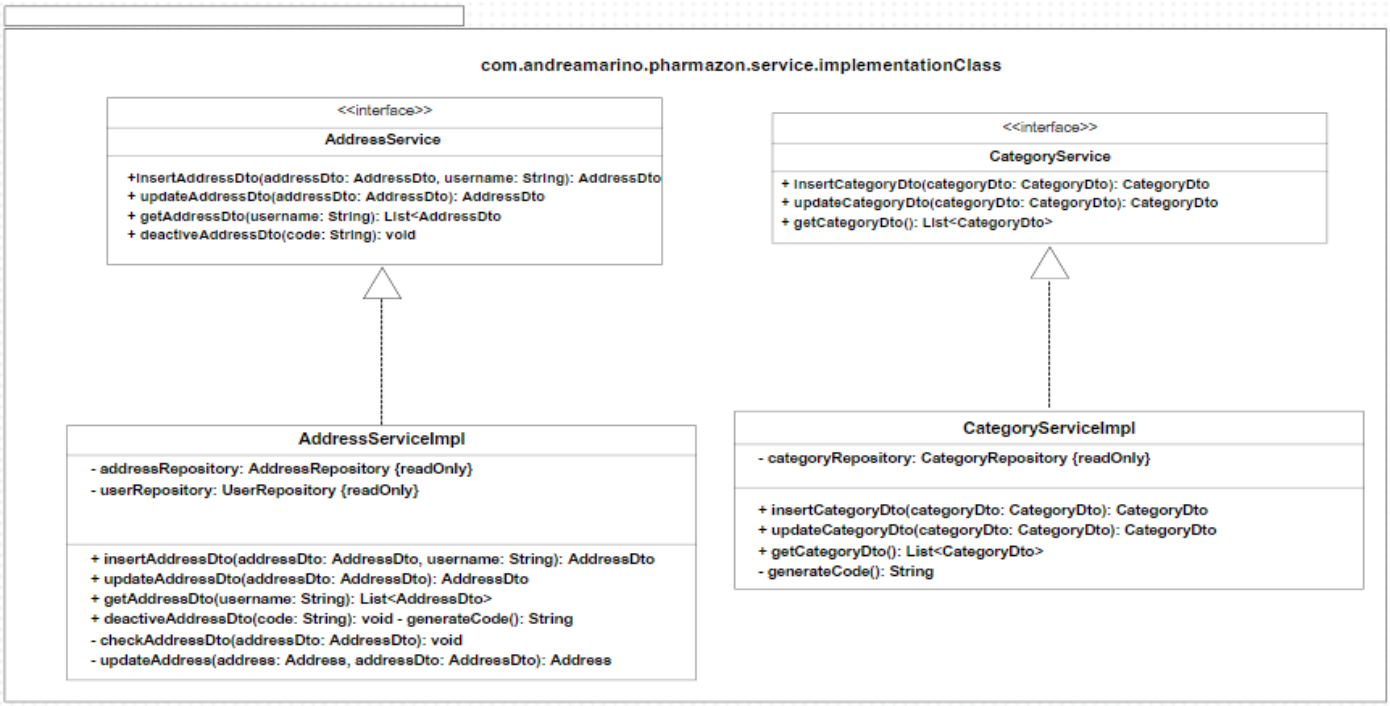
Tale package si trova tra il livello di presentazione, il controller e il livello di accesso ai dati, il repository. Contiene tutte quelle classi che vanno ad implementare la logica di business, permettendo di effettuare: delle **verifiche di validità sui dati** che devono essere inseriti, gestire le **transazioni** assicurando che avvengano, in modo sicuro, le operazioni di lettura e scrittura e per questione di completezza ed importanza, le tre figure, riportano tutte le classi.

Tutte le classi riportate all'interno di questo package, e che abbiamo mostrato tramite queste immagini, implementano la corrispondente interfaccia, molto utile per promuovere il **Principio di Inversione delle Dipendenze**, che rientra nei principi SOLID, dove i moduli di alto livello, non dovrebbero dipendere dai moduli di basso livello.



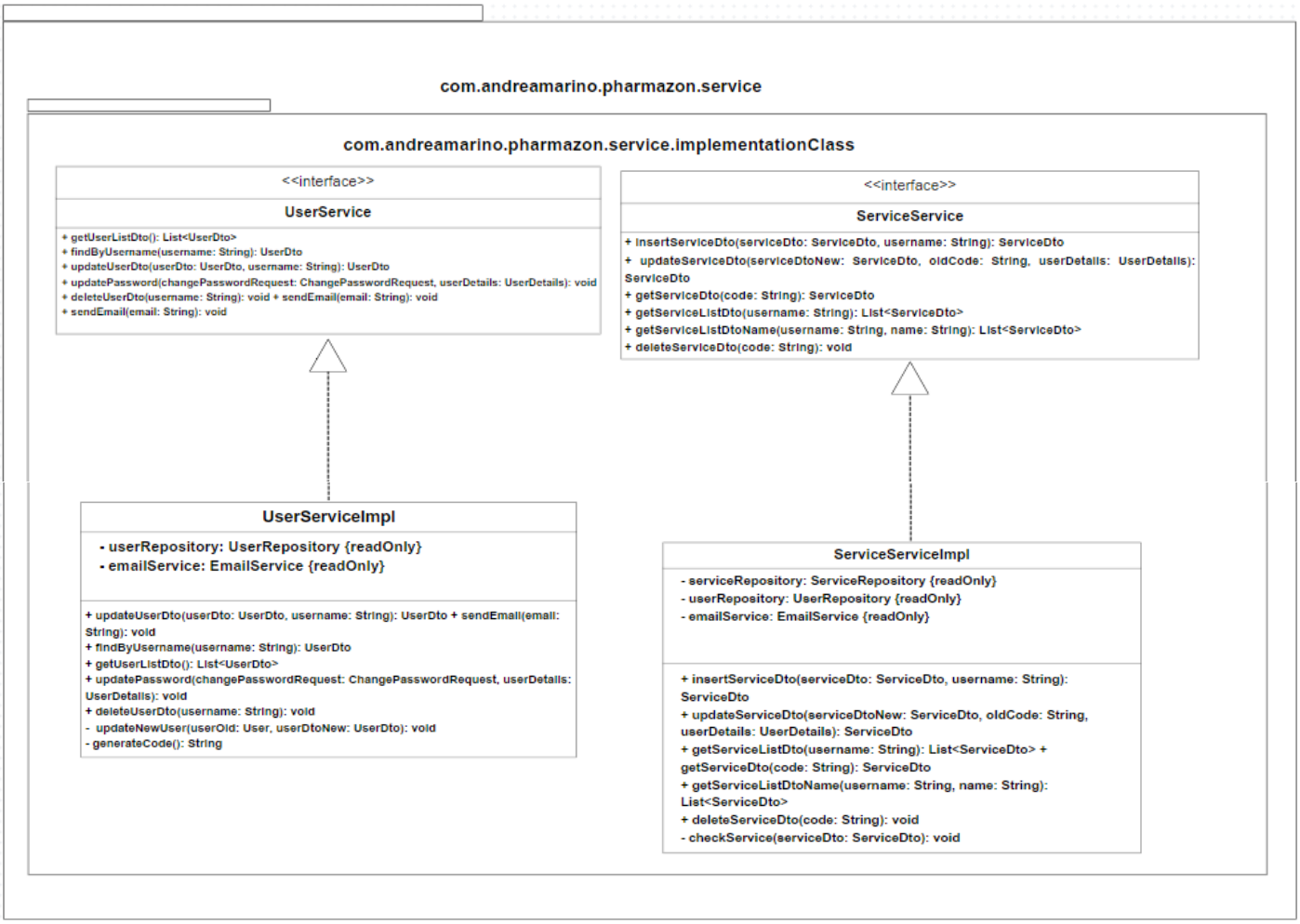
In questa immagine, vengono rappresentati: BookingService e CartService e cioè i due servizi che ci permettono di gestire le prenotazioni e il carrello associato all'utente.

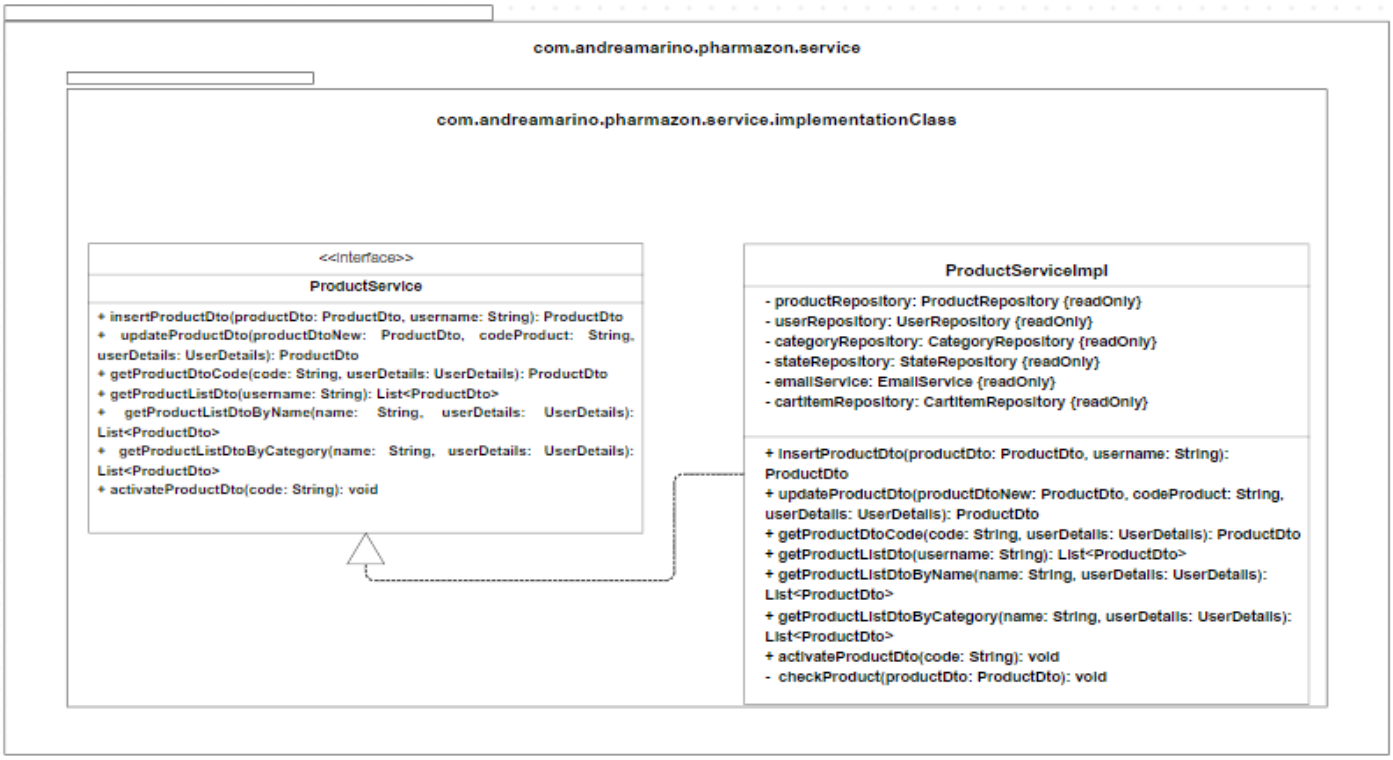




Con queste immagini, vengono mostrati i relativi class diagram per l'implementazione dei servizi:

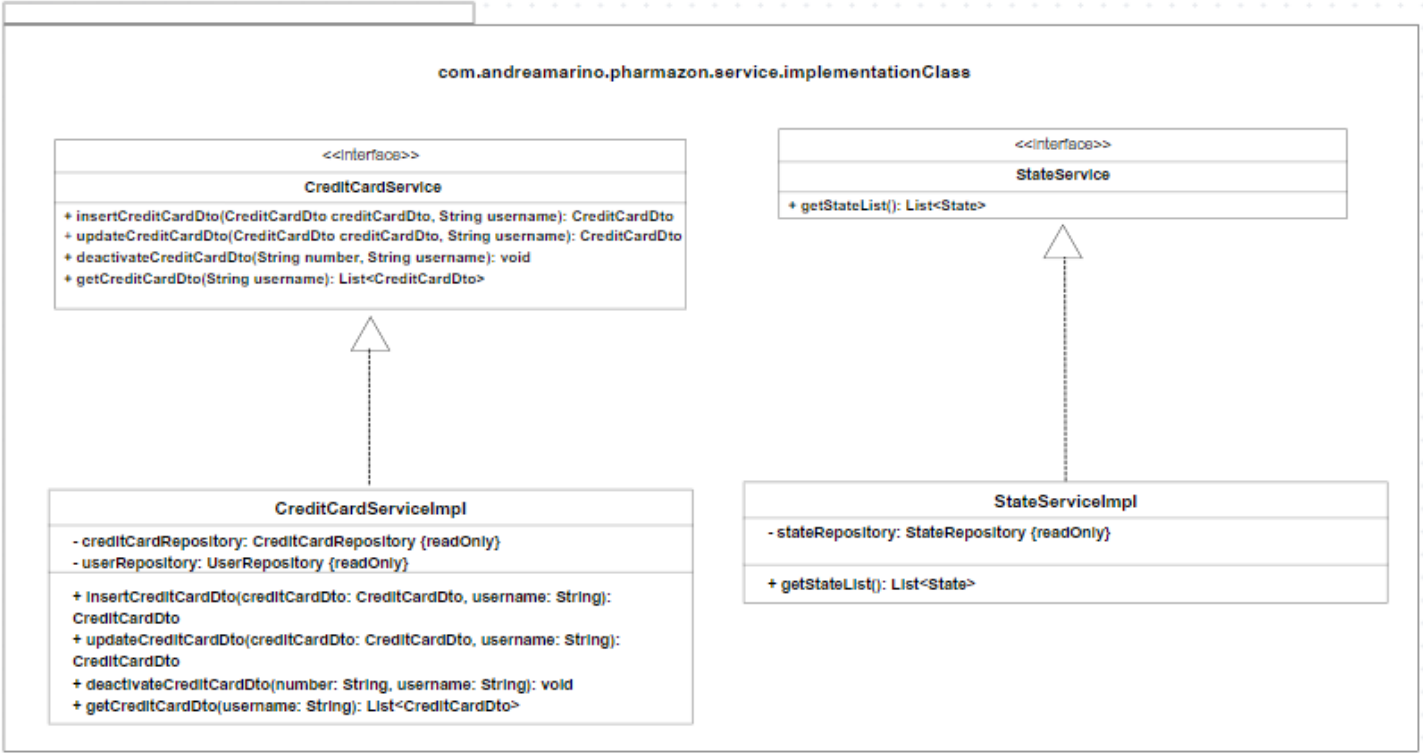
- FeedbackService, servizio utilizzato per la gestione dei feedback;
- OrderService, servizio utilizzato per la gestione degli ordini;
- AddressServcie, servizio utilizzato per la gestione degli indirizzi;
- CategoryService, sevizio utilizzato per la gestione delle categorie associate ai prodotti.





Troviamo invece, in queste due ultime figure, i seguenti service:

- ProductService, servizio utilizzato per la gestione dei prodotti;
- ServiceService, servizio utilizzato per la gestione dei servizi;
- UserService, servizio utilizzato per la gestione degli utenti.



Infine, in quest'ultima figura, ci sono i seguenti service:

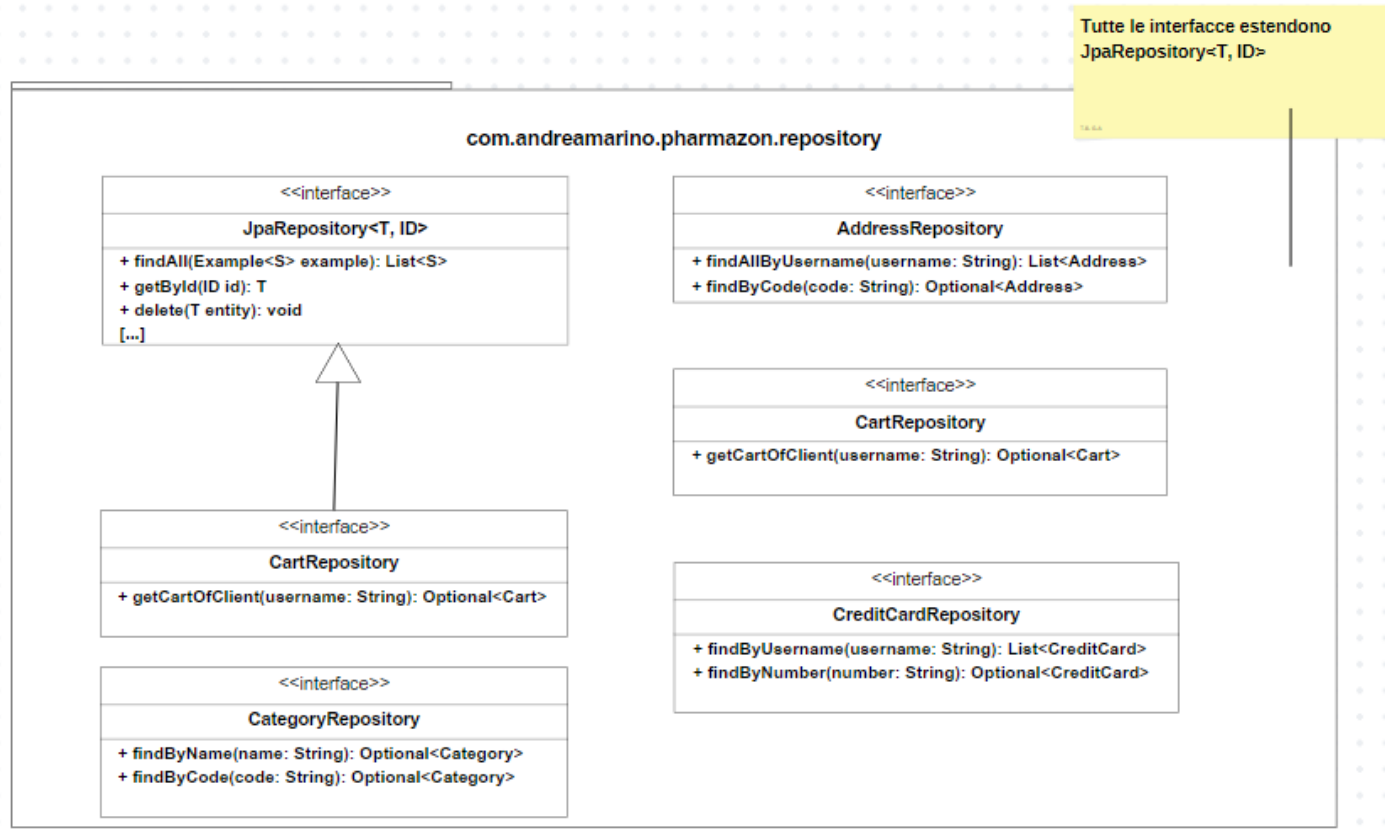
- CreditCardService, servizio utilizzato per la gestione delle carte di credito;
- StateService, sevizio utilizzato per l'ottenimento delle informazioni degli stati associati agli ordini.

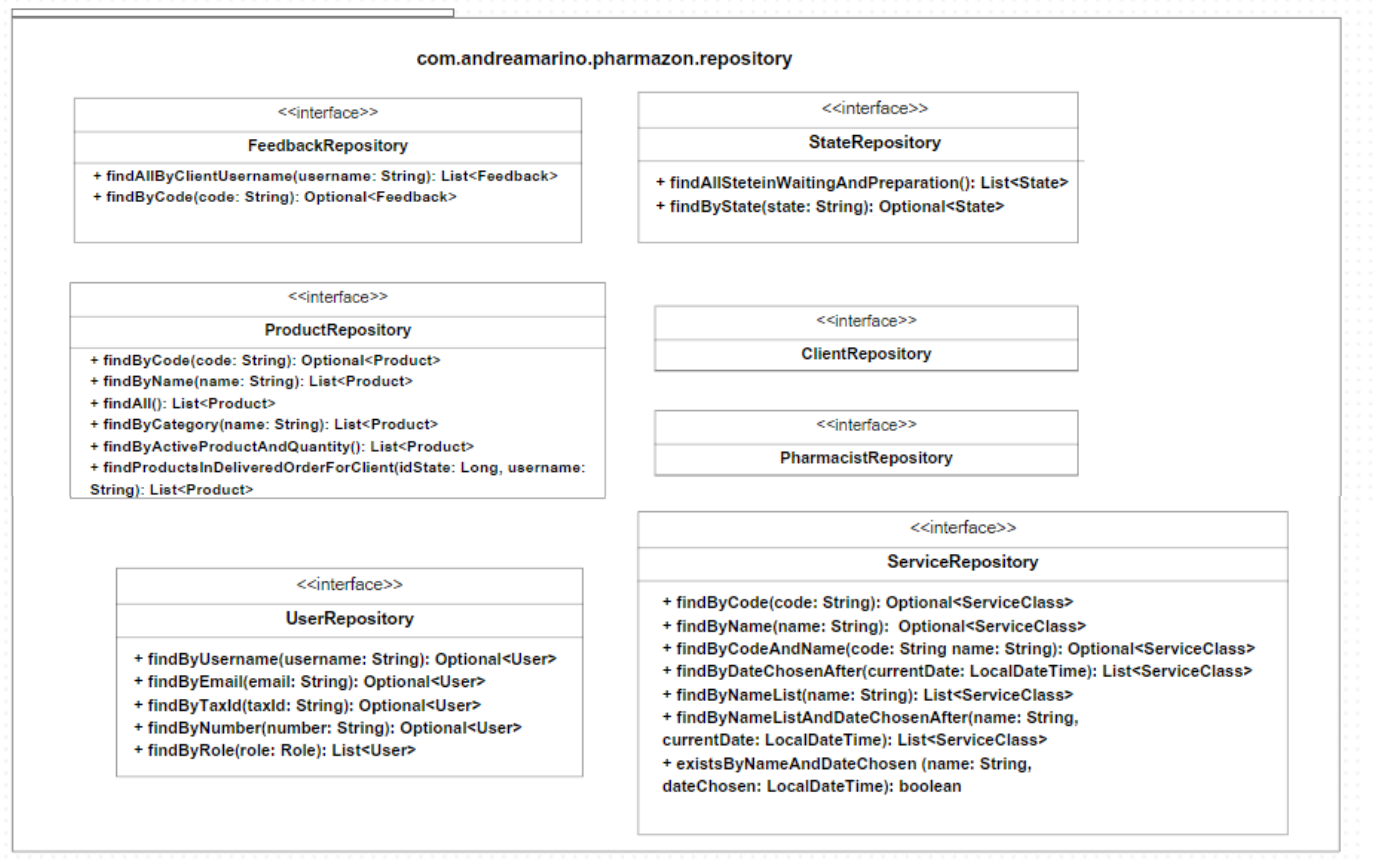
L'annotation che contraddistingue questo package, è senza dubbio *@Service*, che serve ad annotare una classe come Service. Facendo così, stiamo dicendo a Spring Boot che questa classe è un componente Spring, e deve essere gestista dal contesto dell'applicazione. Inoltre, questa annotation, va a sottolineare come in questa classe sia presentata la logica di business, permettendo di migliorare la leggibilità del codice.

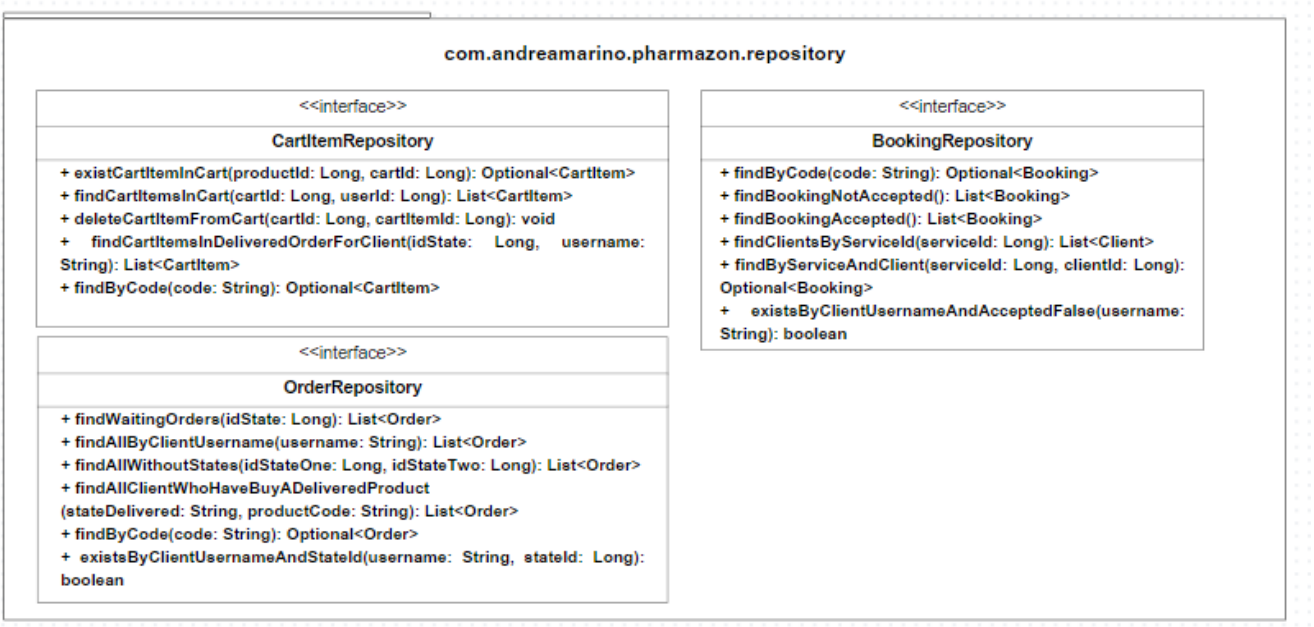
Le altre annotation presenti, sono *@Transactional* che indica come il metodo deve essere eseguito all'interno di una transazione (Spring andrà a gestire automaticamente l'inizio e il commit o rollback della transazione) in base all'esito del metodo, e *@RequiredArgsConstructor* e *@Autowired*.

Considerando l'importanza del package si ritiene opportuno mostrare tutte le classi, tramite class diagram e, per evitare inutili ripetizioni, sono anche rappresentate le interfacce implementate dai service e che si trovano nel package: *com.andreamarino.pharmazon.service.interfaceForClass.*

4.3.9 Repository







Il package *com.andreamarino.pharmazon.repository*, riporta tutte le interfacce che estendono JpaRepository. Così facendo infatti, abbiamo accesso ad alcuni metodi predefiniti che ci permettono di eseguire le operazioni CRUD, ad esempio:

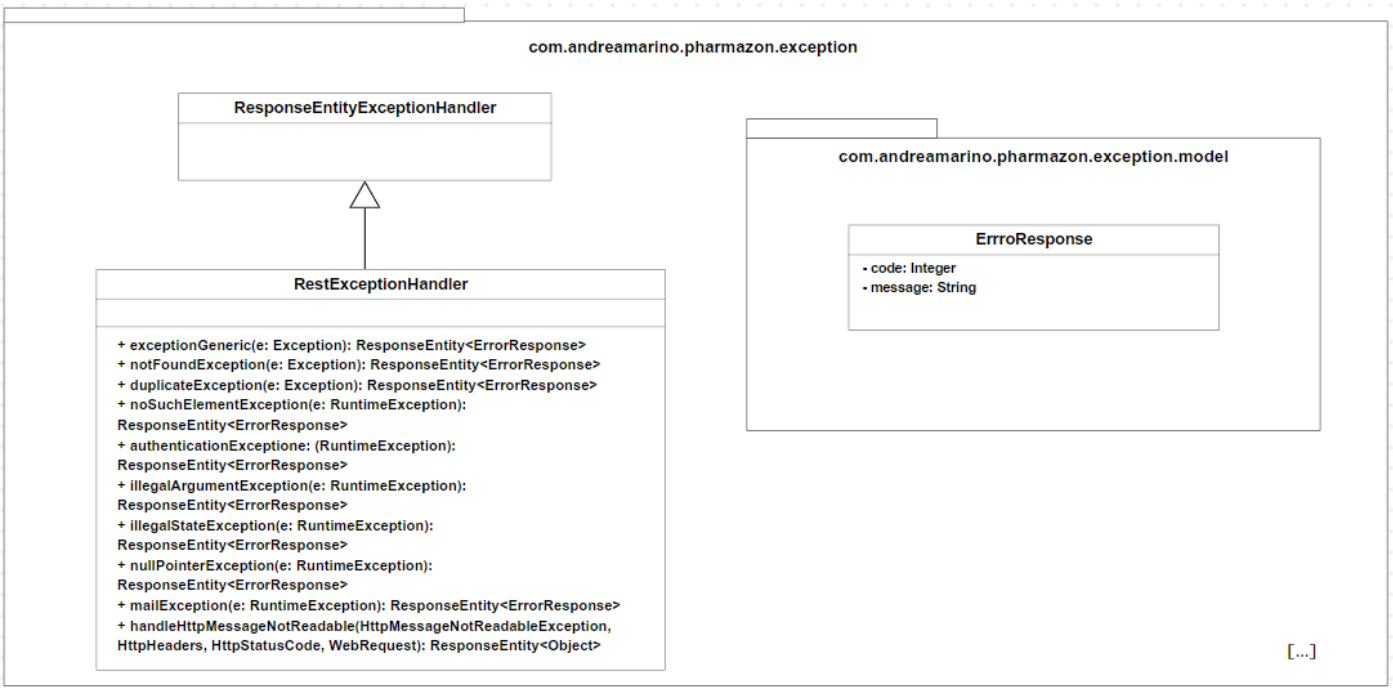
- save, permette di salvare l'entità nel database;
- findById, metodo molto utile per ricercare una entità tramite id nel database;
- deleteById, cancellare una entità tramite id;
- e molto altro.

Tutte le interfacce qui presenti vanno ad utilizzare l'annotation *@Repository*, che è una delle annotazioni fornite da Spring e che serve per indicare che è un componente responsabile della gestione dell'accesso ai dati.

Inoltre, è possibile anche trovare le seguenti annotazioni, in alcune di queste interfacce:

- @Query, utilizzata per definire query personalizzate per l'accesso ai dati;
- @Modifying, (utilizzata insieme all'annotation @Query) è usata pe definire query di aggiornamento o eliminazione dei dati.

4.3.10 Exception



Il package *com.andreamarino.pharmazon.excpetion*, contiene una classe che ci permette di poter gestire le eccezioni. Nello specifico tale classe prende il nome di RestExceptionHandler, ed è il gestore globale delle eccezioni della nostra applicazione Spring Boot. È configurata in modo da restituire una risposta HTTP appropriata, quando viene sollevata una certa eccezione.

La classe RestExceptionHandler predispone al suo interno alcune annotation come:

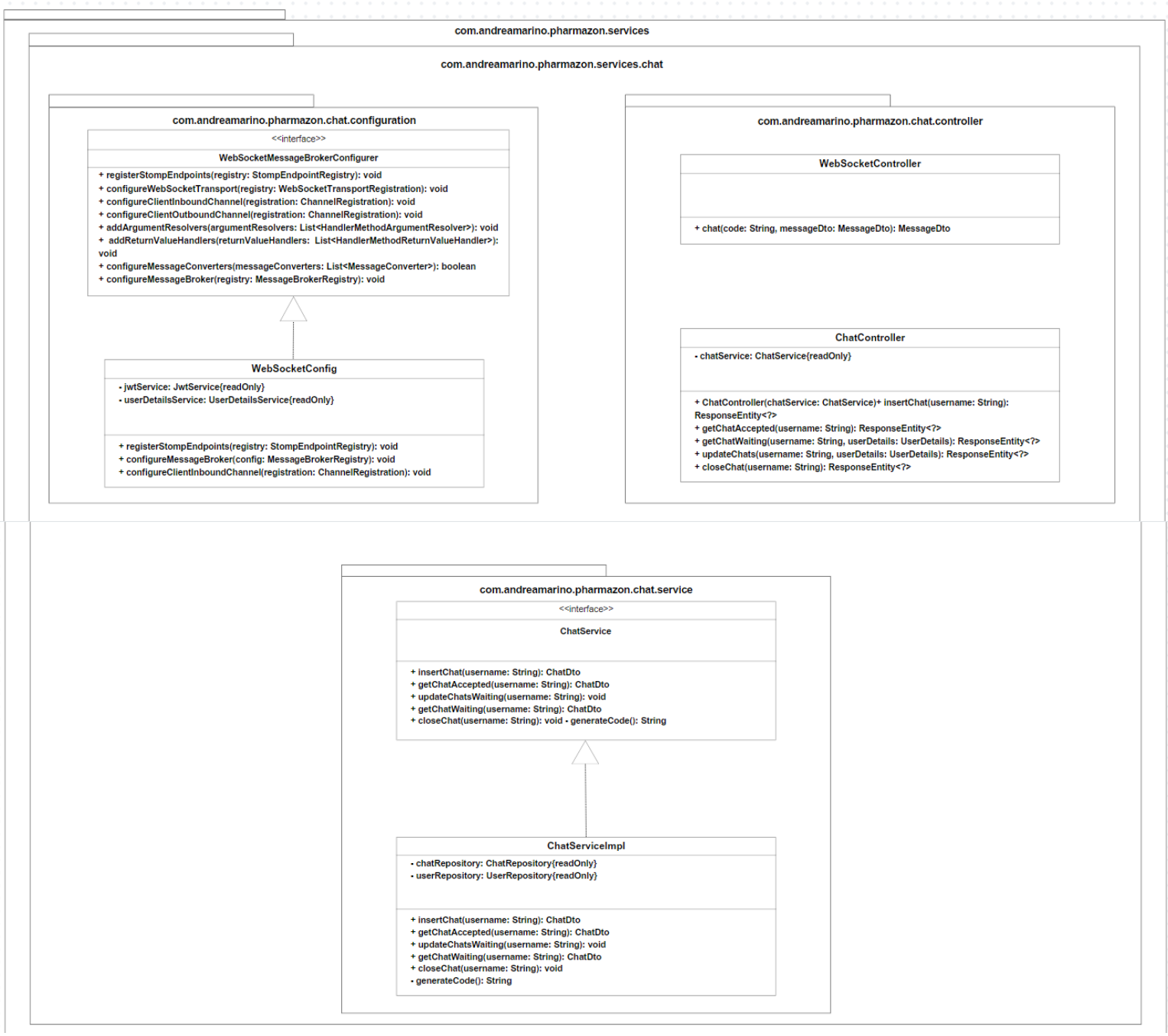
- @RestController, che indica come questa classe fornisca risposte HTTP, che vengono restituite come JSON o XML;
- @ControllerAdvice, consente alla classe di intercettare le eccezioni lanciate dai metodi dei controller, in tutta l'applicazione;
- @ExceptionHandler, che è possibile trovare annotato ad ogni metodo della classe, per indicare una gestione specifica di quella eccezione.

Qui riportato un esempio:

```
@ExceptionHandler(IllegalArgumentException.class)
public final ResponseEntity<ErrorResponse> illegalArgumentException(RuntimeException e){
    ErrorResponse error = new ErrorResponse(HttpStatus.BAD_REQUEST.value(), e.getMessage());
    return new ResponseEntity<ErrorResponse>(error, HttpStatus.BAD_REQUEST);
}
```

Inoltre, da notare la presenza del metodo handleHttpMessageNotReadable che consente di gestire le eccezioni sollevate, nel momento in cui il corpo della richiesta HTTP non è leggibile o manca.

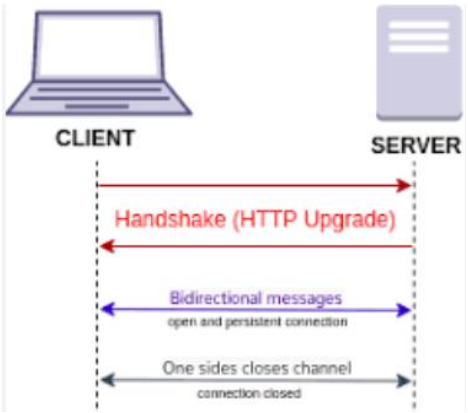
4.3.11 Services, Chat & E-mail



Il package *com.andreamarino.pharmazon.services* è stato creato per contenere tutti quei servizi che hanno bisogno di una implementazione diversa, dai comuni service, e che sfruttano dei **protocolli**.

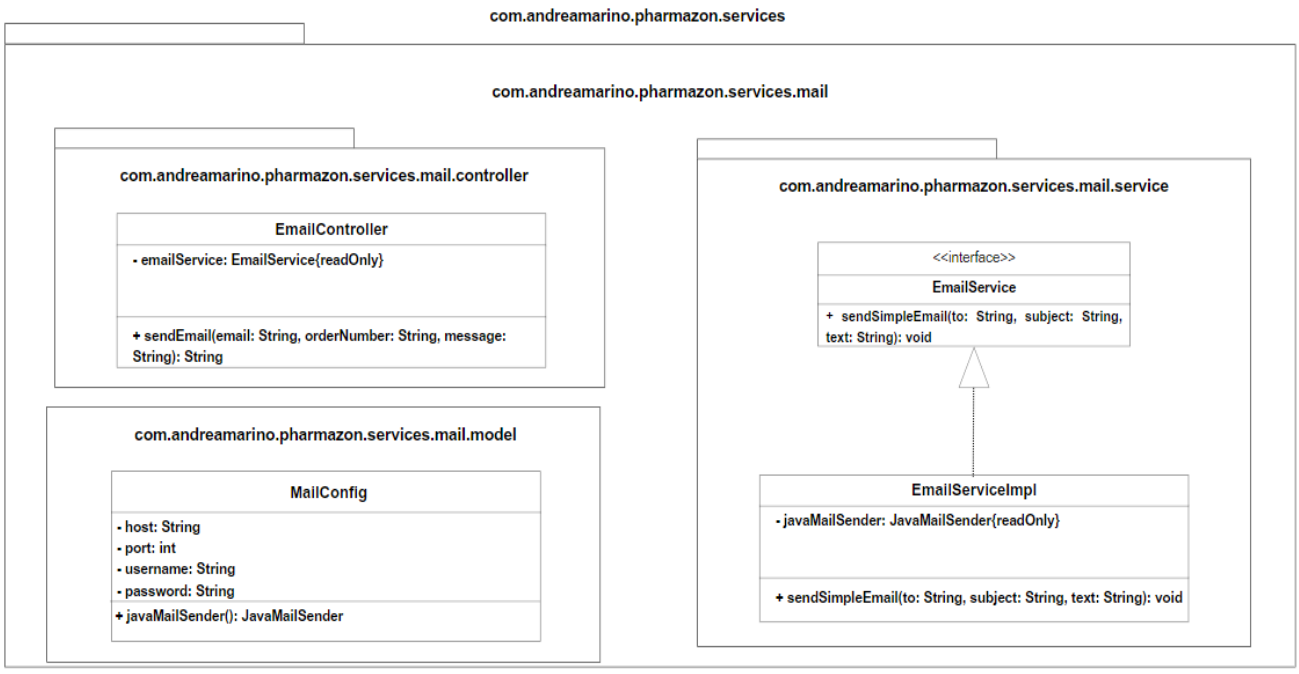
In questa immagine, viene riportato il class diagram relativo all'utilizzo della chat e, come è possibile notare, è presente una classe relativa alla gestione del **WebSocket**, cioè un protocollo di comunicazione bidirezionale full-duplex, che consente la comunicazione e lo scambio di dati contemporaneamente tra Client e Server su una singola connessione TCP, permettendo lo scambio di informazioni tra cliente e farmacista.

Per garantire un certo livello di sicurezza, viene implementato nella classe WebSocketConfig, il metodo *configureClientInboundChannel*, che consente di verificare l'autenticazione delle connessioni verso il WebSocket, effettuate dal Client tramite **WebSocket STOMP** (Simple Text Oriented Messaging Protocol, protocollo di messaggistica).



Viene verificato che il comando eseguito dal Client sia il **CONNECT** (prima di iniziare una nuova connessione), e che la richiesta di connessione verso il WebSocket, abbia all'interno dell'header un JWT valido, appartenente effettivamente ad un utente registrato alla nostra applicazione.

Inoltre, viene anche creata una entità Chat, con il relativo controller e service, per salvare all'interno del database alcune informazioni utili, come la data e ora della creazione della chat, e gli utenti facenti parte della conversazione creata.



All'interno di questo package, troviamo anche il sotto package email, riportato in questa ultima immagine, che contiene le informazioni relative all'utilizzo del servizio di email. Nello specifico andremo ad utilizzare il protocollo SMTP (Simple Mail Transfer Protocol): standard utilizzato per l'invio delle email.

È possibile individuare la classe MailConfig annotata con *@Configuration*, che contiene quattro campi:

- Host, che rappresenta l'indirizzo dell'host del server SMTP utilizzato per inviare le email;
- Port, rappresenta il numero di porta del server SMTP utilizzato per inviare le email;
- Username, rappresenta il nome utente utilizzato per l'autenticazione al server SMTP;
- Password, rappresenta la password utilizzata per l'autenticazione al server SMTP.

Questi campi sono tutti annotati con *@Value*, usato per iniettare i valori delle proprietà presenti nel file di configurazione di Spring, l'**application.properties**.

La classe inoltre contiene anche un metodo chiamato *javaMailSender* per creare una istanza di JavaMailSenderImpl, impostando l'host, la porta, lo username e la password, con i valori provenienti dal file application.properties e vengono impostate le proprietà specifiche per il protocollo SMTP tramite l'oggetto Properties.

Infine, la classe EmailServiceImpl ci consente di poter avere un metodo chiamato *sendSimpleEmail*, che permette di inviare una email ad un certo utente, con un certo oggetto e con un testo associato (corpo). Il metodo inoltre, è annotato con *@Async* che indica a Spring di eseguire questo metodo in modo asincrono su un thread separato, in modo da non dipendere dal processo di invio della email, per svolgere altre operazioni. In questo metodo è possibile trovare anche l'istanza SimpleMailMessage che rappresenta la nostra email, inviata tramite il javaMailSender.

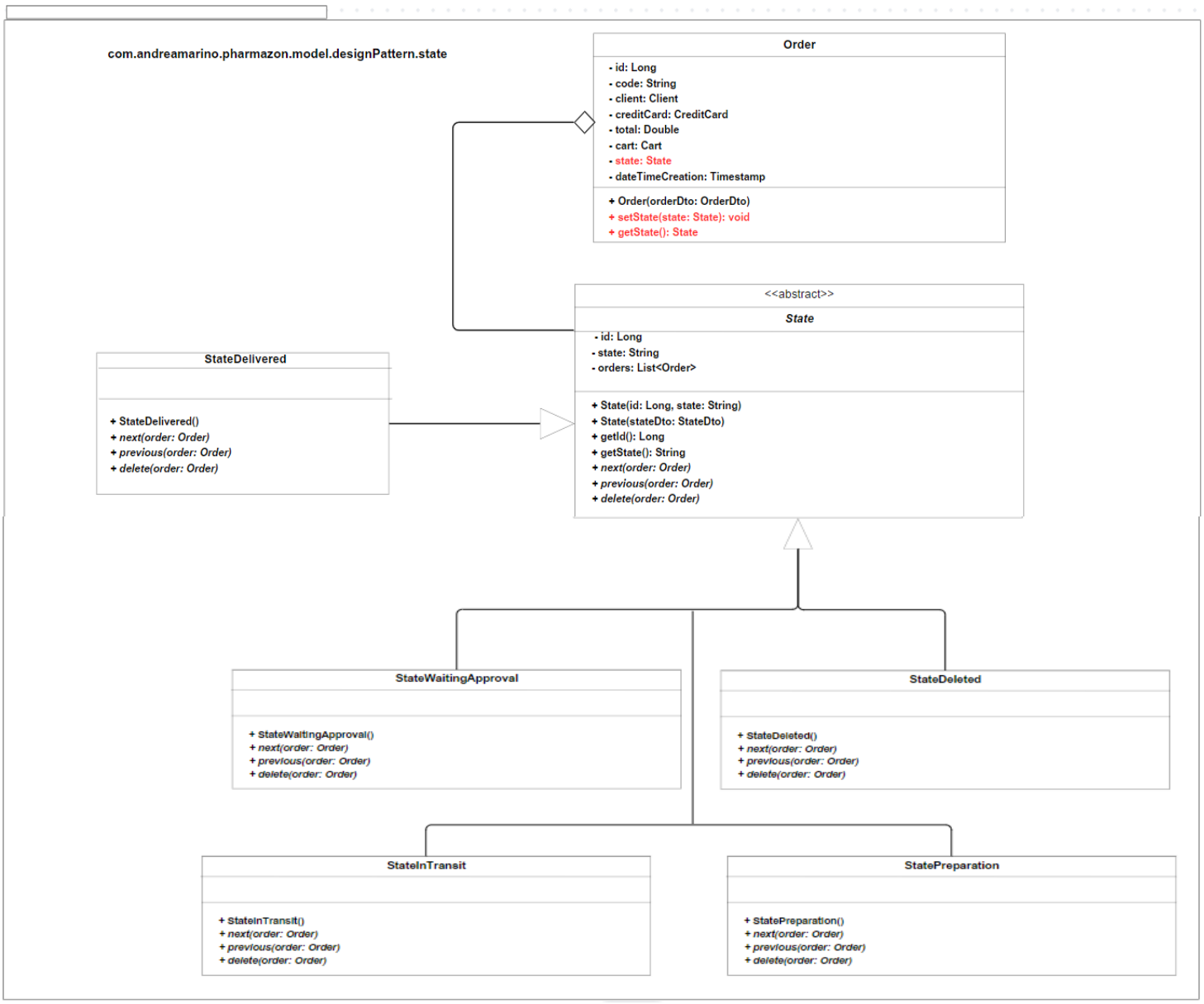
```
SimpleMailMessage message = new SimpleMailMessage();
message.setTo(to);
message.setSubject(subject);
message.setText(text);
javaMailSender.send(message);
```

4.4 Class Diagram- Design Pattern

Di seguito, viene riportato il package contenente al suo interno le classi e interfacce per la realizzazione dei design pattern: *com.andreamarino.pharmazon.model.designPattern*, che ha a sua volta una cartella per il pattern State e per il pattern Observer.

4.4.1 State

Obiettivo del pattern: Lo state pattern è progettato per permettere ad un oggetto di poter cambiare il proprio comportamento in base allo stato in cui si trova.



In questa ultima immagine, vediamo l'implementazione del pattern state, facente parte della cartella *state*. Per una questione di completezza e chiarezza, è stata inserita nella creazione del diagramma, anche la classe Order, che fa parte invece del package: *com.andreamarino.pharmazon.model*.

La classe principale, per la realizzazione del pattern è la classe astratta **State**, composta da attributi quali id e state (una stringa contenente il valore dello stato) e alcuni metodi importanti, implementati nelle classi concrete, come next, previous e delete, dove i primi due rispettivamente permettono all'ordine, di passare allo stato successivo, allo stato precedente e invece la funzione delete consente di passare allo stato di eliminazione.

Tale classe inoltre utilizza le annotazioni:

- @Inheritance(strategy = InheritanceType.SINGLE_TABLE), indica la strategia di ereditarietà che JPA utilizza per mappare la gerarchia di classi sulle tabelle del database, in questo caso SINGLE_TABLE che indica che tutte le classi nella gerarchia di ereditarietà saranno mappate su una singola tabella nella base di dati;
- @DiscriminatorColumn(name="entity_type", discriminatorType = DiscriminatorType.STRING), indica la colonna discriminante per distinguere i diversi tipi di entità, in questo essendo DiscriminatorType.STRING, ciò indica che il valore discriminante è una stringa.

La classe astratta State, è estesa dalle seguenti classi concrete:

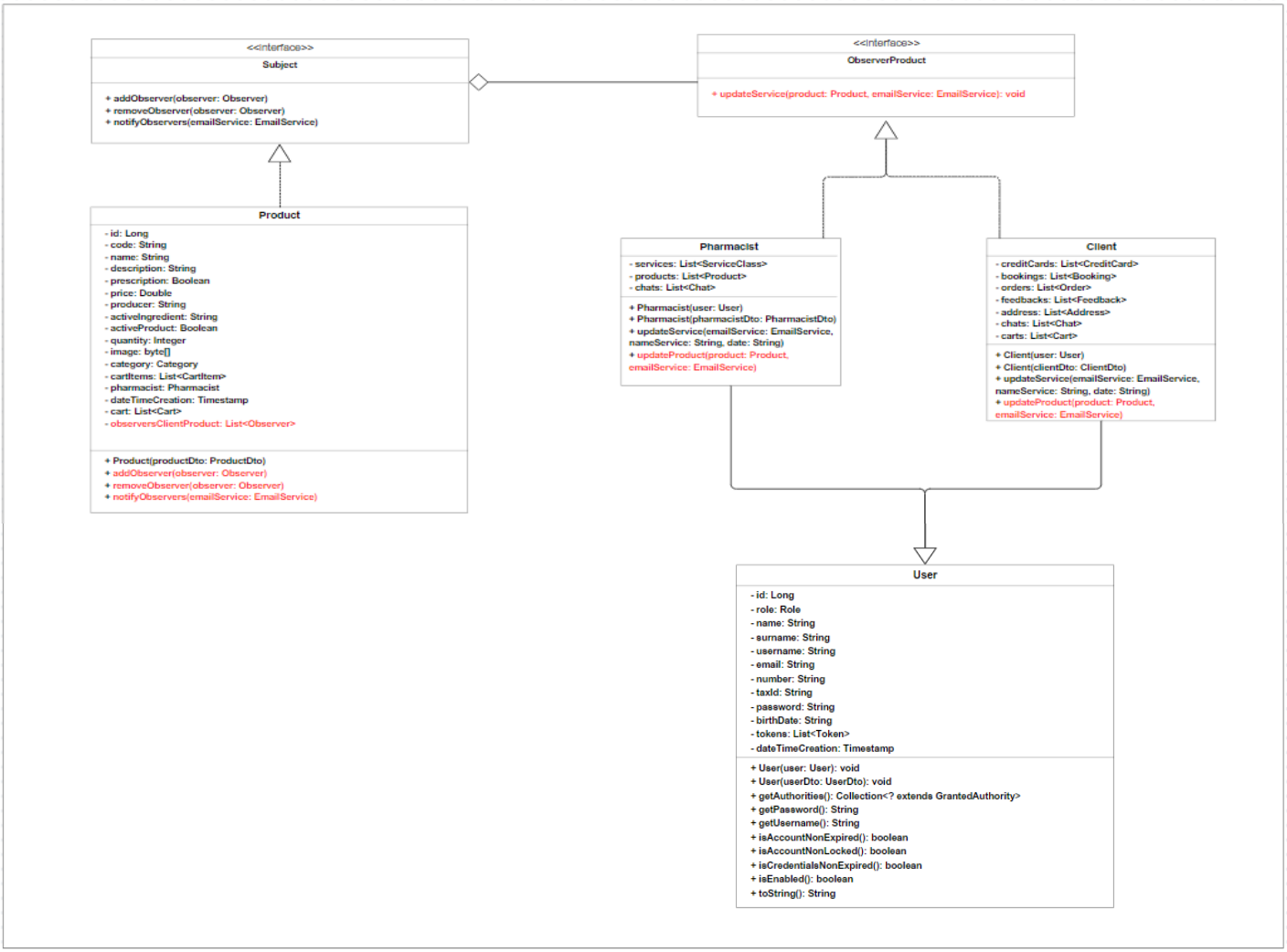
- StateWaitingApproval, la classe che indica lo stato "IN ATTESA DI APPROVAZIONE";
- StatePreparation, la classe che indica lo stato "IN PREPARAZIONE";
- StateInTransit, la classe che indica lo stato "IN TRANSITO";
- StateDelivered, la classe che indica lo stato "CONSEGNATO";
- StateDeleted, la classe che indica lo stato "ELIMINATO".

Tutte queste classi implementano l'annotation @DescriminatorValue che specifica il valore discriminatore per una particolare entità in una gerarchia di ereditarietà.

4.4.2 Observer

Obiettivo del pattern: Il pattern observer è progettato per consentire ad un oggetto, chiamato Subject, di notificare automaticamente ad una serie di oggetti, cioè gli Observer, il cambiamento del proprio stato. È situato nella cartella observer e a sua volta è possibile trovare due cartelle legate all'implementazione del pattern, sia per i servizi che per i prodotti.

Prodotti



In questa immagine invece, è possibile vedere l'implementazione del pattern observer specificatamente per i prodotti. Per una questione di completezza e chiarezza, sono stati inseriti nella creazione del diagramma anche Product, User, Client e Pharmacist facenti parte del package: *com.andreamarino.pharmazon.model.*

È utile notare come sia stata utilizzata l'**ereditarietà** con la class User (classe padre) nei confronti delle classi figlie e cioè Client e Pharmacist, permettendo di ereditare alcuni attributi e metodi e poter averne dei propri definiti nelle singole classi.

L'obiettivo è quello di notificare al cliente, che un prodotto acquistato in passato è prossimo a terminare (quantità uguale a 2) o è terminato. Inoltre, l'ordine che ha effettuato

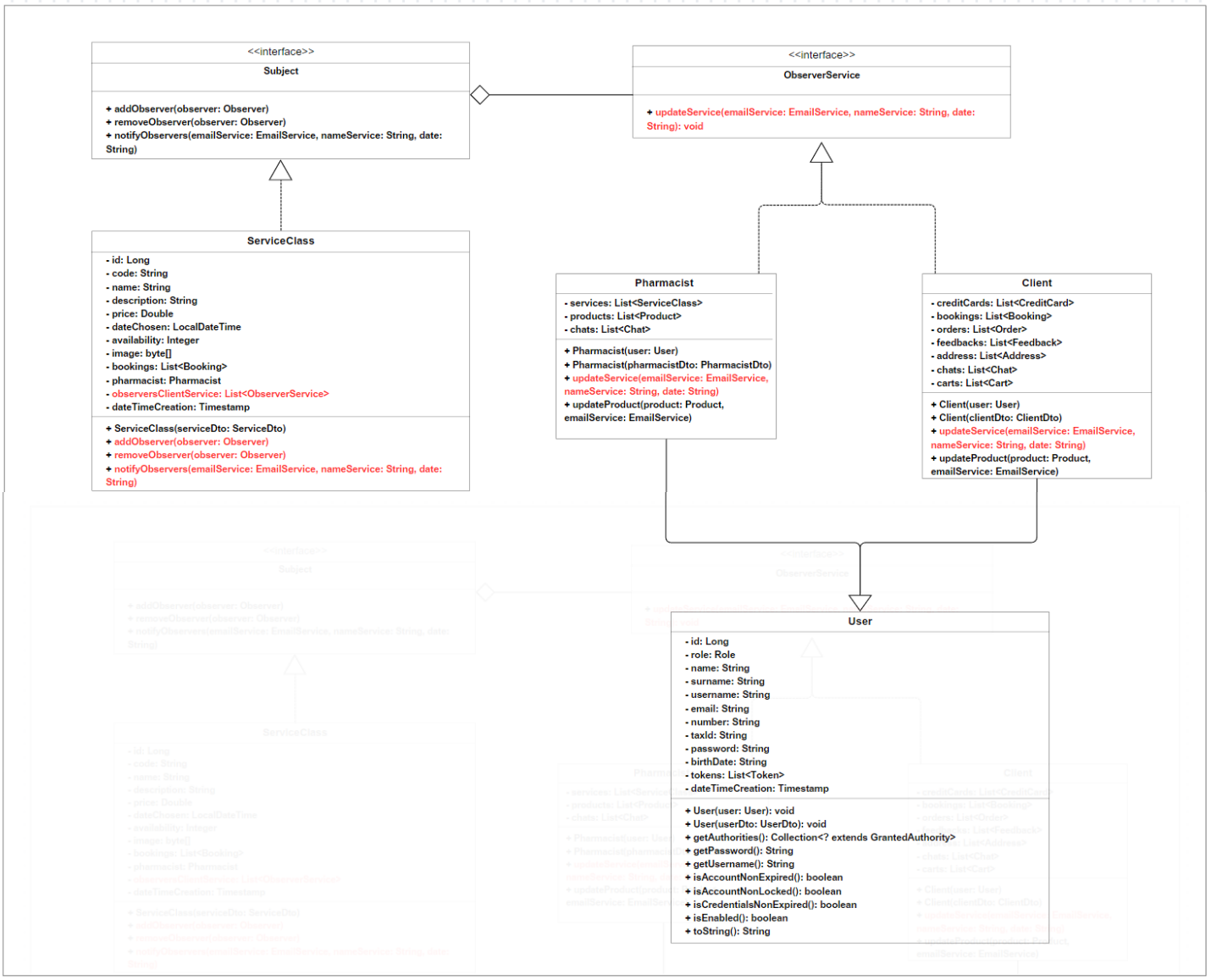
contenente tale prodotto, deve essere in stato di CONSEGNATO, affinché il cliente possa essere notificato. Questo controllo della quantità e notifica verso l'utente, avviene quando è modificato il prodotto (solo se inserita una quantità diversa, da parte del farmacista) o viene creato un nuovo ordine contenente quel prodotto (e di conseguenza viene modificata la sua quantità) o viene approvato un ordine. Il farmacista viene anche egli notificato, nelle stesse situazioni del cliente, ed il farmacista ad essere notificato, è l'ultimo che ha modificato il prodotto.

Per mettere in pratica il pattern, la classe Product implementa l'interfaccia Subject, implementandone i metodi:

- addObserver, che consente di inserire un nuovo utente alla lista degli osservatori;
- removeObserver, che consente di rimuovere un utente dalla lista degli osservatori;
- notifyObserver, che consente di notificare gli osservatori in caso di eventuali aggiornamenti.

Invece la classe Client e Pharmacist, andranno ad implementare l'interfaccia ObserverProduct, implementandone il metodo updateProduct, che consente di notificare l'utente tramite la sua email, dell'aggiornamento di quantità del prodotto.

Servizi



Nell'ultima immagine, della pagina precedente, è possibile vedere l'implementazione del pattern observer specificatamente per i servizi. Per una questione di completezza e chiarezza, sono stati inseriti nella immagine anche ServiceClass, User, Client e Pharmacist facenti parte del package: *com.andreamarino.pharmazon.model.*

L'obiettivo è quello di notificare il cliente, che è prenotato ad un certo servizio, che il servizio a cui si era prenotato, non è più disponibile. La notifica verso l'utente avviene quando è effettuata l'eliminazione del servizio (solo quando è ancora valido, cioè la data in cui si dovrà effettuare tale prestazione non è stata superata) o una modifica delle sue informazioni e perciò verrà eliminata anche la prenotazione a tale servizio, dato che la modifica implica dei cambiamenti al servizio, che l'utente non ha approvato al momento della prenotazione. Un esempio potrebbe essere la modifica del nome del servizio da "Test glicemia" a "Test celiachia".

Inoltre, verrà notificato anche il farmacista che ha creato tale servizio, solo quando il servizio viene cancellato e questo è ancora valido (la data in cui si deve effettuare tale prestazione, non è stata superata), in modo da poterlo comunicare anche con eventuali clienti che non utilizzano la piattaforma. È utile notare che nel caso in cui il servizio venga modificato, il farmacista che verrà notificato, sarà l'ultimo che ha effettuato la modifica.

Per mettere in pratica il pattern, la classe ServiceClass implementa l'interfaccia Subject, implementandone i metodi:

- addObserver, che consente di inserire un nuovo utente alla lista degli osservatori;
- removeObserver, che consente di rimuovere un utente dalla lista degli osservatori;
- notifyObserver, che consente di notificare gli osservatori in caso di eventuali aggiornamenti.

Invece la classe Client e Pharmacist, andranno ad implementare l'interfaccia ObserverService, implementandone il metodo updateService, consentendo di notificare l'utente tramite la sua email, dell'aggiornamento di stato del servizio.

5. Test

Una fase fondamentale durante la realizzazione dell'applicativo è la fase di testing. Tale fase, infatti, ci permette di verificare eventuali problematiche relative al codice. Nello specifico sono stati effettuati gli Unit Test, utilizzando il framework JUnit, che ci consente di testare in modo efficiente il codice da noi scritto e di trovare eventuali anomalie di comportamento. Il numero di test effettuati affinché si potesse studiare in maniera completa e isolata, il comportamento delle singole classi dei vari package e nello specifico dei loro metodi, è di 892 test.

5.1 Configurazione dei Test

Qui di seguito vengono riportate delle annotazioni che è comune ritrovare nelle classi di test:

- @ExtendWith(MockitoExtension.class), utilizzata per abilitare l'integrazione di Mockito con JUnit 5;
- @Mock, utilizzata per creare oggetti di mock nei nostri test;
- @InjectMocks, utilizzata per creare una istanza della classe da testare;
- @Test, indica che il metodo è un metodo di test.

Inoltre, è possibile trovate anche alcuni dei seguenti metodi statici:

- when, utilizzato per specificare il comportamento dei metodi mock:
- doNothing, utilizzato per configurare i metodi void, in modo che non facciano nulla quando sono chiamati;
- verify, usato per verificare che un determinato metodo del mock sia chiamato con argomenti specifici;
- assertEquals, metodo utilizzato che due valori siano uguali;
- assertThrows, metodo utilizzato per verificare che una determinata esecuzione del codice, lanci una specifica eccezione.

Per la **nomenclatura** dei metodi è stata compiuta questa scelta:

NomeMetodoDaTestare_ComportamentoAttuale_ComportamentoAtteso

```
@Test
public void getAddress_WhenValidInput_ReturnedListEmpty(){
    //Setup
    List<Address> listAddress = new ArrayList<>();
    String username = "andrysea";

    //Mock
    when(addressRepository.findAllByUsername(anyString())).thenReturn(listAddress);

    //Test
    assertThrows(expectedType:NoSuchElementException.class, () ->
        addressServiceImpl.getAddressDto(username));

    verify(addressRepository, times(wantedNumberOfInvocations:1)).findAllByUsername(username);

}
```

5.1 Code Coverage (Copertura del codice)

Qui di seguito riportiamo, in questa immagine a seguire, la code coverage dei singoli package dell'applicativo, ed è possibile notare le seguenti caratteristiche:

- Copertura delle classi, **92%**;
- Copertura dei metodi, **94%**;
- Copertura delle linee, **92%**;
- Copertura dei rami; **92%**.

Element	Class, %	Method, %	Line, %	Branch, %
com.andreamarino.pharmazon	92% (73/79)	94% (270/287)	91% (1873/2041)	92% (967/1048)
configuration	100% (2/2)	100% (13/13)	100% (24/24)	100% (2/2)
controller	100% (11/11)	100% (54/54)	100% (134/134)	100% (6/6)
dto	100% (14/14)	100% (14/14)	100% (82/82)	100% (0/0)
exception	100% (3/3)	91% (11/12)	90% (20/22)	100% (0/0)
model	100% (19/19)	98% (57/58)	99% (219/221)	97% (35/36)
repository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
security	71% (5/7)	76% (19/25)	50% (111/222)	65% (47/72)
service	100% (11/11)	100% (71/71)	99% (1059/1069)	95% (707/738)
services	70% (7/10)	69% (16/23)	74% (94/126)	85% (60/70)
util	100% (1/1)	100% (15/15)	97% (130/133)	88% (110/124)

Attenendoci alla copertura delle linee, la percentuale di coverage del progetto è del **92%**.

Di seguito viene riportata una lista di alcuni test che sono stati effettuati:

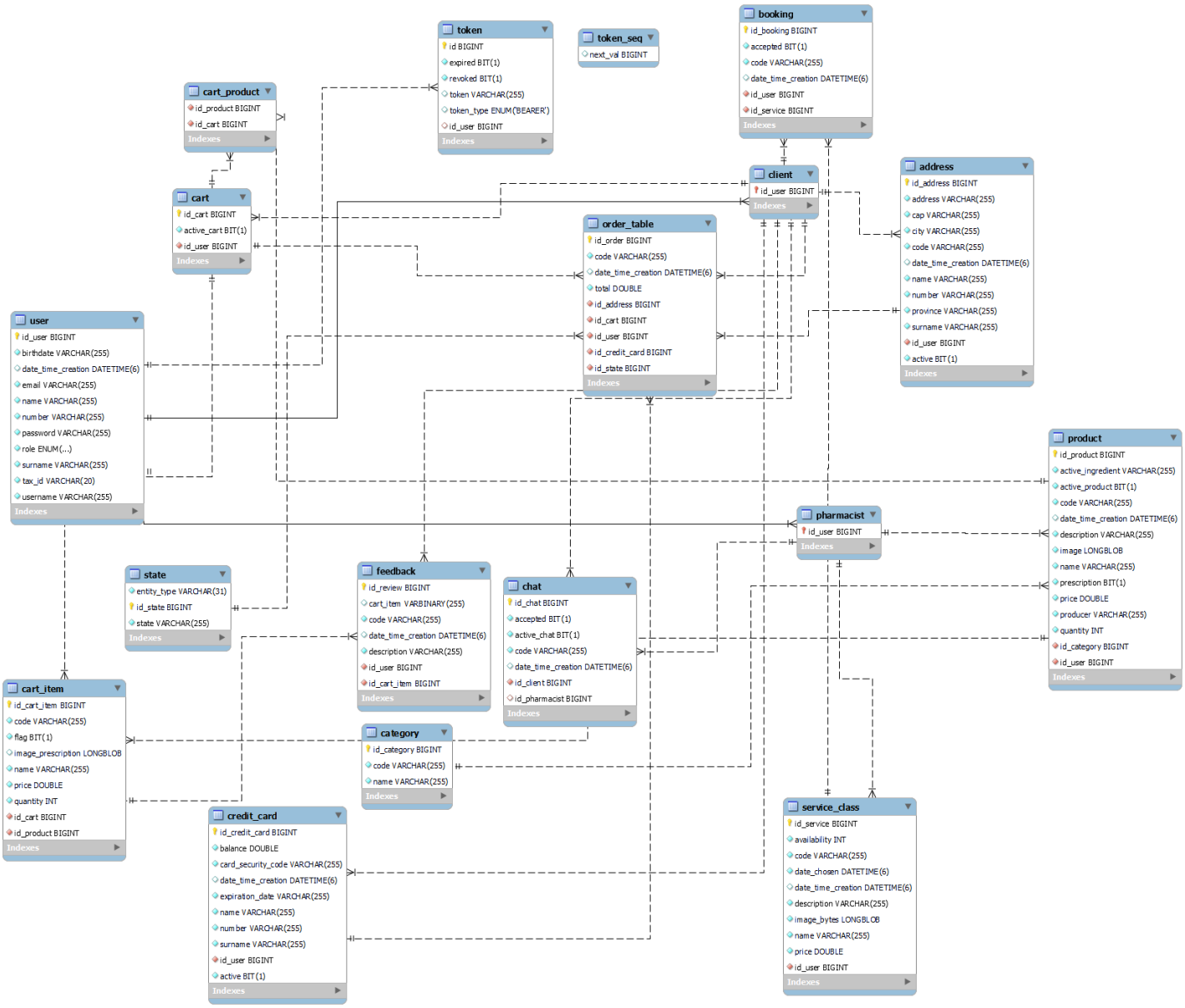
- insertOrder_WhenValidInputWithPrescription_ReturnObject(), ci permette di testare il metodo insertOrder, quando si inserisce un prodotto con prescrizione all'interno dell'ordine e quando l'esecuzione del metodo va a buon fine;
- insertProductDto_WhenValidInputCategoryNotFound_NotFoundException(), ci permette di testare il metodo insertProductDto, verificando che venga sollevata una eccezione quando voglio inserire un nuovo prodotto nel catalogo, con una categoria non presente nel database;
- insertProductDto_WhenValidInputAndProductNoInCart_ReturnedObjectInsert(), ci permette di testare il metodo insertProductDto nel contesto di inserimento del prodotto nel carrello dell'utente, aspettandoci che il test vada a buon fine, quando il prodotto non è nel carello e ci sono dei valori validi associati ad esso;
- updateServiceDto_WhenValidInput_ReturnObject(), ci permette di testare il metodo updateServiceDto, verificando che la modifica dell'oggetto service, vada a buon fine.

La parte relativa al funzionamento dei WebSocket, è stata testata manualmente e non tramite unit test.

6. Database

6.1 Diagramma E-R

Il database impiegato, per lo sviluppo del progetto è di tipo relazionale, e di seguito viene riportato il suo diagramma entità – relazione:



Il diagramma Entità-Relazione (E-R) è uno strumento di modellazione dei dati utilizzato per rappresentare il modo in cui le entità in un sistema sono correlate tra loro. È composto da entità, relazioni e attributi.

Prospettive future

Il sistema attuale permette di poter mettere a disposizione dell'utente finale, sia per il Farmacista che per il Cliente, tutte le funzionalità che sono state descritte in questo documento.

Qui di seguito, vengono riportate alcune idee di aggiornamenti ed implementazioni future da attuare nell'applicativo:

- Inserimento dell'utente visitatore, per garantire la possibilità anche agli utenti non registrati di poter visionare i prodotti e i servizi;
- Inserimento di una chat con bot, in caso in cui il farmacista non fosse disponile;
- Notificare i clienti di eventuali sconti e promozioni;
- Utilizzo di un certificato CA (Certificate Authority), rilasciato da una entità di fiducia, sostituendolo con il certificato attuale self-signed;
- Criptaggio dei dati nel Database.