TheKnife

Manuale Tecnico

Lorenzo Radice
Laurea Triennale in Informatica, Università degli Studi dell'Insubria
Matricola: 753252 — Sede Como

Anno Accademico 2024/2025

Indice

1 Progettazione

1.1 Use-Case Diagram

Per ciascuna tipologia di utente (ospite, cliente, ristoratore) sono stati individuati i casi d'uso principali, dalla ricerca di ristoranti alla gestione di recensioni e risposte. É stato redatto uno Use-Case Diagram che rappresentasse le principali interazioni tra gli utenti e il sistema.

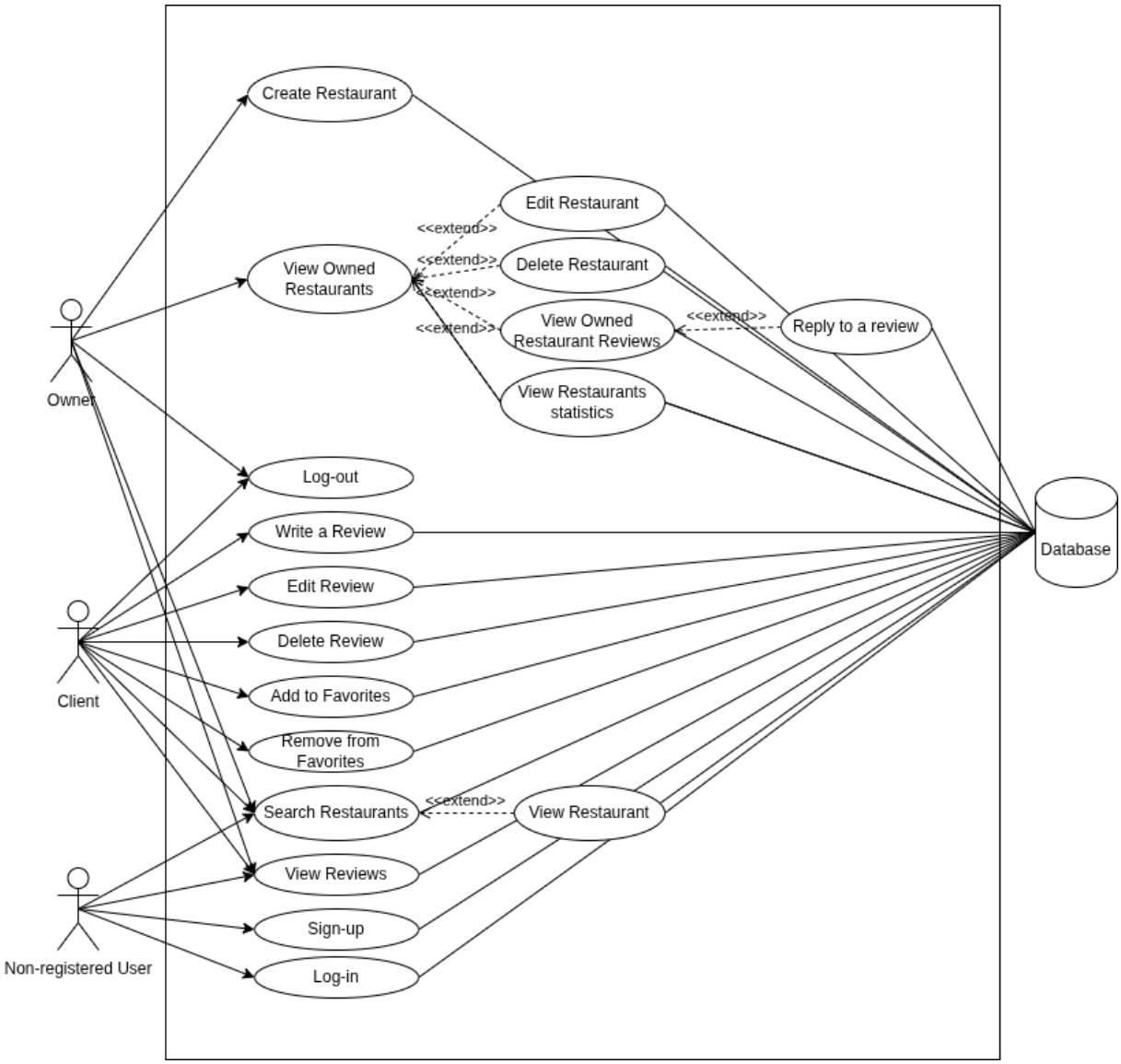

Figura 1: User-Case Diagram

Basandosi sul diagramma si è poi proceduto ad articolare le singole funzionalità in modo da poterle implementare in maniera modulare attraverso la Progettazione Orientata agli Oggetti.

1.2 Class Diagram

Al fine di rappresentare le classi e le loro relazioni, dallo Use-Case Diagram è stato sviluppato un Class Diagram che potesse racchiudere le classi principali del sistema.

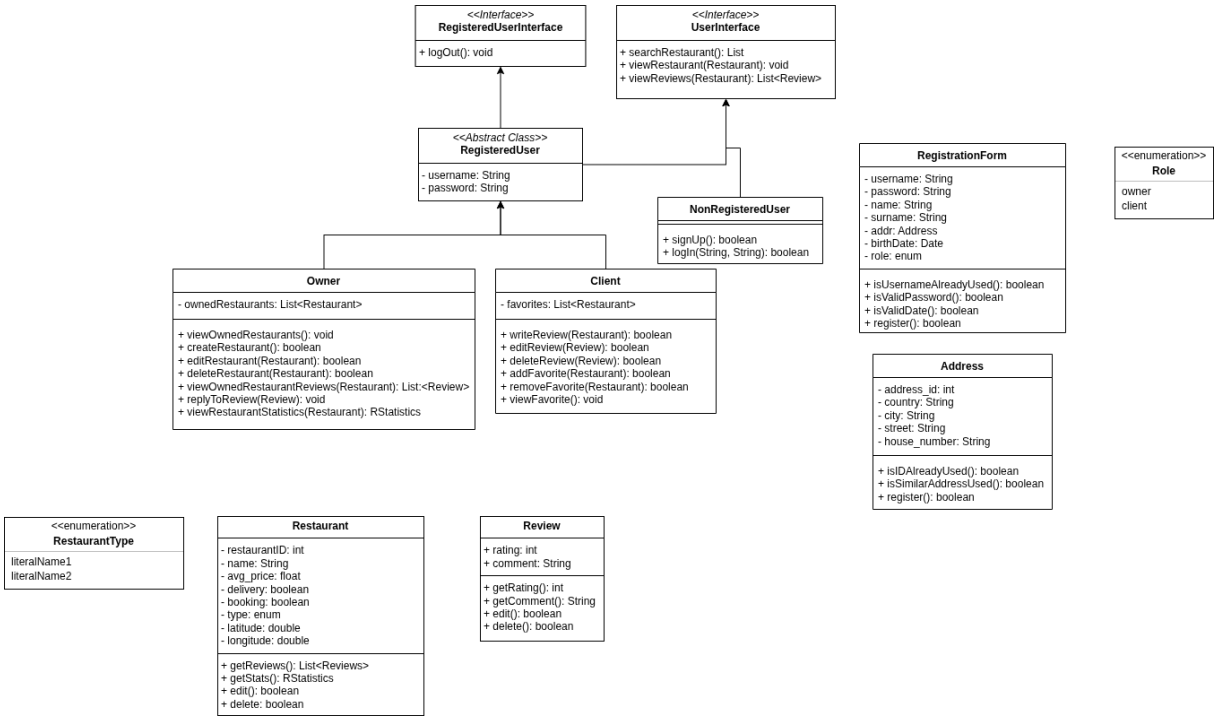

Figura 2: Class Diagram

Il diagramma è stato poi utilizzato come spunto per implementare le classi in Java, seguendo le relazioni e le associazioni principali.
In particolare, le classi e i metodi principali sono stati inseriti nel modulo *common* e utilizzati come base fondante del progetto per una corretta interazione tra client e server.

1.3 Architettura del Sistema

Il sistema è basato su un'architettura Client-Server.
Si è scelto di utilizzare Java RMI per la comunicazione tra client e server, in modo da poter sfruttare le funzionalità di Remote Method Invocation e garantire una comunicazione semplice ed efficace. Il server espone i servizi attraverso un Registry RMI sulla porta `1099`, permettendo ai client di accedere ai metodi remoti. Il client, a sua volta, si connette al server per invocare i metodi e ricevere le risposte.

É stato fondamentale definire delle interfacce comuni a Client e Server, in modo da garantire una comunicazione coerente e facilitare l'implementazione delle funzionalità. Per questa ragione il progetto è suddiviso in tre moduli principali:

- **client**: implementa il client e la sua interfaccia grafica
- **common**: contiene le classi e le interfacce comuni per il trasferimento dei dati tra client e server
- **server**: implementa il server e gestisce la connessione con il database

1.4 Flusso delle Informazioni

Nel modulo *common* sono definite le interfacce comuni e le classi serializzabili definite con il pattern Data Transfer Object (DTO), che permettono di trasferire i dati tra client e server in modo strutturato.
Il trasferimento di informazione avviene attraverso le interfacce definite nel package *common* implementate nel server e accessibili al client tramite il Registry. I DTO serializzabili possono essere passati come argomento ai metodi invocati. L'elaborazione viene presa in carico dagli oggetti Data Access Object (DAO) presenti sul server che gestiscono l'interazione con il database.
Il server, una volta ricevuti i dati, li elabora e restituisce le risposte al client, che le visualizza nell'interfaccia grafica.
Questa architettura consente di mantenere una chiara separazione tra le responsabilità del client e del server, facilitando la manutenzione e la semplicità del sistema.

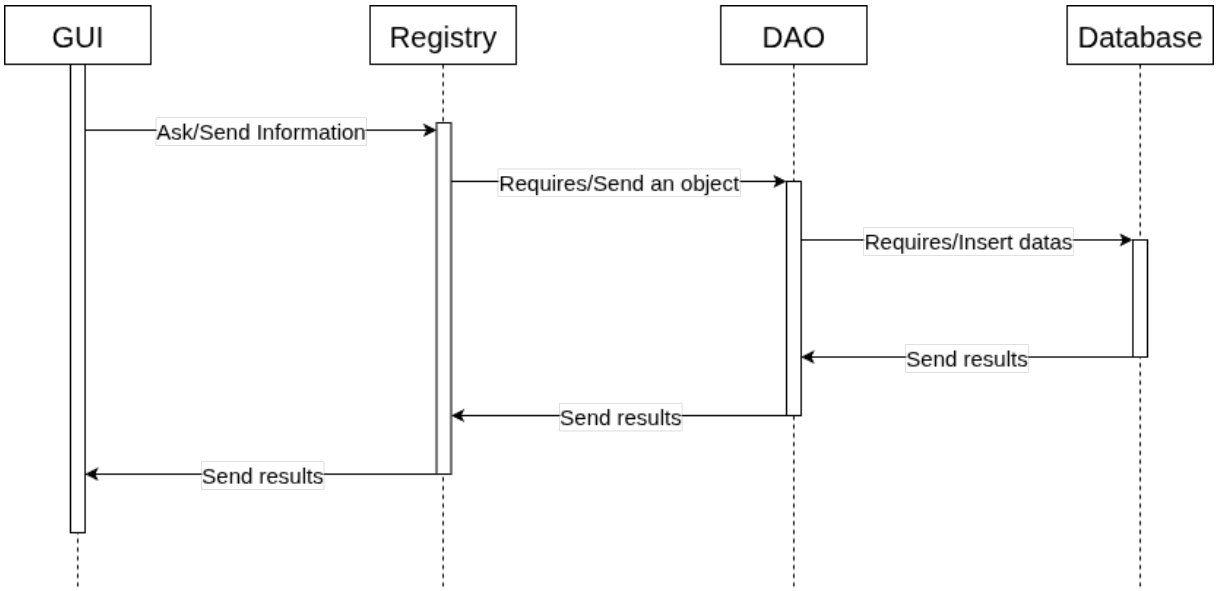

Figura 3: Interaction Diagram - Client-Server Communication

2 Database

Si è deciso di utilizzare un database relazionale per la gestione dei dati dell'applicazione. Come database è stato scelto PostgreSQL.

2.1 Schema del Database

2.1.1 Diagramma Entità-Relazione

Il database è stato progettato realizzando un diagramma Entità-Relazione (Figura 4) che racchiude i dati che si ritiene necessario gestire.

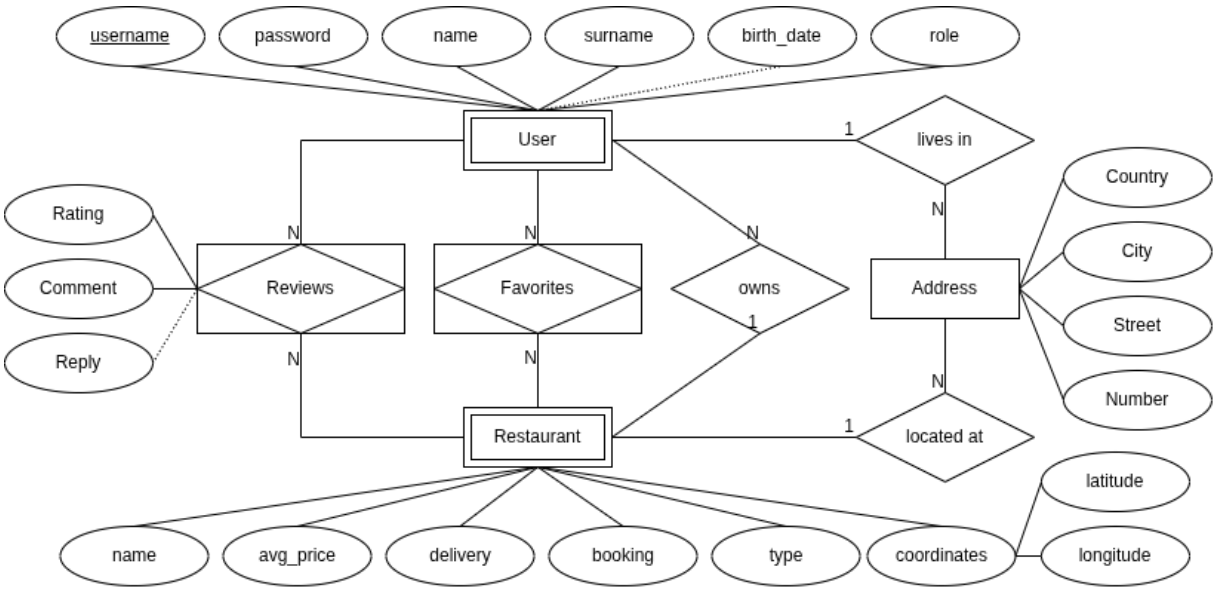

Figura 4: Diagramma E-R

Come si evince dal diagramma l'entità principale è *Address* che contiene i dati relativi agli indirizzi. *User* e *Restaurant* sono in relazione con *Address* e a loro volta sono in relazione tra loro. *Review* e *Favorite* sono entità che mettono in relazione *User* e *Restaurant* per gestire le recensioni e i ristoranti preferiti dagli utenti.

2.1.2 Rielaborazione del Diagramma

Successivamente è stata eseguita una rielaborazione del diagramma (Figura 5) per normalizzare le tabelle.

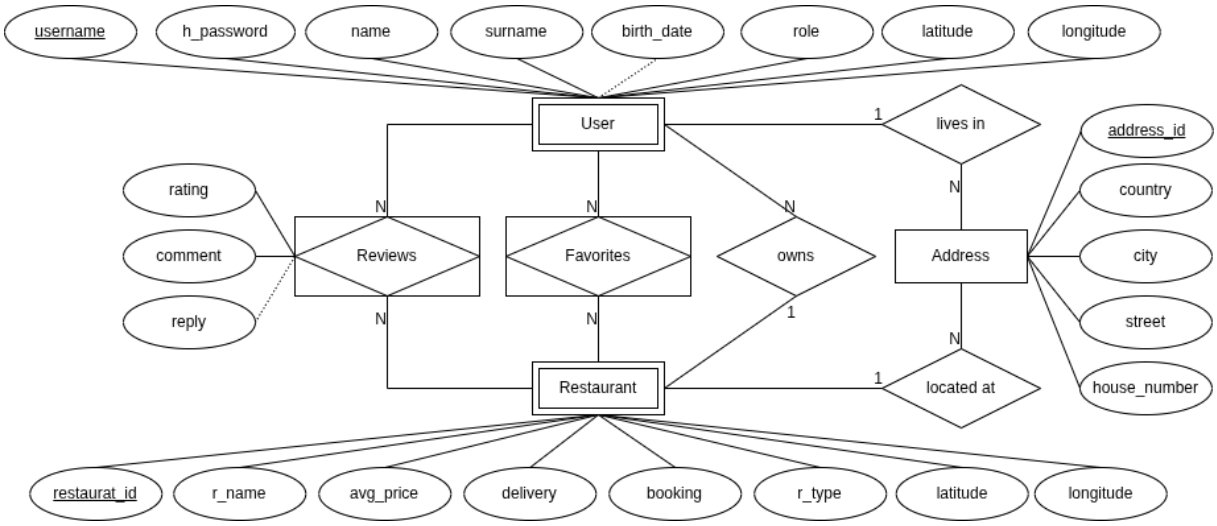

Figura 5: Diagramma E-R Normalizzato

Dopo questa rielaborazione le coordinate da attributo derivato sono diventati attributi semplici. Tuttavia, nell'implementazione finale, si è deciso di spostare le coordinate nell'entità *Address* per una migliore gestione. É infatti utile poter calcolare la distanza tra un *Restaurant* e un *User* per mostrare i ristoranti più vicini all'utente. Attribuendo questo campo ad entrambe le entità si è potuto gestire in modo efficace questo calcolo evitando ridondanze e ulteriori queries al database.

2.1.3 Schema

Infine, si sono elencate le tabelle e i relativi attributi per avere una visione immediata delle tabelle che compongono il database.

- **addresses** (address_id, country, city, street, house_number, latitude, longitude)
- **users** (username, h_password, name, surname, birth_date, role, address_id$^{\text{addresses}}$)
- **restaurants** (restaurant_id, r_owner, r_name, avg_price, delivery, booking, r_type, address_id$^{\text{addresses}}$)
- **favorites** (username$^{\text{users}}$, restaurant_id$^{\text{restaurants}}$)
- **reviews** (username$^{\text{users}}$, restaurant_id$^{\text{restaurants}}$, rating, comment, reply)

Questa lista di tabelle è presente nel progetto nel file *TheKnife/server/src/main/resources/db/Structure.md* con link diretti alle relative tabelle per poter consultare agevolmente le singole tabelle e la loro implementazione.

2.2 Tabelle

2.2.1 Addresses

La tabella *addresses* contiene gli indirizzi degli utenti e dei ristoranti.
Gli attributi sono:

- **address_id**: identificativo univoco dell'indirizzo
- **country**: nazione
- **city**: città
- **street**: via
- **house_number**: numero civico
- **latitude**: latitudine
- **longitude**: longitudine

Sono stati posti dei vincoli di integrità per garantire che le coordinate siano valide, nello specifico che la latitudine sia compresa tra -90 e 90 e la longitudine tra -180 e 180.

2.2.2 Users

La tabella *users* contiene gli utenti registrati nell'applicazione.
Gli attributi sono:

- **username**: nome utente univoco
- **h_password**: password cifrata con algoritmo di hashing
- **name**: nome
- **surname**: cognome
- **birth_date**: data di nascita
- **role**: ruolo dell'utente (owner o client)
- **address_id**: identificativo dell'indirizzo dell'utente

Il campo *role* è stato implementato creando un tipo di dato enumerativo che può assumere i valori *owner* o *client*.
Il campo *h_password* prevede che la password venga memorizzata in forma di hash per garantire la sicurezza. Si è scelto di utilizzare l'algoritmo di hashing Argon2 suggerito da OWASP.
Il campo *address_id* è una chiave esterna che fa riferimento alla tabella *addresses*. In caso di modifica dell'indirizzo di un utente, si aggiornerà il campo *address_id*, in caso di cancellazione dell'indirizzo non verrà cancellato l'utente.

2.2.3 Restaurants

La tabella *restaurants* contiene i ristoranti creati dai ristoratori.
Gli attributi sono:

- **restaurant_id**: codice univoco del ristorante
- **r_owner**: proprietario del ristorante (username dell'utente)
- **r_name**: nome del ristorante
- **avg_price**: prezzo medio del ristorante inserito dal ristoratore
- **delivery**: servizio di consegna disponibile (booleano)
- **booking**: servizio di prenotazione disponibile (booleano)
- **r_type**: tipo di cucina del ristorante (es. cinese, italiano, etc.)
- **address_id**: identificativo dell'indirizzo del ristorante

Il campo *r_owner* è una chiave esterna che fa riferimento alla tabella *users* e rappresenta il proprietario del ristorante. In caso di modifica di *username* dell'utente, si aggiornerà il campo *r_owner*, in caso di cancellazione dell'utente verrà cancellato anche il ristorante.
Il campo *address_id* è una chiave esterna che fa riferimento alla tabella *addresses*. In caso di modifica dell'indirizzo del ristorante, si aggiornerà il campo *address_id*, in caso di cancellazione dell'indirizzo non verrà cancellato il ristorante.
È stato posto un vincolo di integrità per garantire che il prezzo medio sia un numero positivo.

2.2.4 Favorites

La tabella *favorites* server a gestire i ristoranti preferiti dagli utenti.
Gli attributi sono:

- **username**: identificativo dell'utente
- **restaurant_id**: identificativo del ristorante

Il campo *username* è una chiave esterna che fa riferimento alla tabella *users* e il campo *restaurant_id* è una chiave esterna che fa riferimento alla tabella *restaurants*. In caso di modifica dei due campi si aggiorneranno i rispettivi campi, in caso di cancellazione di un utente o di un ristorante, verrà cancellata la preferenza.

2.2.5 Reviews

La tabella *reviews* serve a gestire le recensioni che gli utenti possono lasciare ai ristoranti.
Gli attributi sono:

- **username**: identificativo dell'utente
- **restaurant_id**: identificativo del ristorante
- **rating**: valutazione da 1 a 5
- **comment**: commento della recensione
- **reply**: risposta del ristoratore alla recensione

Il campo *username* è una chiave esterna che fa riferimento alla tabella *users* e il campo *restaurant_id* è una chiave esterna che fa riferimento alla tabella *restaurants*. In caso di modifica dei due campi si aggiorneranno i rispettivi campi, in caso di cancellazione di un utente o di un ristorante, verrà cancellata la recensione.
Sul *rating* è stato posto un vincolo di integrità per garantire che sia un numero intero compreso tra 1 e 5.

3 Common

Il package `Common` contiene i files comuni al client e al server per permetterne l'interazione. È il package principale del progetto su cui si basa l'intera architettura del sistema applicativo.

3.1 Data Transfer Objects

Il design pattern *Data Transfer Object* (DTO) è utilizzato per trasferire dati tra il client e il server. I DTO sono classi che contengono solo i dati necessari per la comunicazione, senza logica di business. Questo approccio semplifica la serializzazione e deserializzazione dei dati durante le richieste poiché tutte le classi DTO implementano `Serializable`.
Le classi principali che utilizzano il design pattern corrispondono alle entità principali del database e sono le seguenti:

- `AddressDTO`
- `UserDTO`
- `RestaurantDTO`
- `ReviewDTO`

Come si può notare dalla lista, non è stato creato un DTO per l'entità *Favorite* poiché non si è ritenuto necessario creare una classe apposita per la sua gestione. È stata invece utilizzata nei metodi interni al server per le elaborazioni, ma non come oggetto trasferibile.
Si è ritenuto opportuno creare due tipi enumerabili per rappresentare i tipi di cucina e di utenti:

- `CuisineType`
- `UserRoleDTO`

Per facilitare la gestione della ricerca e i relativi filtri è stata creata una classe apposita chiamata `SearchCriteriaDTO`.

3.1.1 AddressDTO

La classe `AddressDTO` rappresenta un indirizzo e contiene campi analoghi a quelli della tabella *addresses* del database ad eccezione del campo *address_id*.

3.1.2 UserDTO

La classe `UserDTO` rappresenta un utente e contiene campi analoghi a quelli della tabella *users*.
L'indirizzo dell'utente è rappresentato da un campo *address* di tipo `AddressDTO` che durante l'inserimento nel database viene convertito nell'ID corrispondente e in fase di lettura nel corrispettivo oggetto.
L'attributo *role* è di tipo `UserRoleDTO` ed è gestito dalla classe enumerabile `UserRoleDTO`.

3.1.3 RestaurantDTO

La classe `RestaurantDTO` rappresenta un ristorante e contiene campi analoghi a quelli della tabella *restaurants*.
L'indirizzo dell'utente è rappresentato da un campo `address` di tipo `AddressDTO` che durante l'inserimento nel database viene convertito nell'ID corrispondente e in fase di lettura nel corrispettivo oggetto.
Sono presenti due campi aggiuntivi per fornire maggiori informazioni. I campi sono: `rating` e `reviewsNumber`. Essendo essi deducibili dal Database è necessario poterli trasferire al client e si è ritenuto opportuno farlo inserendoli nell'oggetto dalla cui entità corrispettiva si derivano.

3.1.4 ReviewDTO

La classe `ReviewDTO` rappresenta una recensione ad un ristorante e contiene campi analoghi a quelli della tabella *reviews*.
L'utente che ha scritto la recensione è rappresentato dal campo `username` di tipo `String` e il ristorante dal campo `restaurant_id` di tipo `String`. Non è stato ritenuto opportuno utilizzare dei tipo semplici piuttosto che i rispettivi DTO poiché si ritiene che essi siano sufficienti e che invece inserire ulteriori oggetti possa causare inutile overhead.

3.1.5 CuisineType

La classe `CuisineType` è un tipo enumerabile che rappresenta i 276 tipi di cucina disponibili nel sistema. I tipi di cucina sono predefiniti e sono gli medesimi sia per la classe `CuisineType` che per il database. Questo approccio consente di evitare errori di battitura e di garantire la coerenza tra applicazione e database.
I tipi di cucina sono rappresentati come stringhe e sono utilizzati per filtrare i ristoranti durante le ricerche.

3.1.6 UserRoleDTO

La classe `UserRoleDTO` è un tipo enumerabile che rappresenta i ruoli degli utenti nel sistema. I ruoli sono:

- `CLIENT`: cliente registrato
- `OWNER`: proprietario di un ristorante

I ruoli sono predefiniti e identici sia per l'applicazione Java che per il database.

3.1.7 SearchCriteriaDTO

La classe `SearchCriteriaDTO` rappresenta i criteri di ricerca per i ristoranti. I suoi campi rappresentano i filtri applicabili alla ricerca.

- `cuisineType`: tipologia di cucina
- `minPrice`: prezzo minimo in euro
- `maxPrice`: prezzo massimo in euro
- `deliveryAvailable`: disponibilità del servizio di consegna
- `onlineBookingAvailable`: disponibilità della prenotazione
- `minRating`: valutazione minima (scala 1-5)
- `latitude`: latitudine dell'utente
- `longitude`: longitudine dell'utente

Tutti i campi sono opzionali ad eccezione delle coordinate, necessarie a ordinare i ristoranti in base alla distanza dall'utente tramite la *Formula di Haversine*. Per la creazione dell'oggetto `SearchCriteriaDTO` è stato utilizzato il design pattern *Builder* per facilitarne la creazione e la gestione dei criteri. Il pattern *Builder* consente di creare oggetti complessi in modo flessibile e leggibile, permettendo di impostare solo i campi desiderati senza dover gestire costruttori con molti parametri. Questo approccio permette flessibilità senza compromettere la leggibilità del codice, facilitando la creazione dell'oggetto `SearchCriteriaDTO`.

3.2 Services

I servici che il server fornisce al client sono definiti nel package `Services`. Questi servizi sono implementati come interfacce che definiscono i metodi disponibili per il client implementati dal server.

3.2.1 UserService

Il servizio `UserService` i principali metodi per la gestione degli utenti, ovvero registrazione attraverso il metodo `register` e login attraverso il metodo `login`.

3.2.2 RestaurantService

Il servizio `RestaurantService` fornisce metodi per l'interazione coi ristoranti. I metodi definiti sono:

- `searchRestaurants`: per cercare ristoranti in base ai criteri di ricerca
- `getFavoriteRestaurants`: per ottenere i ristoranti preferiti di un utente
- `getOwnedRestaurants`: per ottenere i ristoranti posseduti da un ristoratore
- `getReviewedRestaurants`: per ottenere i ristoranti recensiti da un utente
- `addFavoriteRestaurant`: permette ad un utente di aggiungere un ristorante ai preferiti
- `removeFavoriteRestaurant`: permette ad un utente di rimuovere un ristorante dai preferiti
- `createRestaurant`: permette ad un ristoratore di creare un nuovo ristorante

3.2.3 ReviewService

Il servizio `ReviewService` fornisce metodi per l'interazione con le recensioni. I metodi definiti sono:

- `getReviews`: per ottenere tutte le recensioni di un ristorante
- `createOrUpdateReview`: per creare o aggiornare la recensione di un cliente
- `deleteReview`: permette di eliminare la recensione di un cliente

3.3 Exceptions

Il package `Exceptions` contiene le eccezioni personalizzate che i servizi possono lanciare in caso di errori. Le eccezioni gestite sono:

- `UserException`: eccezione riguardante errori relativi ai dati degli utenti
- `AddressException`: eccezione riguardante errori relativi ai dati degli indirizzi

4 Server

Il package `Server` contiene le classi necessarie per la gestione del server, che si occupa di gestire le richieste dai client interfacciandosi con il database.

4.1 Main

La classe `ServerTK` contiene il metodo `main` che avvia il server e gestisce le connessioni dei client. All'avvio del server vengono richieste le credenziali per stabilire una connessione con il database e, una volta creata, viene generato un registro RMI per poter esporre i servizi offerti dal server.

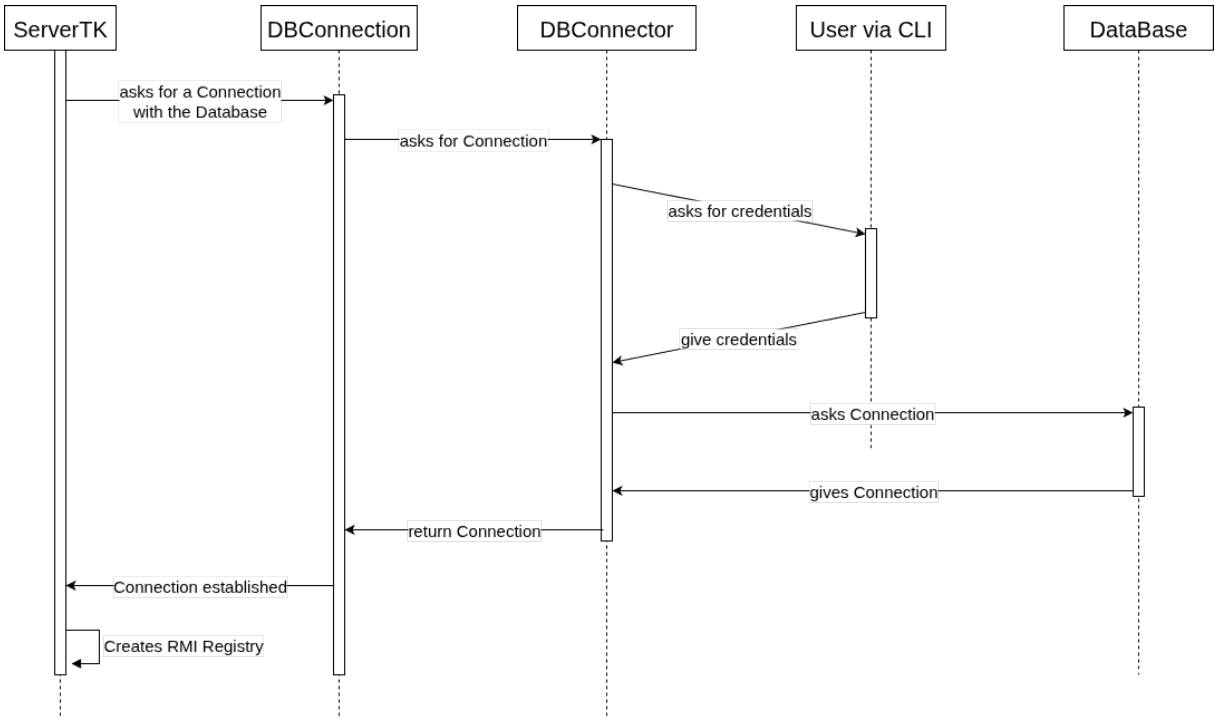

Figura 6: Interaction Diagram - Server Start-Up

4.1.1 DBConnection

La classe `DBConnection` si occupa di stabilire una singola connessione col Database e rendenderla disponibile alle altre classi del server tramite il design pattern *Singleton*. Per ottenere la connessione al database, viene utilizzata la classe *DBConnector* che si occupa di richiedere le credenziali all'utente e tentare la connesione.

4.2 Data Access Object

Il package `dao` contiene le classi che implementano il pattern Data Access Object (DAO) per l'interazione con il database isolando la logica di persistenza. Queste classi sono responsabili dell'interazione del database con l'applicazione. Le classi implementate sono le seguenti:

- `UserDAO`: gestisce le operazioni relative alla tabella *users*
- `RestaurantDAO`: gestisce le operazioni relative alla tabella *restaurants*
- `AddressDAO`: gestisce le operazioni relative alla tabella *addresses*
- `ReviewDAO`: gestisce le operazioni relative alla tabella *reviews*

Queste classi semplificano la gestione dei dati permettendo di interfacciarsi col database senza preoccuparsi delle specifiche implementative dello stesso. In queste classi sono implementate le queries per l'inserimento, la modifica e l'eliminazione dei dati e gestiscono in autonomia le eccezioni che possono verificarsi durante l'interazione con il database.

PreparedStatement Tutte le queries sono implementate utilizzando la classe `PreparedStatement` per garantire la sicurezza dell'applicazione, prevenire attacchi di tipo SQL Injection, migliorare le performance attraverso caching delle queries compilate e semplificarne la scrittura.

Favorites Notare che non è stato implementato un DAO per la tabella *favorites* poiché essendo un'entità di relazione tra *users* e *restaurants* si è ritenuto sufficiente gestirla tramite `RestaurantDAO`; le queries per la gestione dei preferiti sono dunque implementate all'interno di quest'ultima classe.

Formula di Haversine Per il calcolo della distanza tra due punti sulla superficie terrestre è stata implementata la *Formula di Haversine*.

```
(
  6371 * ACOS(
    COS(radians(?)) * COS(radians(latitude)) *
    COS(radians(longitude) - RADIANS(?)) +
    SIN(radians(?)) * SIN(RADIANS(latitude))
  )
) AS distance
```

4.3 Server Services

Il package `server_services` contiene le classi che implementano le interfacce definite nel package `common`. Queste classi vengono esposte dal database in un `Registry` e rese accessibili ai client tramite JavaRMI. I metodi delle classi garantiscono una corretta gestione della concorrenza e utilizzano le classi DAO per interagire con il database. Le classi implementate sono le seguenti:

- `RestaurantServiceImpl.java` implementa `RestaurantService`
- `ReviewServiceImpl.java` implementa `ReviewService`
- `UserServiceImpl.java` implementa `UserService`

5 Client

Il package `Client` contiene le classi necessarie per la gestione del client, che si occupa di interagire con l'utente attraverso la Graphical User Interface e di utilizzare i servizi offerti dal server tramite JavaRMI. L'architettura client implementa il pattern MVC dove:

- **Model**: Rappresentato dai DTO ricevuti dai servizi RMI
- **View**: Definita nei file FXML con componenti JavaFX
- **Controller**: Classi Java che gestiscono la logica di presentazione e l'interazione utente

5.1 ClientTK

La classe `ClientTK` contiene il metodo `main` che all'avvio lancia l'interfaccia grafica dell'applicazione e istaura una connessione col server per poter utilizzare i servizi offerti tramite il Registro RMI.
La connessione con il Registro è unica per ogni client e viene effettuata tramite la classe `ServerAddress` che, oltre a contenere le informazioni riguardanti l'indirizzo IP (o corrispettivo hostname) e la porta del server, contiene anche il metodo `getRegistry()` (unico metodo pubblico fornito dalla classe) che restituisce l'istanza `Registry` del server. La classe `ServerAddress` è implementata con il design pattern *Singleton* per garantire che ogni client ne utilizzi una sola istanza così da evitare che vi siano più connessioni da uno stesso + client e ridurre così il carico sul server.

5.2 Sessione Utente

La classe `UserSession` contiene le informazioni relative allo stato della sessione. Essa è implementata con il design pattern *Singleton* per garantirne l'unicità. I campi principali sono `isLoggedIn` e `currentUser`. Il primo è un campo booleano che indica se l'utente è attualmente loggato, il secondo è un oggetto `UserDTO` che contiene le informazioni dell'utente loggato. Oltre a conoscere lo stato di log di un utente è possibile, qualora l'utente sia loggato, saperne la tipologia così da poter differenziare l'esperienza tra cliente e ristoratore. Il campo `currentUser` è determinante anche per la ricerca dei ristoranti in quanto, contenendo le coordinate dell'utente, permette di trovare i ristoranti in base alla loro distanza.

5.3 User Interface

L'interfaccia grafica è stata interamente realizzata tramite il framework JavaFX. Essa è divisa in più viste, ognuna delle quali è definita tramite un file *FXML* presente nella cartella *TheKnife/client/src/main/resources/it/uninsubri*
Ogni componente grafico è nominato come *name-view.fxml* ed è corrisposta da una classe Java nominata *NameController.java* che ne gestisce gli eventi e le interazioni con l'utente.
I componenti importanti sono:

- **AddReply** finestra per l'aggiunta di una risposta ad una recensione (visibile solo dai ristoratori)
- **AddRestaurant** finestra per l'aggiunta di un ristorante (visibile solo dai ristoratori)
- **AddReview** finestra per l'aggiunta di una recensione (visibile solo dagli utenti registrati)
- **Login** finestra per il login
- **MyArea** finestra per la gestione dell'area personale dell'utente (visibile solo agli utenti registrati)
- **Registration** finestra per la registrazione di un nuovo utente
- **RestaurantInfo** finestra per la visualizzazione delle informazioni relative ad un ristorante
- **Search** finestra per la ricerca dei ristoranti

5.3.1 Components

Nella cartella *TheKnife/client/src/main/java/it/uninsubria/controller/ui_components/* sono presenti dei componenti grafici riutilizzabili in più finestre.
Essi sono:

- **GenericResultsComponent** componente generico (ristorante o recensione) per mostrare una lista di *cards* in modo che si possa scorrere
- **RestaurantCardComponent** componente per mostrare i ristoranti come *cards* clickabili
- **ReviewCardComponent** componente per mostrare le recensioni come *cards* clickabili

5.4 Util

Il package `utilclient` contiene la classe `UtilClient` che fornisce un metodo per il calcolo della distanza tra due coordinate geografiche. Questo metodo è utilizzato in vari punti del client.

```
public static double calculateDistance(double lat1, double lon1, double lat2, double lon2) {
    /** Radius of the Earth in kilometers */
    final int EARTH_RADIUS = 6371;
    /** Latitude distances in radians */
    double latDistance = Math.toRadians(lat2 - lat1);
    /** Longitude distances in radians */
    double lonDistance = Math.toRadians(lon2 - lon1);
    /* Haversine Formula */
    double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
            + Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2))
            * Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    return EARTH_RADIUS * c;
}
```

6 Scelte implementative

6.1 Patterns

Il progetto è stato sviluppato seguendo alcuni design patterns per garantire una struttura chiara e mantenibile.
I principali design patterns utilizzati sono:

- **Singleton**: Utilizzato per garantire che alcune classi abbiano un'unica istanza durante l'esecuzione dell'applicazione.
- **Data Transfer Object (DTO)**: Utilizzato per trasferire dati tra il client e il server in modo semplice e senza logica di business.
- **Model-View-Controller (MVC)**: Utilizzato per separare la logica di presentazione dalla logica di business e dalla gestione dei dati, in particolare nel client.
- **Data Access Object (DAO)**: Utilizzato per isolare la logica di persistenza dei dati dal resto dell'applicazione, semplificando l'interazione con il database.
- **Builder**: Utilizzato per costruire l'oggetto `SearchCriteriaDTO` in modo fluido e leggibile.

6.2 Algoritmi e Formule

6.3 Argon2

Per la gestione delle password degli utenti, si è scelto di utilizzare l'algoritmo di hashing *Argon2* come consigliato dalla OWASP. Questo algoritmo è stato scelto per la sua resistenza agli attacchi di tipo brute-force, anche parallelizzati, e per la sua capacità di essere configurabile in termini di memoria e tempo di esecuzione, rendendolo estremamente flessibile.
La sua implementazione prevede che, quando un utente si registri, la sua password venga hashata prima di essere inserita nel database; mentre quando un utente effettua il login, la password inserita venga anch'essa hashata e confrontata con quella presente nel database corrispondente allo username fornito.

6.4 Formula di Haversine

Per calcolare la distanza tra due punti sulla superficie terrestre si è utilizzata la *Formula di Haversine* sia nel client, implementandola in Java che nel server scrivendola in SQL. In particolare, è stata usata per calcolare la distanza tra l'utente e i ristoranti.

$$
\begin{aligned}
a &= \sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos(\varphi_1)\cdot\cos(\varphi_2)\cdot\sin^2\left(\frac{\Delta\lambda}{2}\right) \\
c &= 2\cdot\operatorname{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \\
d &= R\cdot c
\end{aligned}
$$

Legenda

- φ_1, φ_2 - latitudini dei due punti (in radianti)
- λ_1, λ_2 - longitudini dei due punti (in radianti)
- $\Delta\varphi = \varphi_2 - \varphi_1$ - differenza di latitudine
- $\Delta\lambda = \lambda_2 - \lambda_1$ - differenza di longitudine
- R - raggio della Terra (approssimativamente 6.371 km)
- a - valore intermedio usato per calcolare la distanza
- c - angolo centrale tra i due punti (in radianti)
- d - distanza tra i due punti lungo la superficie terrestre (in km)

7 Sitografia

- https://fxdocs.github.io/docs/html5/
- https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html
- https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- https://github.com/p-h-c/phc-winner-argon2
- https://learn.microsoft.com/it-it/aspnet/web-api/overview/data/using-web-api-with-entity-framew
  part-5
- https://rosettacode.org/wiki/Haversine_formula