Manuale Tecnico

TheKnife

Mattia Sindoni, Erica Faccio, Giovanni Isgrò

January 2026

Contents

1 Introduzione **4**
- 1.1 Limitazioni generali . . . 4

2 Requisiti tecnici **5**
- 2.1 Software . . . 5
- 2.2 Hardware . . . 5
- 2.3 Ambiente di sviluppo . . . 5

3 Installazione e configurazione **6**
- 3.0.1 Installazione dipendenze . . . 6
- 3.1 Installazione del database . . . 6

4 Architettura e scelte architetturali **7**
- 4.0.1 Architettura server side . . . 7
- 4.0.2 Architettura client side . . . 7
- 4.1 Gestione della concorrenza e modelli adottati . . . 7
- 4.2 Scelte architetturali derivate dallo sviluppo web . . . 7

5 Strutture dati e scelte algoritmiche **8**
- 5.1 Strutture dati server . . . 8
  - 5.1.1 Entità . . . 8
  - 5.1.2 Gestione richieste: Handler . . . 8
  - 5.1.3 CRUD e accesso ai dati . . . 8
  - 5.1.4 Classi fondamentali lato server . . . 8
- 5.2 Strutture dati lato client . . . 9
- 5.3 Protocollo client-server . . . 9
  - 5.3.1 Panoramica generale . . . 9
  - 5.3.2 Modello di comunicazione . . . 9
  - 5.3.3 Struttura dei messaggi . . . 10
  - 5.3.4 Esempi di messaggi . . . 10
  - 5.3.5 Gestione degli errori . . . 10
  - 5.3.6 Gestione crash e fallback lato client . . . 10
- 5.4 File di configurazione connection.ini . . . 11
  - 5.4.1 Ruolo del file . . . 11
  - 5.4.2 Struttura del file . . . 11
- 5.5 Considerazioni di sicurezza . . . 11
  - 5.5.1 Motivazioni progettuali . . . 11

6 Gestione della concorrenza **12**
- 6.1 Modello di concorrenza . . . 12
- 6.2 Accept loop e gestione connessioni . . . 12
- 6.3 Condivisione delle risorse e sicurezza . . . 12
- 6.4 Gestione degli errori . . . 12
- 6.5 Limiti . . . 13

7 Database **14**
- 7.1 Analisi del dominio . . . 14
  - 7.1.1 Nota sui profili Maven . . . 14
- 7.2 Schema logico relazionale . . . 14
  - 7.2.1 Tabella `utenti` . . . 14
  - 7.2.2 Tabella `ristorantiTheKnife` . . . 15
  - 7.2.3 Tabella `recensioni` . . . 15
  - 7.2.4 Tabella `risposte` . . . 15

7.2.5 Tabella `preferiti` . . . 15
7.3 ER non ristrutturato . . . 16
7.3.1 Considerazioni . . . 16
7.4 ER RISTRUTTURATO . . . 16
7.5 Relazioni fondamentali . . . 17
7.5.1 utenti - RistorantiTheKnife . . . 17
7.6 recensioni - utenti . . . 17
7.7 recensioni - ristorantiTheKnife . . . 17
7.8 preferiti - utenti . . . 17
7.9 Considerazioni finali . . . 17

8 Diagrammi UML **18**
8.1 Class Diagram - Moduli client e server . . . 18
8.1.1 Considerazioni progettuali . . . 18
8.2 Use Case Diagram . . . 19
8.2.1 Osservazioni . . . 19

9 Sicurezza **20**
9.1 Gestione delle credenziali . . . 20
9.2 Separazione delle responsabilità . . . 20
9.3 Protezione delle risorse condivise . . . 20
9.4 Configurazione sicura . . . 20
9.5 Limitazioni note . . . 21

1 Introduzione

Il presente manuale tecnico descrive il funzionamento interno della piattaforma TheKnife ed è rivolto principalmente a sviluppatori e manutentori che necessitano di comprendere nel dettaglio l'architettura e le scelte progettuali adottate.
Il documento, in particolare, fornisce informazioni relative a:
1) Architettura client-server;
2) protocollo di comunicazione;
3) gestione della concorrenza;
4) Struttura e progettazione del database;
5) Aspetti di sicurezza e limitazioni note;
con il fine ultimo quello fornire una base tecnica che consenta di:
- fornire tutte le informazioni necessarie per installare, configurare o estendere il sistema;
- comprendere le principali scelte architetturali;

1.1 Limitazioni generali

Il modello architetturale adottato per la piattaforma *TheKnife* presenta alcune limitazioni note, principalmente dovute alla natura didattica del progetto e alle scelte progettuali orientate alla semplicità e alla chiarezza.

In particolare:

- la scalabilità del server risulta limitata in presenza di un numero elevato di client connessi simultaneamente;
- il modello *thread-per-connection* comporta un consumo crescente di risorse al crescere del numero di connessioni attive;
- l'assenza di meccanismi avanzati di bilanciamento del carico può incidere sulle prestazioni in scenari ad alta concorrenza.

Tali limitazioni rappresentano un compromesso consapevole, adottato per privilegiare la semplicità implementativa, la leggibilità del codice e la facilità di manutenzione, risultando adeguate al contesto accademico del progetto.

2 Requisiti tecnici

2.1 Software

1) Java Development Kit (JDK) 17 o superiore;
necessaria per la compilazione ed esecuzione dei due moduli client e server;
2) Apache Maven 3.8 o superiore
Utilizzato per la gestione delle dipendenze e il ciclo di build del progetto;
3) PostgreSQL 13 o superiore
Utilizzato come sistema di gestione del database relazionale utilizzato dal server
4) JavaFX
richiesto per l'esecuzione dell'interfaccia del client

2.2 Hardware

Processore dual-core, con almeno 4GB di RAM e spazio su disco sufficiente per il database e i file di configurazione.

2.3 Ambiente di sviluppo

Consigliato:
Intellij IDEA 2;
Eclipse IDE 3;
Visual studio code, con supporto Maven e Java.
Tali ambienti consentono una gestione agevole dei moduli del progetto, facilitando il processo di build, il debugging e la manutenzione del codice.

3 Installazione e configurazione

Per poter utilizzare l'applicazione TheKnife, prima di tutto occorre rispettare i requisiti tecnici "vedere paragrafo **2 - Requisiti tecnici**".
Soddisfatti tali requisiti, andrà avviato il processo di build per installare le dipendenze.

3.0.1 Installazione dipendenze

Una volta clonata la repository, andare all'interno della directory principale "LabB" e, una volta dentro, eseguire il seguente comando da terminale:
mvn clean package
esso non solo genererà i .jar all'interno delle rispettive directory client e server, ma andrà anche ad eseguire il processo di build dei due pom.xml lato server e lato client insieme.

3.1 Installazione del database

Per utilizzare l'applicazione, occorre aver prima di tutto aver creato su postgres il database "*theknife*"
. Per poter creare il database postgres occorre restare all'interno della root del laboratorio "*LabB*" e, da terminale, occorre far partire i seguenti comandi:
$ENVPGPASSWORD = "LA-TUA-PASSWORD-POSTGRES"
E successivamente:
mvn -P init-db validate
Tale script maven si occuperà di creare il database theknife con le relative tabelle.

4 Architettura e scelte architetturali

L'applicazione Theknife è basata su un'architettura client-server, nella quale il client e il server sono due moduli distinti che comunicano tramite socket TCP. Questa separazione consente una chiara distinzione di responsabilità, migliorando la manutenibilità e la scalabilità del sistema.

4.0.1 Architettura server side

Il server rappresenta il nodo centrale del sistema, ed è responsabile di:

- Accettare nuove connessioni client;
- gestire la comunicazione concorrente tramite thread dedicati
- coordinare l'accesso alle risorse condivise (database, file di configurazione)
- mantenere lo stato generale dell'applicazione

4.0.2 Architettura client side

Il client, invece, si occupa di:

- stabilire la connessione con il server;
- inviare richieste e ricevere risposte;
- gestire l'interazione con l'utente tramite GUI
- delegare la comunicazione di rete a thread dedicati, evitando il blocco dell'interfaccia.

La comunicazione tra client e server avviene tramite protocollo testuale su TCP, garantendo affidabilità nella trasmissione dei dati.

4.1 Gestione della concorrenza e modelli adottati

A livello client sono state adottate soluzioni tipiche dello sviluppo mobile, adottando il concetto di handler predisposti per la gestione asincrona delle operazioni di rete.
Le operazioni di comunicazione con il server vengono eseguite in thread separati rispetto al thread principale dell'interfaccia, e i risultati delle operazioni asincrone vengono poi gestiti tramite meccanismi di notifica o callback, evitando il blocco dell'interfaccia utente e migliorando la responsività.
Questa scelta progettuale ha consentito:

- una migliore user experience;
- una gestione ordinata degli eventi asincroni;
- una separazione chiara tra logica di comunicazione e logica di presentazione.

4.2 Scelte architetturali derivate dallo sviluppo web

All'interno del progetto sono state applicate anche scelte architetturali tipiche dello sviluppo web, in particolare l'utilizzo del Singleton Pattern.
Il pattern Singleton è stato adottato per gestire componenti che devono avere un'unica istanza condivisa all'interno dell'applicazione, come:

- gestione della connessione;
- componenti di configurazione;
- risorse condivise lato server

L'uso di suddetto pattern ha consentito non solo di evitare duplicazioni non necessarie di risorse, ma anche ha garantito la coerenza dello stato dell'applicazione e ha semplificato l'accesso ai servizi del sistema

5 Strutture dati e scelte algoritmiche

Nella seguente sezione, verranno elencate le scelte algoritmiche prese per garantire il funzionamento dell'applicazione.

5.1 Strutture dati server

5.1.1 Entità

Le entità principali del dominio dell'applicazione (utente, ristorante, recensione, risposta, preferiti) sono rappresentate tramite classi Java che riflettono direttamente la struttura delle tabelle del database.

La corrispondenza uno-a-uno ha consentito:

- semplicità di mapping tra database e applicazione;
- riduzione della complessità logica;
- maggiore leggibilità del codice.

5.1.2 Gestione richieste: Handler

Le richieste provenienti dai client vengono gestite tramite handler predisposti, ciascuno responsabilie di un insieme coerente di operazioni su una specifica entità.
Dal punto di vista delle strutture dati:

- le richieste sono rappresentate da stringhe testuali;
- il contesto della richiesta viene analizzato sequenzialmente dai vari Handler;
- il primo handler che è in grado di riconoscere e gestire la richiesta se ne assume la responsabilità.

5.1.3 CRUD e accesso ai dati

L'accesso al database è organizzato tramite classi CRUD dedicate:
- GenericCRUD viene utilizzato come punto di smistamento centrale;
- CRUD specifici per ciascuna entità (utente, ristorante, recensione, ecc...) Le query SQL vengono costruite in modo parametrico, evitando duplicazioni di codice e migliorando la manutenibilità.

5.1.4 Classi fondamentali lato server

L'architettura server dell'applicazione si basa su un insieme di classi centrali, ciascuna con responsabilità ben definite, al fine di garantire modularità, chiarezza e facilità di estensione.

ServerApplication La classe `ServerApplication` rappresenta il punto di ingresso del server. Essa si occupa di:

- inizializzare il `ServerSocket`;
- avviare l'*accept loop*;
- gestire l'arresto controllato del server tramite comando di terminazione.

ClientThread Per ogni connessione client accettata, viene istanziato un oggetto `ClientThread`. Questa classe gestisce l'intero ciclo di vita della comunicazione con il client:

- lettura delle richieste dal socket;
- inoltro delle richieste agli Handler tramite catena di responsabilità;
- invio delle risposte al client;
- gestione degli errori e chiusura controllata della connessione.

ConnectionManager La classe `ConnectionManager` è responsabile della gestione centralizzata delle informazioni di connessione al database. Essa legge i parametri dal file `connection.ini` e garantisce:

- separazione tra configurazione e logica applicativa;
- unicità dell'accesso al database tramite Singleton Pattern;
- maggiore sicurezza e manutenibilità.

Logger Il `Logger` fornisce un'interfaccia centralizzata per la visualizzazione di messaggi di log, errori e banner informativi, facilitando il debugging e il monitoraggio dello stato del server durante l'esecuzione.

5.2 Strutture dati lato client

Lato client, i dati ricevuti dal server vengono temporaneamente memorizzati in:

- Strutture di tipo List e Map per collezioni di risultati;
- oggetti dominio per la rappresentazione delle entità.

Tali strutture dati sono progettate per:

- Supportare aggiornamenti dinamici dell'interfaccia
- facilitare la navigazione tra le view;
- mantenere la separazione tra logica applicativa e presentazione

5.3 Protocollo client-server

5.3.1 Panoramica generale

La comunicazione tra client e server nell'applicazione TheKnife è basata su socket TCP e utilizza un protocollo applicativo testuale custom, progettato specificamente per le esigenze del progetto.

Il modello adottato è di tipo

- Client-server sincrono
- connection oriented
- thread-per-connection lato server.

Il server resta in ascolto su una porta configurabile e accetta connessioni multiple, assegnando a ciascun client un thread dedicato.

5.3.2 Modello di comunicazione

Il flusso di comunicazione segue il seguente schema:

1. Il client apre una connessione TCP verso il server.
2. Il server accetta la connessione tramite un accept loop.
3. Viene creato un ClientThread dedicato.
4. Il client invia richieste testuali strutturate.
5. Il server interpreta la richiesta e la inoltra agli handler.
6. Il server restituisce una risposta al client.
7. La connessione rimane aperta fino a:

– logout del client;

– chiusura del client;

– spegnimento controllato del server

5.3.3 Struttura dei messaggi

Il protocollo applicativo utilizza messaggi testuali strutturati secondo una sintassi semplice e facilmente estendibile.

Ogni messaggio è composto da:

- un **comando**;
- una sequenza di **parametri**, separati da delimitatori.

La struttura generale è la seguente:

```
COMANDO|parametro1|parametro2|...|parametroN
```

5.3.4 Esempi di messaggi

Login utente

```
LOGIN|username|password
```

Risposta del server

```
OK|LOGIN_SUCCESS
```

oppure

```
ERROR|INVALID_CREDENTIALS
```

Inserimento recensione

```
ADD_REVIEW|idRistorante|stelle|testo
```

5.3.5 Gestione degli errori

Gli errori vengono restituiti sotto forma di messaggi testuali standardizzati, consentendo al client di:

- distinguere errori applicativi da errori di rete;
- visualizzare messaggi coerenti all'utente;
- mantenere la connessione attiva quando possibile.

Questo approccio mantiene il protocollo semplice, leggibile e facilmente debuggabile.

5.3.6 Gestione crash e fallback lato client

Lato client è stata implementata una gestione preventiva delle condizioni di errore e di disconnessione improvvisa dal server, al fine di garantire una transizione pulita dell'interfaccia grafica.

La classe `ONLINEchecker` viene passata a tutti i controller della GUI ed è responsabile di:

- monitorare lo stato della connessione con il server;
- intercettare eventuali crash o errori di rete;
- attivare un meccanismo di *fallback* in caso di disconnessione.

In caso di errore critico, il sistema effettua automaticamente il logout dell'utente, evitando stati incoerenti dell'interfaccia e garantendo una corretta chiusura della sessione.

Un'ulteriore misura di sicurezza è implementata tramite il metodo `getInteractiveNodes` della classe `AppController`, che consente di disabilitare dinamicamente tutti i pulsanti dell'interfaccia grafica in presenza di errori, prevenendo azioni non valide da parte dell'utente.

5.4 File di configurazione connection.ini

Il file connection.ini è un file di configurazione lato server, utilizzato per separare i parametri sensibili e le variabili dalla logica applicativa.

5.4.1 Ruolo del file

Il file connection.ini consente di:
- configurare il collegamento al database senza modificare il codice;
- facilitare il deployement su ambienti diversi;
- migliorare la manuntenibilità del sistema.
Il file viene letto all'avvio del server dalla classe `ConnectionManager`.

5.4.2 Struttura del file

Il file è strutturato come segue
[DATABASE] host=localhost port=5432 databasename=theknife username=postgres password=CHANGE-ME

5.5 Considerazioni di sicurezza

Al primo avvio del server tramite cmd, verrà richiesto all'utente di inserire delle informazioni per la connessione al database postgres.
L'unico campo che è obbligatorio modificare è la password, in modo da inserire la password impostata durante l'installazione del database PostgreSQL.

5.5.1 Motivazioni progettuali

L'introduzione e l'utilizzo di un file .ini ha consentito di:

- evitare hardcoding delle credenziali;
- centralizzare la configurazione
- semplificare estensioni future

6 Gestione della concorrenza

La gestione della concorrenza in TheKnife è un aspetto cruciale dell'architettura server poichè l'intero sistema è progettato per supportare più client connessi simultaneamente attraverso una comunicazione di rete basata su socket TCP.

6.1 Modello di concorrenza

Il server adotta un modello multi-thread concorrente, dove:

- il server rimane in ascolto su una porta TCP;
- per ogni nuova connessione in ingresso, viene creato un thread dedicato;
- ogni thread gestisce in modo indipendente la comunicazione con un singolo client.

Questo modello consente non solo al server di servire più client contemporaneamente, ma anche di isolare le sessioni ed evitare deadlock e race condition.

6.2 Accept loop e gestione connessioni

La classe `ServerApplication` utilizza un accept loop, ossia un ciclo continuo che:

- attende richieste di connessione tramite `ServerSocket.accept()`;
- istanzia un nuovo `ClientThread`;
- assegna il socket al thread appena creato;
- avvia il thread per la gestione della sessione.

Il server rimane attivo fino alla ricezione del comando di terminazione (`quit`), che provoca:

- l'interruzione controllata dell'accept loop;
- la chiusura delle connessioni attive;
- il rilascio delle risorse di sistema.

6.3 Condivisione delle risorse e sicurezza

Le risorse condivise come il database non vengono accedute direttamente dai trhead, ma tramite:

- classi di servizio dedicate;
- istanze centralizzate gestite dal Singleton Pattern

Questo approccio ha ridotto le condizioni di race condition, proteggendo il sistema dagli accessi concorrenti non controllati e da duplicazioni di connessioni al database.

6.4 Gestione degli errori

In caso di:

- errori di rete;
- disconnessione improvvisa del client;
- eccezioni durante l'elaborazione delle richieste

Il thread coinvolto:

- intercetta l'errore;
- chiude correttamente la connessione
- termina la propria esecuzione senza influire sugli altri thread.

6.5 Limiti

Il modello adottato presenta alcune limitazioni:

- Scalabilità limitata in presenza di un numero elevato di client;
- consumo crescente di risorse al crescere dei thread

Ciò tuttavia è un compromesso adottato per una questione di semplicità e di funzionalità.

7 Database

Il sistema *TheKnife* utilizza un database relazionale basato su **PostgreSQL** per la gestione persistente dei dati applicativi. La progettazione del database è stata condotta seguendo un approccio metodologico strutturato, articolato nelle seguenti fasi:

- analisi del dominio applicativo;
- individuazione delle entità e delle relazioni;
- definizione dello schema Entity–Relationship (ER);
- ristrutturazione dello schema ER;
- traduzione in schema logico relazionale;
- implementazione fisica sul databaseMS.

7.1 Analisi del dominio

Dall'analisi dei requisiti funzionali emergono le seguenti entità principali:

- Utente
- Ristorante
- Recensione
- Risposta
- Preferito

Ogni entità rappresenta un concetto autonomo del dominio applicativo ed è stata progettata in modo da ridurre ridondanze e garantire coerenza dei dati.

7.1.1 Nota sui profili Maven

L'inizializzazione del database avviene tramite un profilo Maven dedicato. È fondamentale utilizzare il profilo corretto definito nel progetto:

```
mvn -P init-db validate
```

Il profilo `init-db` esegue gli script necessari alla creazione del database *TheKnife* e delle relative tabelle. Eventuali variazioni del nome del profilo devono essere coerenti con quanto definito nel file `pom.xml`.

7.2 Schema logico relazionale

Il database è composto da cinque tabelle fondamentali.

7.2.1 Tabella `utenti`

- `id` SERIAL PRIMARY KEY
- `nome` VARCHAR(100) NOT NULL
- `cognome` VARCHAR(100) NOT NULL
- `username` VARCHAR(100) UNIQUE NOT NULL
- `password` CHAR(60) NOT NULL

- `data_nascita` DATE
- `latitudine_domicilio` DOUBLE PRECISION NOT NULL
- `longitudine_domicilio` DOUBLE PRECISION NOT NULL
- `is_ristoratore` BOOLEAN NOT NULL

7.2.2 Tabella `ristorantiTheKnife`

- `id` SERIAL PRIMARY KEY
- `nome` VARCHAR(100) NOT NULL
- `nazione` VARCHAR(100) NOT NULL
- `citta` VARCHAR(100) NOT NULL
- `indirizzo` VARCHAR(100) NOT NULL
- `latitudine` DOUBLE PRECISION NOT NULL
- `longitudine` DOUBLE PRECISION NOT NULL
- `fascia_prezzo` INTEGER NOT NULL
- `servizio_delivery` BOOLEAN NOT NULL
- `prenotazione_online` BOOLEAN NOT NULL
- `tipo_cucina` VARCHAR(255) NOT NULL
- `proprietario` INTEGER REFERENCES utenti(id)

7.2.3 Tabella `recensioni`

- `id` SERIAL PRIMARY KEY
- `id_utente` INTEGER REFERENCES utenti(id)
- `id_ristorante` INTEGER REFERENCES ristorantiTheKnife(id) ON DELETE CASCADE
- `stelle` INTEGER NOT NULL
- `testo` VARCHAR(255) NOT NULL

7.2.4 Tabella `risposte`

- `id_recensione` INTEGER PRIMARY KEY REFERENCES recensioni(id) ON DELETE CASCADE
- `testo` VARCHAR(255) NOT NULL

7.2.5 Tabella `preferiti`

- `id_utente` INTEGER REFERENCES utenti(id)
- `id_ristorante` INTEGER REFERENCES ristorantiTheKnife(id) ON DELETE CASCADE
- PRIMARY KEY (id_utente, id_ristorante)

7.3 ER non ristrutturato

Di seguito viene mostrato lo schema ER prima della ristrutturazione

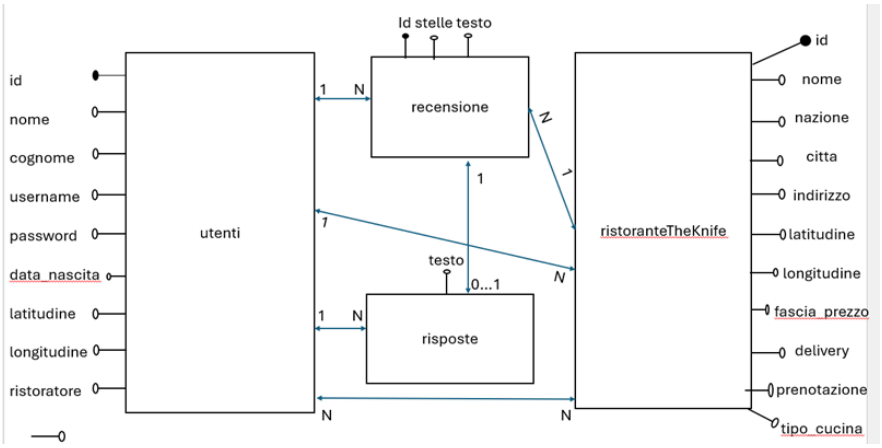

Figure 1: Schema ER non ristrutturato

7.3.1 Considerazioni

Il seguetne modello ER non ristrutturato presenta relazioni implicite e mancanza di vincoli espliciti. Inoltre, le cardinalità a questo stato non sono state ancora ottimizzate, e il ruolo di ristoratore è modellato come relazione e non come attributo.

7.4 ER RISTRUTTURATO

Questo schema rappresenta la versione ristrutturata e normalizzata, coerente con lo schema logico implementato in PostgreSQL:

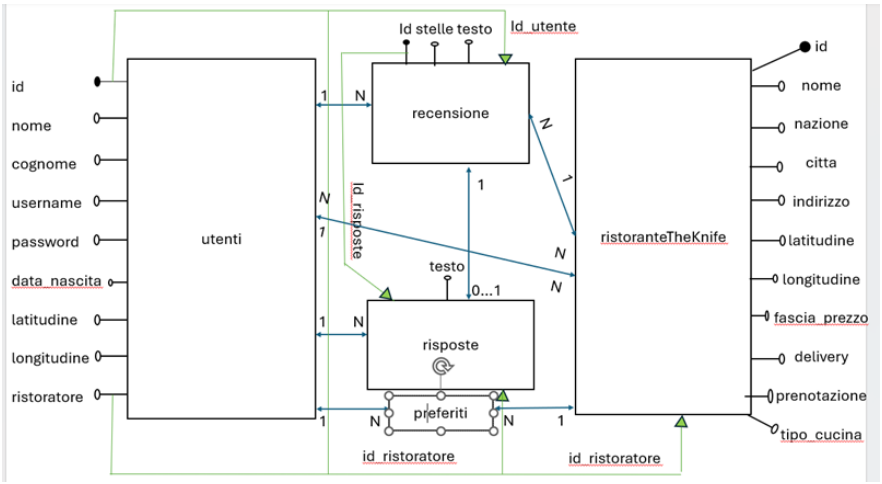

Figure 2: Schema ER ristrutturato

La ristrutturazione dello schema Entity–Relationship è stata effettuata per:

- eliminare ambiguità nelle relazioni;
- introdurre chiavi primarie e chiavi esterne esplicite;

- garantire l'integrità referenziale;
- semplificare la traduzione in schema logico relazionale.

In particolare:

- la relazione *Preferito* è stata trasformata in una tabella di associazione con chiave composta;
- la relazione uno-a-uno tra *Recensione* e *Risposta* è stata resa esplicita tramite chiave primaria condivisa;
- il ruolo del ristoratore è stato modellato tramite attributo e vincolo di proprietà.

7.5 Relazioni fondamentali

7.5.1 utenti - RistorantiTheKnife

uno a molti: un utente ristoratore può avere più ristoranti, ma un ristorante è di proprietà di un solo ristoratore

7.6 recensioni - utenti

uno a molti: una recensione viene scritta da un solo utente, ma un utente può scrivere più recensioni per ristoranti diversi

7.7 recensioni - ristorantiTheKnife

uno a molti: un ristorante può ricevere più recensioni, mentre ogni recensione si riferisce a un solo ristorante

7.8 preferiti - utenti

uno a molti: un utente ha più preferiti, un preferito appartiene a un utente

7.9 Considerazioni finali

Le scelte progettuali adottate garantiscono:

- integrità referenziale;
- assenza di ridondanze;
- scalabilità futura;
- coerenza tra modello concettuale e implementazione fisica.

8 Diagrammi UML

I seguenti diagrammi garantiscono una visione a 360 gradi della struttura dell'applicazione

8.1 Class Diagram - Moduli client e server

Il seguente class diagram rappresenta la struttura del sistema TheKnife e permette di definire:

- La separazione tra modulo client e server
- l'applicazione di pattern architetturali
- la suddivisione delle responsabilità di rete, logica applicativa ed accesso ai dati

Il diagramma evidenzia come il client sia responsabile esclusivamente dell'interazione dello user e della comunicazione col server, mentre il server gestisce concorrenza, logiche e l'accesso al database

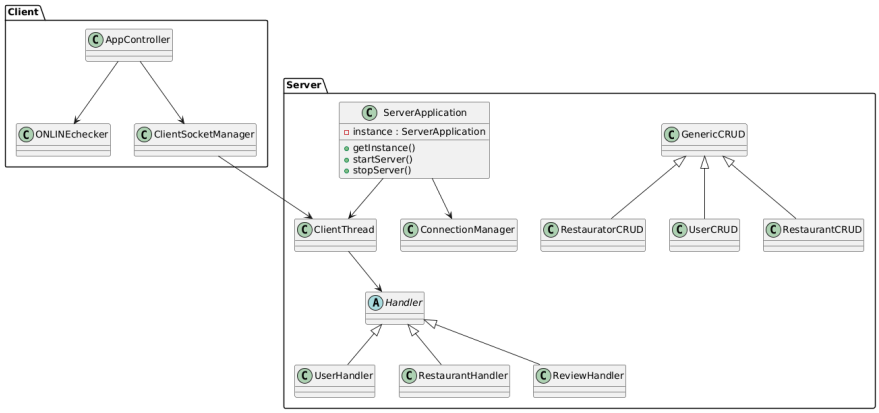

Figure 3: Class Diagram moduli client-server

8.1.1 Considerazioni progettuali

- L'applicazione del Singleton Pattern nella serverApplication garantisce un'unica istanza del server;
- gli Handler smistano le richieste in base al contesto;
- Separazione delle responsabilità: GUI, networking, logica e persistenza vengono nettamente distinti.

8.2 Use Case Diagram

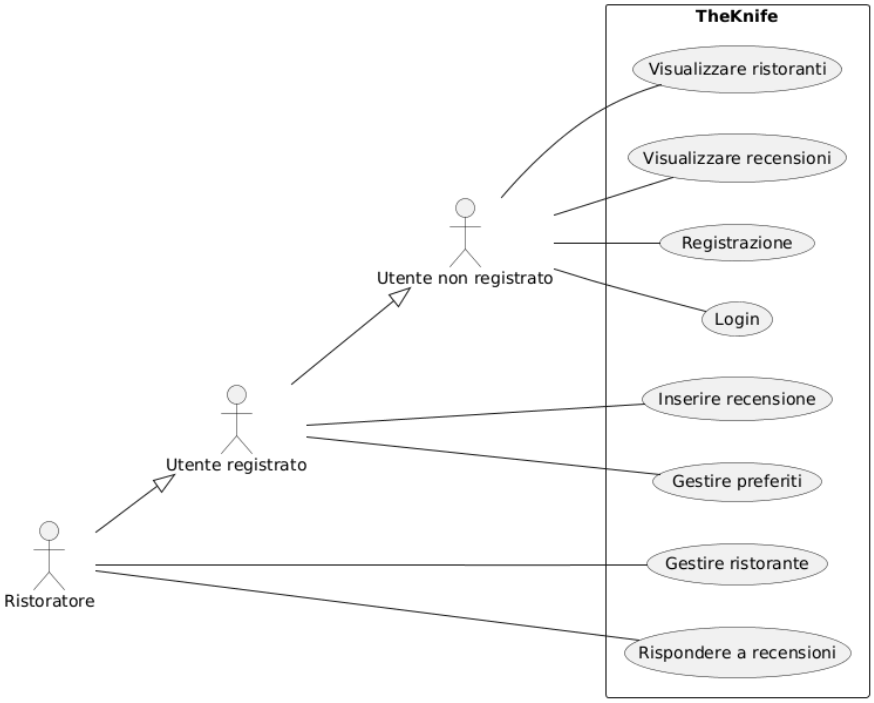

Figure 4: Use-case Diagram

Lo Use case Diagram rappresenta le funzionalità offerte dal punto di vista dell'utente finale.
Il diagramma consente di collegare i requisiti funzionali all'implementazione software in base agli utilizzatori dell'applicazione:

- User non registrato
- User registrato
- Ristoratore

8.2.1 Osservazioni

- La gerarchia degli attori riflette il modello dei ruoli;
- le funzionalità sono incrementali in base al livello di autenticazione
- il diagramma fornisce un ponte tra manuale utente e manuale tecnico.

9 Sicurezza

La sicurezza della piattaforma *TheKnife* è stata affrontata tenendo conto del contesto accademico del progetto, adottando soluzioni semplici ma corrette dal punto di vista progettuale.

9.1 Gestione delle credenziali

Le credenziali degli utenti non vengono mai memorizzate in chiaro. Le password sono salvate nel database sotto forma di **hash**, rendendo impossibile la loro ricostruzione diretta in caso di accesso non autorizzato al database.

L'autenticazione avviene esclusivamente lato server, evitando che il client possa effettuare verifiche autonome sulle credenziali.

9.2 Separazione delle responsabilità

Il sistema è progettato in modo da separare nettamente:

- interfaccia utente (client);
- logica applicativa (server);
- persistenza dei dati (database).

Questa separazione riduce la superficie di attacco e limita l'impatto di eventuali vulnerabilità.

9.3 Protezione delle risorse condivise

L'accesso alle risorse critiche, come il database e i file di configurazione, avviene tramite:

- classi centralizzate;
- istanze gestite tramite Singleton Pattern.

Questo approccio evita:

- accessi concorrenti non controllati;
- creazione multipla di connessioni al database;
- inconsistenze nello stato applicativo.

9.4 Configurazione sicura

L'uso del file `connection.ini` consente di:

- evitare l'hardcoding delle credenziali;
- escludere informazioni sensibili dal codice sorgente;
- facilitare la rotazione delle credenziali.

Il file non è destinato alla distribuzione pubblica ed è pensato per essere protetto a livello di filesystem.

9.5 Limitazioni note

Il progetto presenta alcune limitazioni dal punto di vista della sicurezza:

- assenza di cifratura TLS sul canale TCP;
- protocollo testuale non cifrato;
- gestione delle sessioni basata sulla connessione.

Tali limitazioni sono accettabili nel contesto del progetto didattico, ma andrebbero superate in un contesto produttivo tramite:

- utilizzo di TLS;
- token di sessione;
- meccanismi di rate limiting.