UNIVERSITÀ
DEGLI STUDI
FIRENZE

JavaBrew: Gestione smart per vending machines

Anno 2025

Matteo Minin, Simone Pellicci, Simone Suma

Professore: Enrico Vicario

Corso di laurea triennale in Ingegneria Informatica

Indice

1 Introduzione

Il progetto, JavaBrew nasce dall'esigenza di modernizzare la gestione e l'erogazione di prodotti attraverso i distributori automatici per snack e bevande. L'obiettivo principale è creare una piattaforma integrata, semplice e sicura, che supporti in modo completo l'utente finale quanto gli operatori e gli amministratori del sistema.

JavaBrew è un software di gestione per una rete di distributori automatici connessi alla rete. Grazie all'applicazione mobile e a un wallet digitale associato all'account personale, l'utente può acquistare articoli inquadrando con lo smartphone il codice QR univoco visualizzato sulla macchina. Il gestionale della piattaforma offre una dashboard che mostra statistiche sulle vendite e sui malfunzionamenti, permettendo agli amministratori di monitorare vendite, scorte e pianificare in modo efficiente le attività di manutenzione e rifornimento.

In tal modo, JavaBrew mira a ottimizzare l'operatività quotidiana delle vending machine, migliorare l'esperienza d'uso degli utenti e ridurre i tempi di intervento per gli operatori, tramite un servizio scalabile e in linea con le tecnologie moderne.

2 Analisi dei Requisiti

2.1 Descrizione del Problema

Le moderne vending machine per caffè e snack spesso soffrono di mancanza di integrazione digitale:

- Non tengono traccia in tempo reale delle scorte
- Offrono pagamenti solo in contanti o chiavette prepagate (metodi di pagamento datati)
- Non dispongono di strumenti centralizzati per la gestione e la manutenzione.

JavaBrew intende risolvere queste lacune, offrendo una piattaforma che unisce pagamenti digitali e monitoraggio remoto, migliorando la gestione delle macchine e l'esperienza utente.

2.2 Attori Coinvolti

Customer: L'utente può registrarsi o accedere con il proprio account per visualizzare il saldo del wallet (borsellino digitale) e lo storico delle transazioni. Può ricaricare il conto in contanti recandosi presso una macchinetta oppure tramite altri metodi di pagamento online. L'acquisto dei prodotti si effettua connettendosi a un distributore automatico, scansionando il codice QR univoco della macchinetta e selezionando il prodotto che si vuole acquistare; il saldo verrà scalato automaticamente dal wallet digitale.

Admin: L'amministratore della piattaforma è responsabile della configurazione delle macchine, della gestione dei prezzi, dei report di vendita e delle statistiche generali. Ha accesso a funzionalità CRUD per tutti gli utenti (in particolare i manutentori), per i singoli distributori e gli articoli.

Worker: Tecnico incaricato della manutenzione e del rifornimento dei distributori. È guidato da report automatici generati dall'autodiagnosi dei distributori che segnalano scorte insufficienti o malfunzionamenti.

2.3 Possibili Svantaggi dell'Approccio

Dipendenza dalla Connettività: In assenza di connessione a Internet, il distributore automatico non sarà in grado di segnalare guasti né di registrare le transazioni effettuate. Questo problema potrebbe essere affrontato nelle successive iterazioni del software, implementando un meccanismo di rilevamento dei distributori non connessi (ad esempio tramite polling) e introducendo un *offline register* per tracciare localmente le transazioni avvenute durante il periodo di disconnessione. Una volta ristabilita la connessione, tali transazioni verrebbero sincronizzate e inserite nel database centrale.

Curva di Apprendimento per l'Utente: Alcuni utenti potrebbero inizialmente trovare complesso l'uso dell'applicazione e la connessione tramite QR code. Nelle successive iterazioni del software, questo problema potrebbe essere risolto permettendo a utenti non registrati (**anonymous users**) di effettuare transazioni solamente in contante.

Costi Infrastrutturali: L'installazione di modem e server comporta un investimento iniziale considerevole, che però dovrebbe essere ammortizzato nel tempo tramite manutenzioni più rapide e una selezione strategica dei prodotti in base ai gusti locali.

3 Ambiente e tecnologie utilizzate

L'applicazione è stata sviluppata usando le seguenti tecnologie

3.1 Linguaggio e Tool di Build

- **Java 24** - È stato scelto per la compatibilità con le specifiche Jakarta EE e le versioni supportate da Hibernate ORM.
- **Maven 3.9.9** - Strumento di build e gestione delle dipendenze.

3.2 Database e Persistenza

- **PostgreSQL 16** - Database relazionale utilizzato in produzione.

- **H2 Database** - Utilizzato per i test automatici. Il database è in-memory e consente test rapidi.
- **Jakarta Persistence API (JPA) 3.2.0** - Standard per la gestione della persistenza tramite ORM.
- **Hibernate ORM 6.5.2.Final** - Implementazione di JPA utilizzata per ORM. Permette la generazione automatica dello schema a partire dal modello a oggetti (`hibernate.hbm2ddl.auto`).
- **Jakarta Transaction API 2.0.1** - Utilizzata per la gestione delle transazioni distribuite.

3.3 Containerizzazione

- **Docker** - Utilizzato per containerizzare l'applicazione, il database PostgreSQL. La scelta è motivata dalla necessità di garantire ambienti facilmente distribuibili.
- **Docker Compose** - Utilizzato per definire e avviare l'intera architettura multi-container tramite un unico file (`docker-compose.yml`). Consente la gestione coordinata dell'applicazione, del database e degli strumenti di supporto.

3.4 Testing e Analisi

- **JUnit 5.11.0** - Framework per l'esecuzione dei test unitari.
- **Mockito 5.18.0** - Utilizzato per la simulazione di dipendenze durante i test (mocking).
- **Jacoco** - Strumento per la generazione dei report di copertura del codice.

4 Use Cases

I seguenti template descrivono alcuni dei principali casi d'uso identificati per l'applicazione durante la fase di analisi e progettazione. Ogni caso d'uso è descritto utilizzando una struttura standardizzata, che comprende i seguenti campi:

- **ID**: Identificativo univoco del caso d'uso.
- **Nome**: Titolo descrittivo del caso d'uso.
- **Livello**: Classificazione del caso d'uso (es. Function Goal, User Goal).
- **Attori**: Utenti coinvolti nell'interazione.
- **Pre-condizioni**: Condizioni che devono essere vere prima dell'avvio del flusso.
- **Post-condizioni**: Risultato atteso alla fine del flusso.

- **Flusso Principale**: Sequenza di passi standard che descrivono il comportamento del sistema.
- **Flusso Alternativo**: Eventuali deviazioni dal flusso principale, incluse eccezioni o errori gestiti.
- **Test**: Test di integrazione o funzionali previsti per verificare il corretto funzionamento del caso d'uso.

4.1 User Use Cases

Il seguente use case diagram descrive i principali casi d'uso di user (Admin, Customer, Worker).

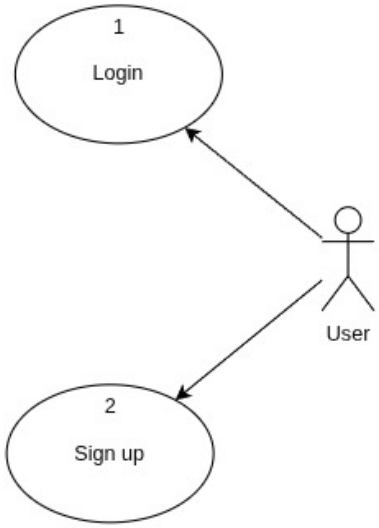


Figura 1: User Use Cases

4.1.1 User Login

UC-1	**User Login**
Level	System Goal
Actors	User
Pre-conditions	Lo user ha un account nel sistema ma non si è ancora autenticato.
Post-conditions	Lo user è autenticato e ha accesso all'interfaccia personalizzata del sistema.
Basic Flow	1. Lo user apre la pagina di login (Fig. 6). 2. Inserisce la propria mail e password e clicca il tasto Login. 3. Il sistema verifica le credenziali. 4. Il sistema fornisce accesso all'interfaccia personalizzata (ad esempio, Fig. 9a per il Cliente o Fig. 8 per l'Amministratore, a seconda del ruolo).
Alternative Flow	3.1 Credenziali errate: messaggio di errore che invita a riprovare. 4.1 Errore di sistema: messaggio di errore che invita a riprovare più tardi.
Test	Vedi i test correlati

4.1.2 User Sign up

UC-2	**User Registration**
Level	System Goal
Actors	User
Pre-conditions	Lo user non è autenticato nel sistema e non ha un profilo.
Post-conditions	Lo user ha un profilo, è autenticato e ha accesso all'interfaccia personalizzata del sistema.
Basic Flow	1. Lo user apre la pagina per registrare un account (Fig. 7). 2. Inserisce nome, cognome, email e password. 3. Il sistema valida i campi inseriti. 4. Il sistema crea un nuovo account.
Alternative Flow	3.1 Campi errati o vuoti: il sistema mostra un messaggio di errore. 4.1 Errore di autenticazione: appare un messaggio che invita a riprovare più tardi.
Test	Vedi i test correlati

4.2 Customer Use Cases

Il seguente use case diagram descrive i principali use cases che riguardano il customer.

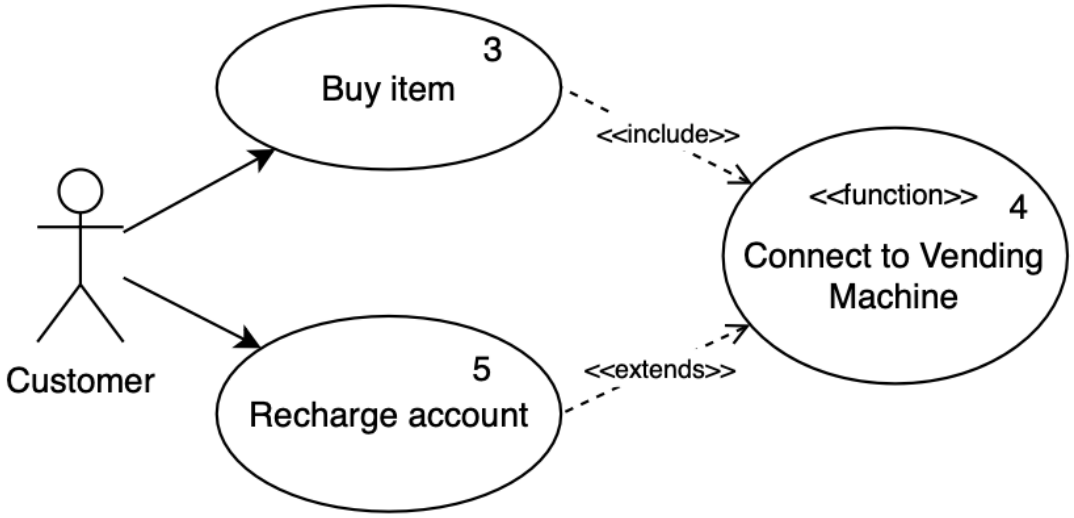


Figura 2: Cusomer Use Cases

4.2.1 Buy Item

UC-3	**Buy Item**
Level	User Goal
Actors	Customer
Pre-conditions	Il customer è autenticato ed è connesso a una vending machine.
Post-conditions	Il prodotto è stato erogato e il saldo aggiornato.
Basic Flow	1. Il customer si trova nell'interfaccia del catalogo prodotti (Fig. 10). 2. Seleziona un prodotto. 3. Il sistema controlla saldo e disponibilità. 4. Il sistema scala il saldo ed eroga il prodotto. 5. Il customer viene disconnesso.
Alternative Flow	3.1 Saldo insufficiente: errore e disconnessione. 3.2 Prodotto esaurito: errore.
Test	Vedi i test correlati

4.2.2 Connect to Vending Machine

UC-4	**User Vending Machine Connection**
Level	Function Goal
Actors	Customer
Pre-conditions	Il customer è autenticato.
Post-conditions	Il customer è connesso alla vending machine e vede l'interfaccia relativa alla macchinetta.
Basic Flow	1. Il customer scannerizza il QR code. 2. Il sistema collega la vending machine associata. 3. Il customer vede l'interfaccia dell'inventario (Fig. 10).
Alternative Flow	• Vending machine già connessa: errore. • Vending machine fuori uso: errore.
Test	Vedi i test correlati

4.2.3 Recharge Balance

UC-5	**Recharge Balance**
Level	User Goal
Actors	Customer
Pre-conditions	Il customer è autenticato.
Post-conditions	Il customer vede il nuovo saldo aggiornato.
Basic Flow	1. Il customer si trova nella propria pagina personale (Fig. 9a). 2. Sceglie metodo di pagamento e importo. 3. Il sistema verifica e processa la transazione.
Alternative Flow	3.1 Pagamento non riuscito.
Test	Vedi i test correlati

4.3 Worker Use Cases

Il seguente use diagramma descrive i principali casi d'uso dell'utente Worker.

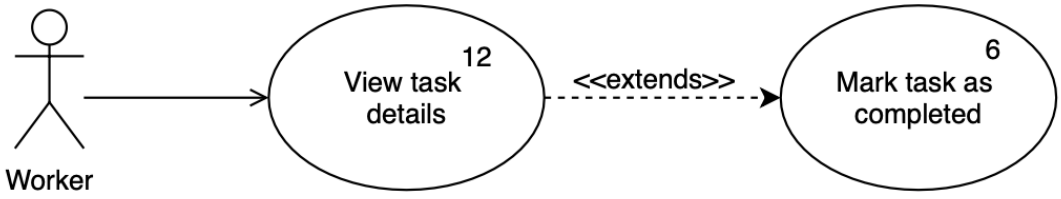


Figura 3: Worker Use Cases

4.3.1 Finish Task

UC-6	**Mark task as completed**
Level	User Goal
Actors	Worker
Pre-conditions	Il worker è autenticato.
Post-conditions	La task completata non appare più nella lista.
Basic Flow	1. Clicca sulla task in corso. 2. Clicca su “Fine” (nella Dashboard del Tecnico, Fig. 9b). 3. Il sistema registra la task nei log.
Alternative Flow	• Errore nel salvataggio: messaggio di errore.
Test	Vedi i test correlati

4.4 Admin Use Cases

Questa sezione illustra i principali casi d’uso disponibili per l’amministratore dell’applicazione.

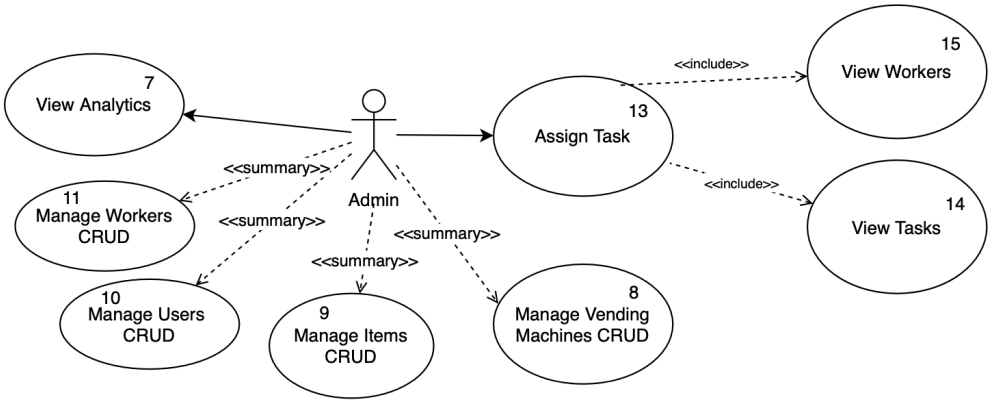


Figura 4: Admin Use Cases

4.4.1 View Analytics

UC-7	**View Analytics**
Level	User Goal
Actors	Admin
Pre-conditions	L'admin è autenticato.
Post-conditions	L'admin vede le analytics degli utenti e degli items.
Basic Flow	1. L'admin accede alla pagina personale (Fig. 8). 2. Clicca su "Analytics". 3. Il sistema carica i dati.
Alternative Flow	3.1 Errore di caricamento: messaggio di errore.
Test	Vedi i test correlati

4.4.2 Create New Vending Machine

UC-8.1	**Create New Vending Machine**
Level	User Goal
Actors	Admin
Pre-conditions	L'admin è autenticato.
Post-conditions	Il sistema contiene una nuova vending machine che appare nelle analytics.
Basic Flow	1. L'admin accede alla pagina di creazione. 2. Compila tutti i campi. 3. Il sistema verifica i dati. 4. Il sistema salva la vending machine nel database.
Alternative Flow	3.1 Campi mancanti o errati: errore. 4.1 Errore nel salvataggio: messaggio e richiesta di riprovare.
Test	Vedi i test correlati

5 Progettazione dell'Interfaccia Utente (Mockups)

Questa sezione presenta la progettazione dell'interfaccia utente (UI) per l'applicazione **JavaBrew**, tramite una serie di mockup. Tali rappresentazioni non si limitano all'aspetto estetico, ma delineano l'architettura dell'interazione uomo-macchina (HCI), specificando i flussi operativi previsti per ciascun ruolo utente. L'obiettivo è fornire una visualizzazione chiara dell'esperienza utente (UX), evidenziando come il design sia stato concepito per supportare le funzionalità del sistema in modo intuitivo ed efficiente.

5.1 Diagramma di Navigazione

Il diagramma di navigazione in Figura 5 illustra il flusso utente attraverso le principali schermate dell'applicazione JavaBrew, differenziando i percorsi in base al ruolo dell'utente (Customer, Worker, Admin).

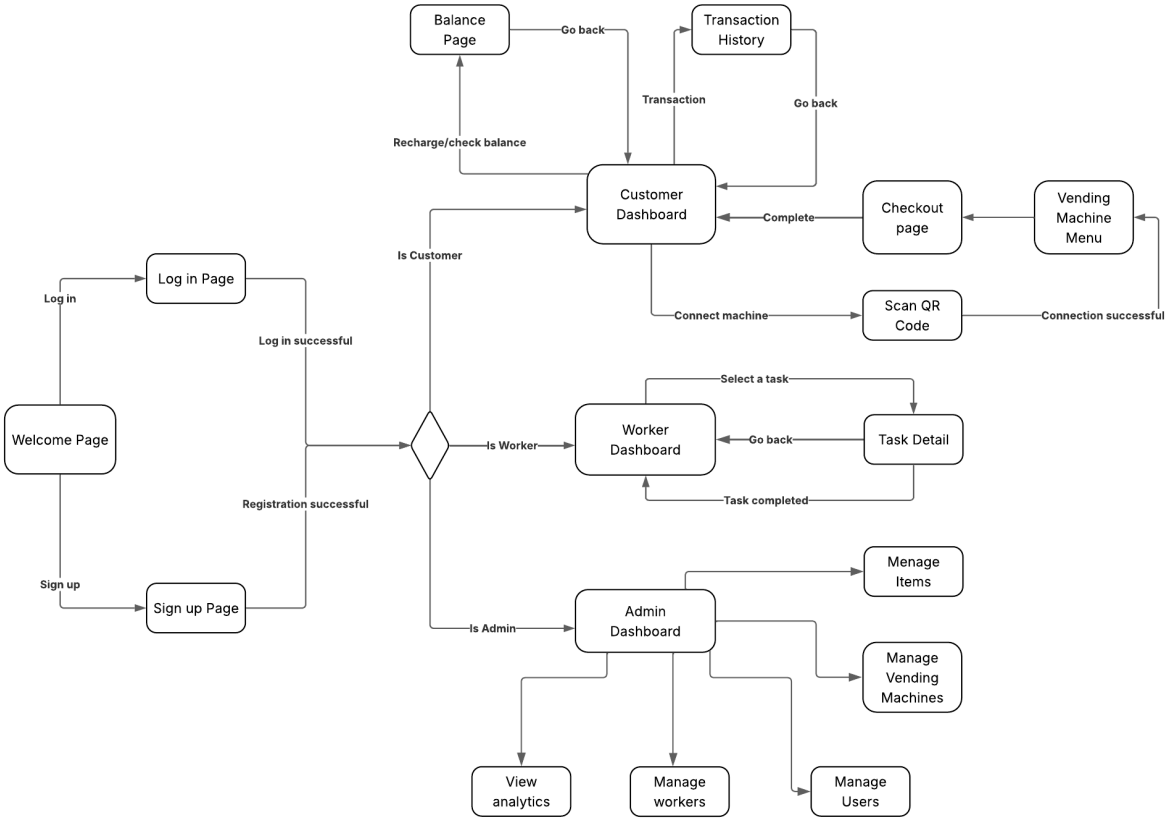


Figura 5: Flusso di navigazione dell'applicazione.

5.2 Flusso di Accesso e Registrazione

Il flusso di autenticazione è articolato in una schermata di login (Figura 6) e un modulo di registrazione (Figura 7). Le interfacce sono caratterizzate da un design essenziale, che richiede unicamente le informazioni strettamente necessarie, facilitando un processo di iscrizione e accesso diretto e privo di complessità superflue.

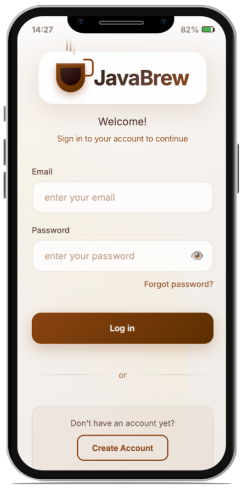


Figura 6: Schermata di login.

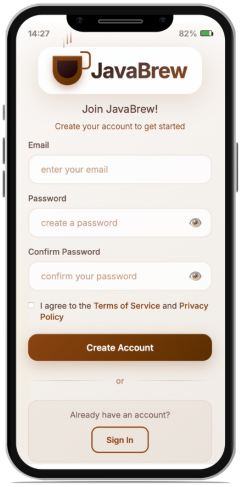


Figura 7: Schermata di registrazione.

5.3 Dashboard per Ruolo Utente

Ogni attore del sistema è dotato di una dashboard personalizzata. La dashboard dell'Amministratore (Figura 8) funge da pannello di controllo, offrendo una visione d'insieme del sistema. Le interfacce dedicate al Cliente e al Tecnico (Figura 9) sono ottimizzate per l'utilizzo su dispositivi mobili.

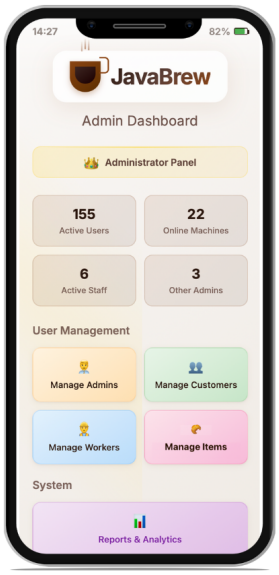
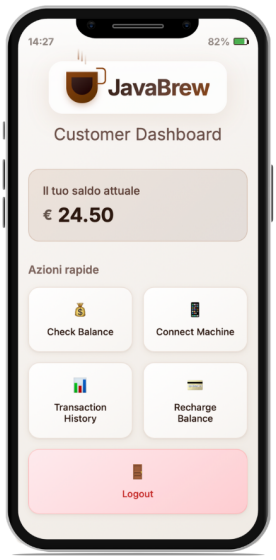
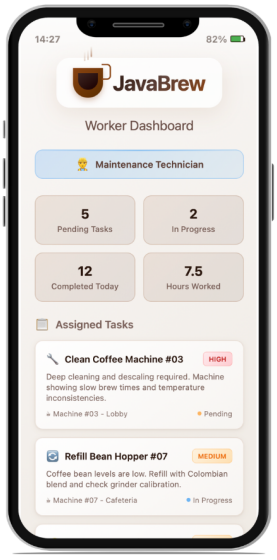


Figura 8: Dashboard dell'Amministratore.



(a) Dashboard del Cliente.



(b) Dashboard del Tecnico.

Figura 9: Dashboard per i ruoli Cliente e Tecnico.

5.4 Catalogo Prodotti e Acquisto

La schermata di selezione dei prodotti (Figura 10) implementa un catalogo digitale interattivo. Il design è studiato per guidare l'utente attraverso un processo di selezione e acquisto dei prodotti.

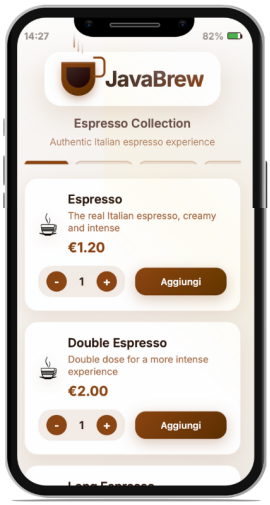


Figura 10: Schermata di selezione dei prodotti.

In sintesi, i mockups presentati illustrano come la progettazione dell'interfaccia utente sia stata definita in base ai requisiti funzionali e alle esigenze specifiche di ciascun attore. Il design delle interfacce si concentra sull'efficienza operativa e sulla fruibilità, aspetti fondamentali per l'aderenza del sistema ai requisiti utente.

6 Architettura del progetto

In Figura 11 è presentata una panoramica dell'architettura a strati del sistema, che offre una rappresentazione visiva completa dell'intero impianto. Nelle sezioni successive verrà esaminato in dettaglio ciascun componente, analizzandone i singoli ruoli e le interazioni reciproche.

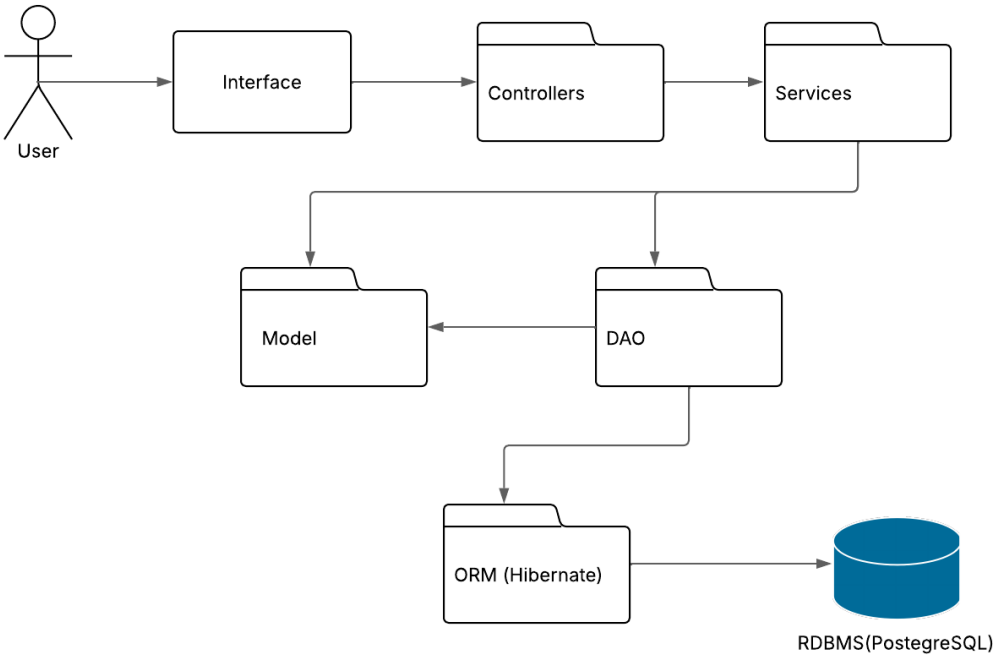


Figura 11: Diagramma dell'architettura a strati del sistema JavaBrew.

6.1 UML diagram

In figura è riportato il diagramma UML completo dell'applicazione. Ogni singolo macro-componente verrà successivamente analizzato nel dettaglio.

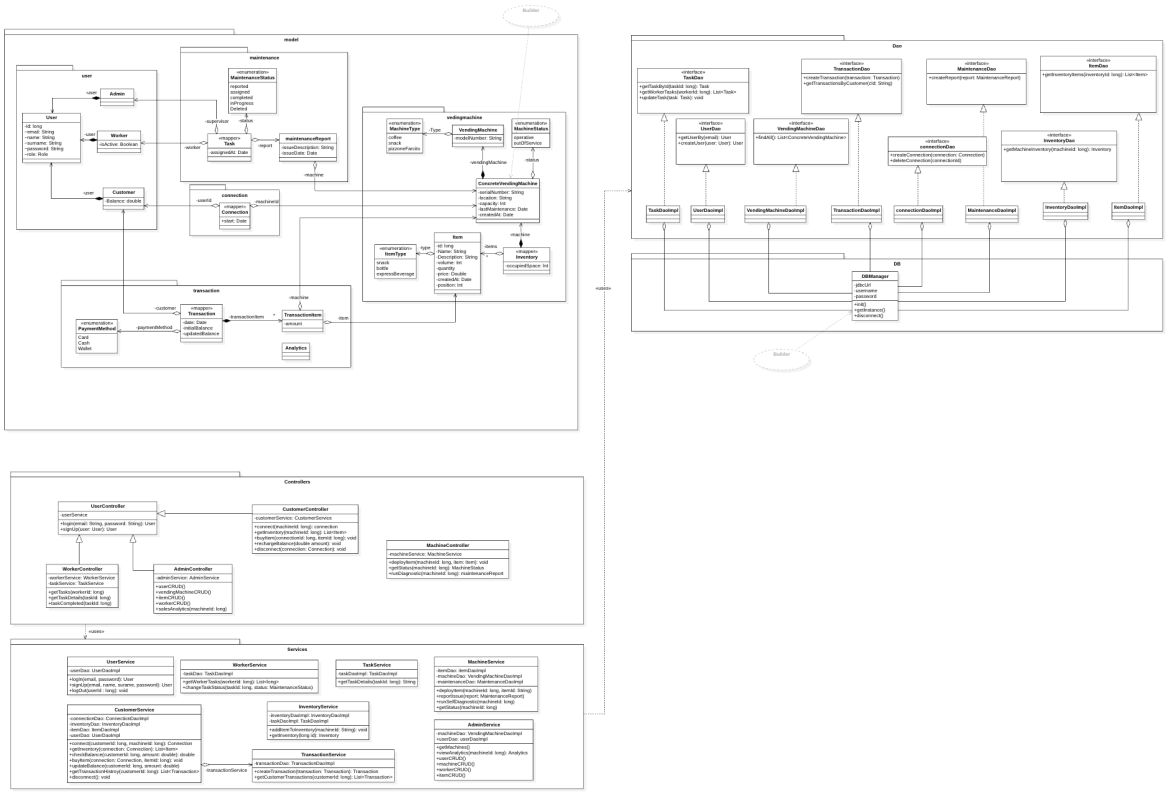


Figura 12: UML diagram

6.2 Struttura del codice

la struttura della directory del progetto è stata organizzata in base alla separazione dei package di Java.

```
|-- Dockerfile
|-- compile.sh
|-- docker-compose.yml
|-- pom.xml
|-- readme.md
|-- src
|   |-- main
|   |   |--java
|   |   |   |--Main.java
|   |   |   |-- controllers
|   |   |   |-- dao
|   |   |   |-- db
|   |   |   |-- model
|   |   |   |-- services
|   |   |-- resources
|   |-- test
|       |--java
|       |   |-- controllers
|       |   |-- dao
|       |   |-- db
|       |   |-- model
|       |   |-- services
|       |-- resources
```

La directory dei test rispecchia la struttura principale per mantenere gli unit test organizzati. L'adozione di queste pratiche garantisce un progetto chiaro, strutturato e facilmente manutenibile, evitando al contempo problemi legati alla visibilità dei package.

Ogni package interagisce con gli altri per svolgere compiti specifici. Questo approccio strutturato assicura che ogni componente del sistema abbia una responsabilità ben definita, rendendo il codice più facile da navigare, comprendere ed estendere. Inoltre, facilita la collaborazione tra i membri del team, poiché ogni package può essere sviluppato e testato in modo indipendente.

6.3 Domain Model

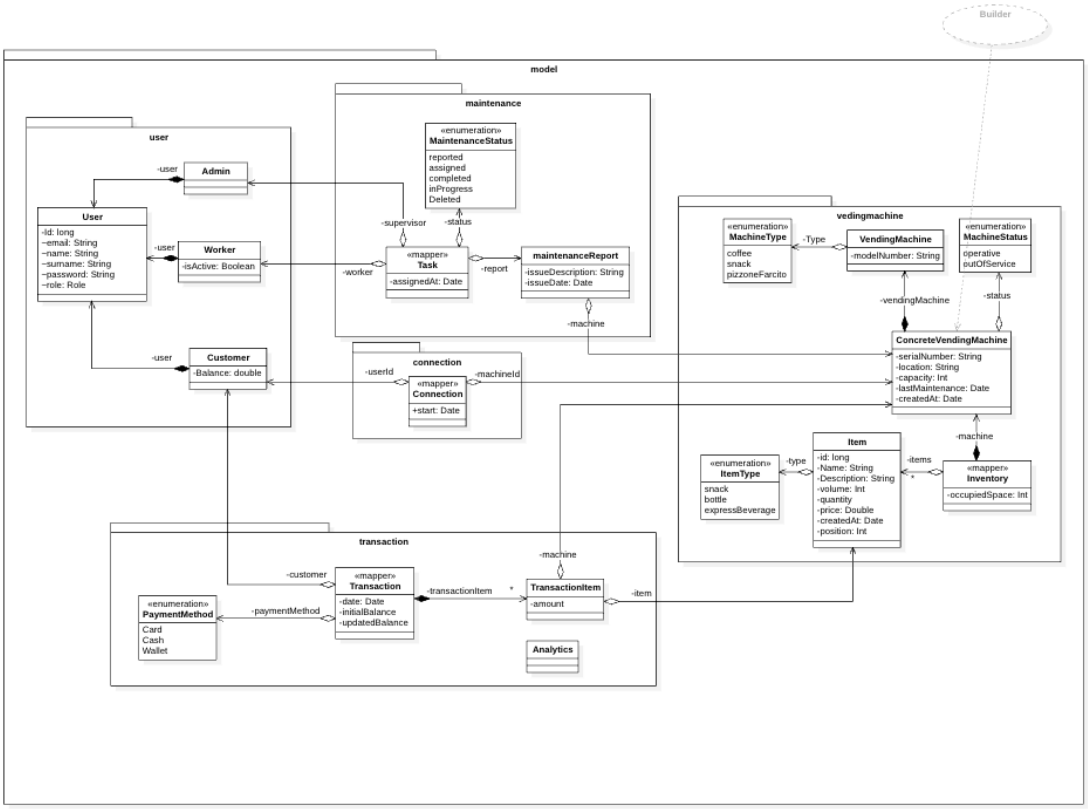


Figura 13: Domain Model

La Fig. 13 illustra le entità chiave e le relazioni all'interno del *domain layer*. In linea con i principi del Domain-Driven Design, ogni classe incapsula invarianti e comportamenti specifici.

ConcreteVendingMachine e il suo Builder

Per la classe `ConcreteVendingMachine` è stato implementato il pattern *Builder*. Tale scelta deriva dal fatto che questa classe presenta numerosi attributi, alcuni obbligatori (come `serialNumber`, `location`, `capacity` e `createdAt`) e altri opzionali (come ad esempio `lastMaintenance`). L'uso del *Builder* consente di definire e validare facilmente tali attributi al momento della creazione.

Questo approccio migliora la leggibilità e la manutenibilità del codice, oltre a garantire che ogni istanza creata rispetti automaticamente tutte le regole e i vincoli definiti dall'architettura del sistema.

```java
public class ConcreteVendingMachine {
    private final String serialNumber;
    private final String location;
    private final int capacity;
    private final MachineStatus status;
    private final Instant createdAt;
    private final Instant lastMaintenance;

    // Optional fields can be added as needed

    private ConcreteVendingMachine(Builder builder) {
        this.serialNumber = builder.serialNumber;
        this.location = builder.location;
        this.status = builder.status != null ? builder.status :
    MachineStatus.Operative;
        this.createdAt = builder.createdAt != null ? builder.
    createdAt : Instant.now();
        this.lastMaintenance = builder.lastMaintenance != null ?
    builder.lastMaintenance : Instant.now();
    }

    ///setter methods

    public static class Builder {
        private final String serialNumber;
        private final String location;
        private final int capacity;
        private MachineStatus status;
        private Instant createdAt;
        private Instant lastMaintenance;

        public Builder(String serialNumber, String location, int
    capacity) {
            this.serialNumber = serialNumber;
            this.location = location;
            this.capacity = capacity;
        }

        //setter methods

        public ConcreteVendingMachine build() {
            return new ConcreteVendingMachine(this);
        }
    }
}
```

Listing 1: Snippet semplificato del codice di `ConcreteVendingMachine` con `Builder` integrato.

Composizione al posto di estensione

Nel nostro modello UML abbiamo scelto di legare le entità tramite *composizione*, anziché tramite estensione, per sottolineare rapporti «parte–tutto» chiari e preservare l'incapsulamento delle responsabilità.

Ad esempio, una `ConcreteVendingMachine` possiede un oggetto `Inventory` per rappresentare il suo magazzino interno: non sarebbe corretto modellare l'inventario come sottoclasse della macchina, poiché esso non è una specializzazione della macchina stessa, bensì un suo componente intrinseco, vincolato al suo ciclo di vita.

Allo stesso modo, una `Transaction` include una o più istanze di `TransactionItem` per tenere traccia dei singoli articoli venduti: ogni elemento esiste solo nel contesto specifico della transazione che lo contiene.

Questo approccio favorisce una progettazione modulare, in cui le parti possono essere sviluppate, testate e manutenute indipendentemente, pur garantendo che la distruzione dell'oggetto «contenitore» determini automaticamente anche l'eliminazione delle sue «parti», evitando così incongruenze o oggetti orfani nel sistema.

Pattern Mapper

Sono presenti diverse classi indicate come *mapper* (ad esempio: `TaskMapper`, `ConnectionMapper`, `InventoryMapper`, `TransactionMapper`). Questo suggerisce che sia stato adottato il pattern *Data Mapper* per separare chiaramente il modello di dominio dalle operazioni di persistenza o di conversione tra i diversi layer applicativi.

I principali vantaggi di questa scelta includono:

- facilità nella manutenzione del codice e nella gestione di cambiamenti strutturali;
- riduzione delle dipendenze tra il modello di dominio e la logica tecnica o di persistenza.

6.4 DAO (Data Access Object)

Il livello Data Access Object (DAO), come illustrato in Figura 14, è stato progettato per astrarre e gestire tutte le operazioni di persistenza dei dati dell'applicazione. Questa architettura garantisce un disaccoppiamento completo tra la logica di business e le specifiche del database sottostante.

Il design del livello DAO si articola in interfacce dedicate (quali `TaskDao`, `UserDao`, `VendingMachineDao`, `TransactionDao`, `ConnectionDao`, `MaintenanceDao`, `InventoryDao`, e `ItemDao`), ciascuna definente i contratti per le operazioni CRUD (Create, Read, Update, Delete) e query specifiche per le rispettive entità del dominio. A ciascuna interfaccia corrisponde una o più implementazioni concrete (es. `TaskDaoImpl`, `UserDaoImpl`), che contengono la logica effettiva di interazione con il sistema di persistenza.

L'interazione con il database è centralizzata tramite il componente `DBManager`, il quale è progettato secondo il pattern Singleton per assicurare un punto di accesso globale e univoco alla connessione al database. Il `DBManager` gestisce i dettagli della connessione e fornisce metodi per l'inizializzazione (`init`), l'ottenimento dell'istanza (`getInstance`) e la disconnessione (`disconnect`). Questa configurazione assicura che i DAO operino su un'unica sorgente di connessione, ottimizzando la gestione delle risorse.

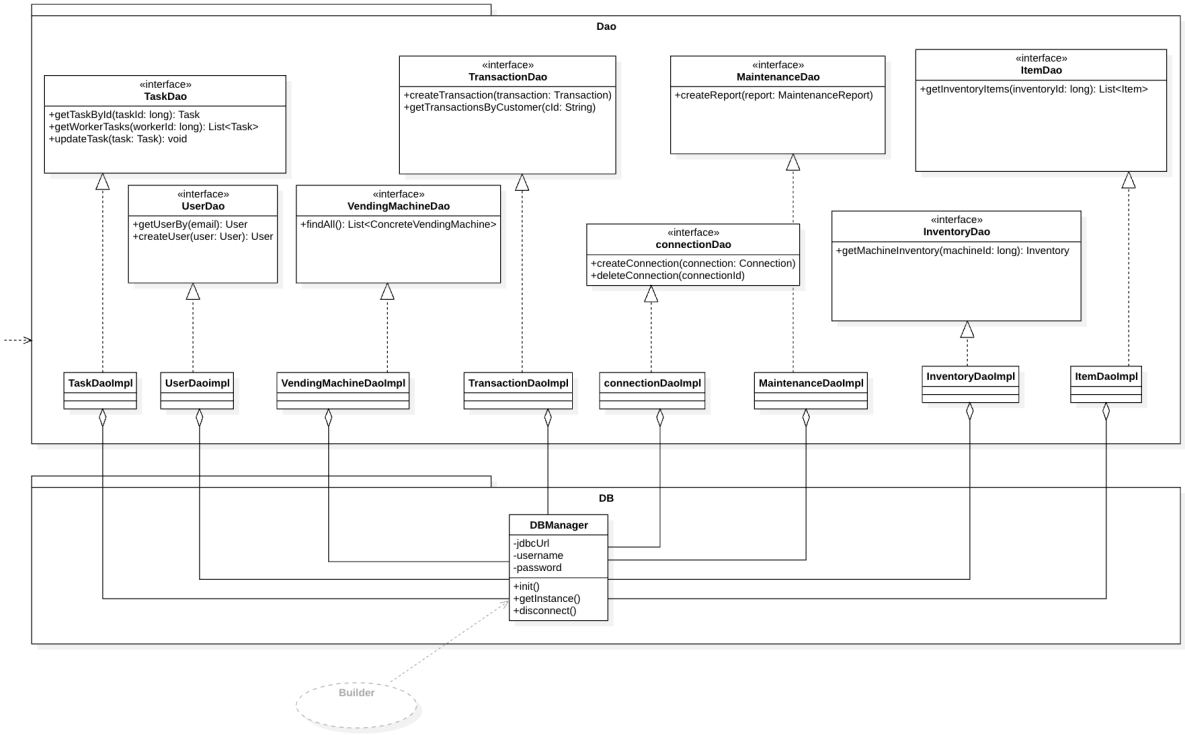


Figura 14: Dao

6.5 Business Logic

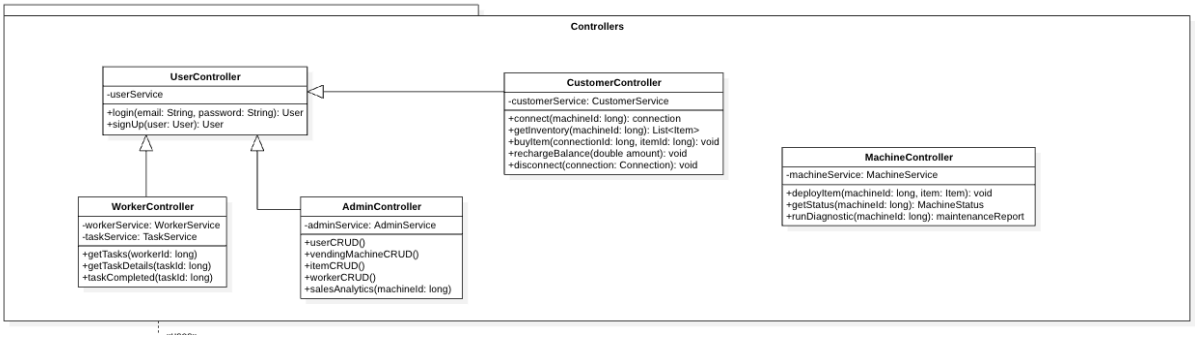


Figura 15: Controllers

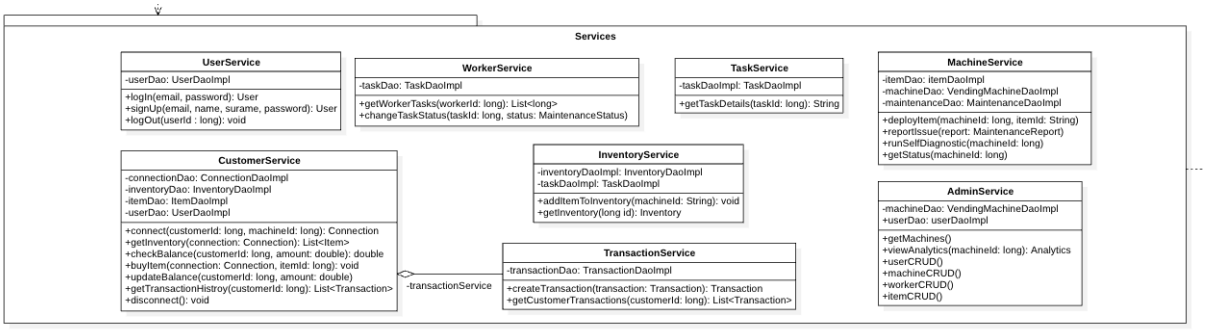


Figura 16: Services

L'implementazione della *business logic* del sistema è stata progettata secondo un approccio modulare, adottando i seguenti principi architetturali e di progettazione:

- **Separazione netta tra Controller e Service:** la logica applicativa è interamente delegata ai `Service`, permettendo ai `Controller` di occuparsi esclusivamente della gestione delle richieste e delle risposte verso il client. Questa separazione migliora significativamente la leggibilità, la manutenibilità e la testabilità del codice.
- **One Controller per Actor:** ogni attore primario del sistema (`Customer`, `Worker`, `Admin`) dispone di un proprio controller dedicato, in accordo con i principi del Domain-Driven Design. Questo approccio assicura una netta separazione delle responsabilità e facilita l'evoluzione indipendente delle funzionalità associate a ciascun attore.

7 Database

La persistenza dei dati nell'applicazione è gestita tramite un database **PostgreSQL** in ambiente di produzione, mentre per i test viene utilizzato un database **in-memory H2**. L'accesso e la mappatura degli oggetti al database avvengono attraverso la tecnologia **JPA** (Jakarta Persistence API), utilizzando **Hibernate** (versione 6.5.2.Final) come implementazione.

Le specifiche tecniche principali sono:

- **ORM:** JPA (Jakarta Persistence API versione 3.2.0)
- **Implementazione:** Hibernate `hibernate-core` versione 6.5.2.Final
- **Database di produzione:** PostgreSQL, con driver `postgresql` versione 42.7.3
- **Database di test:** H2
- **Gestione delle transazioni:** Jakarta Transaction API (`jakarta.transaction-api` versione 2.0.1)

Lo schema del database è stato derivato automaticamente dal domain model mediante l'Object-Relational Mapping (ORM), in modo da mantenere coerenza tra la struttura delle classi e le tabelle relazionali persistite nel database. Il database viene inizializzato automaticamente con l'avvio del container Docker.

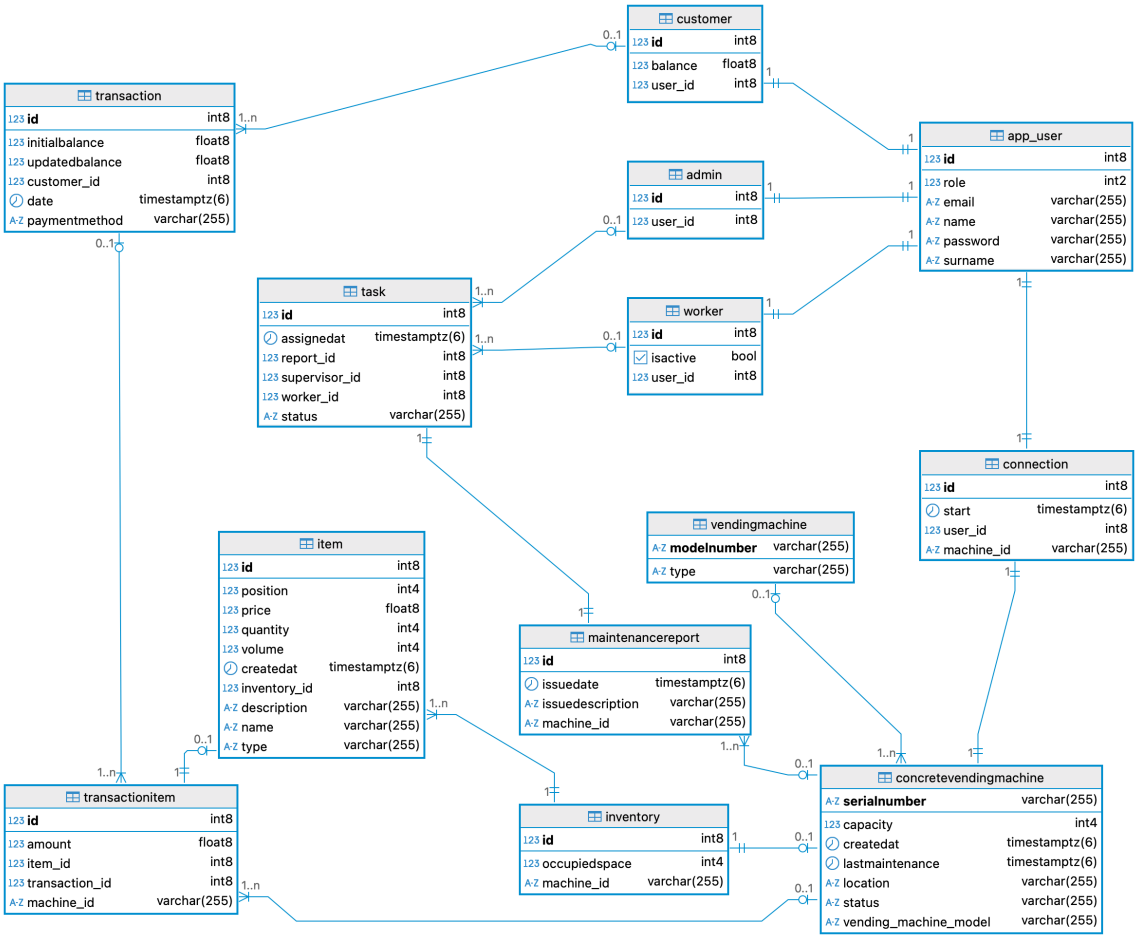


Figura 17: Schema E-R del database

7.1 Schema del Database

7.1.1 Gestione degli Utenti

Sono presenti 3 tipologie di utente:

- `admin`: Gestisce l'intera applicazione, con accesso completo a tutte le funzionalità.
- `worker`: Tecnico che gestisce le operazioni di manutenzione e assistenza.
- `customer`: Utente finale che interagisce con le macchine per acquistare prodotti.

La tabella `app_user` contiene le informazioni comuni a tutti gli utenti, come `username`, `password`, `email` e il tipo di utente (`role`). Le tabelle `admin`, `worker` e `customer` estendono `app_user` con attributi specifici necessari per ciascun ruolo. Questa separazione consente di gestire il login degli utenti in modo più efficiente, mantenendo le informazioni comuni in un'unica tabella e le specifiche in tabelle separate.

7.1.2 Gestione delle Macchine e dell'Inventario

Le vending machine sono rappresentate da un modello generico e da istanze specifiche:

- `vendingmachine`: Definisce i modelli generici (es. `modelnumber`, `type`).
- `concretevendingmachine`: Rappresenta un'istanza fisica (es. `serialnumber`, `location`, `status`, `capacity`), collegata a un modello generico (`vending_machine_model`).
- `inventory`: Ogni macchina concreta ha un inventario (relazione uno-a-uno) che traccia lo spazio occupato (`occupiedspace`) e collega gli articoli.
- `item`: Elenca i prodotti disponibili (es. `name`, `price`, `quantity`, `position`), collegati all'inventario di una macchina specifica (`inventory_id`).

7.1.3 Gestione delle Transazioni

Le transazioni degli utenti sono gestite attraverso due tabelle:

- `transaction`: Intestazione della transazione (es. `customer_id`, `paymentmethod`, `initialbalance`, `updatedbalance`).
- `transactionitem`: Dettaglio della transazione, realizzando una relazione molti-a-molti tra transazioni e articoli. Ogni record collega un `transaction_id` a un `item_id` e specifica l'importo (`amount`).

7.1.4 Gestione Connessioni

Supporta la comunicazione tra clienti e macchine:

- `connection`: Traccia le connessioni attive tra utenti e macchina (es. `user_id`, `machine_id`, `start` istante in cui inizia la connessione).

La connessione avviene tra un `user` e una `concretevendingmachine` (invece che tra un `customer` e una `vendingmachine`), in modo da rendere possibile in un futuro sviluppo dell'applicazione la gestione di feature come manutenzione remota (es. sblocco di un prodotto rimasto bloccato) da parte del tecnico o dell'admin.

8 Testing

8.1 Strategia di Testing

Il software è stato validato attraverso **test di unità** e **test di integrazione**. Questo approccio a livelli è essenziale per verificare il sistema da diverse prospettive: la correttezza logica dei singoli componenti e la loro corretta interazione all'interno dell'architettura. Gli strumenti configurati nel file `pom.xml` hanno permesso di automatizzare e validare il funzionamento del codice a vari livelli.

8.1.1 Test di Unità e Mocking

I test di unità sono stati sviluppati con **JUnit 5** per verificare l'isolamento e la correttezza funzionale dei singoli componenti. Per testare il livello Service in assenza di dipendenza dal livello di persistenza, è stato utilizzato il framework di mocking **Mockito**. Questo ha consentito di simulare il comportamento dei DAO (tramite l'annotazione `@Mock`) e di iniettarli nel servizio sotto test (tramite `@InjectMocks`), focalizzando l'analisi esclusivamente sulla logica di business. Tale isolamento è cruciale per l'esecuzione rapida dei test (centinaia di test in pochi secondi, senza latenza del database) e per la verifica deterministica di ogni percorso logico e caso limite del servizio.

```
1  public class CustomerServiceTest {
2      @Mock private CustomerDaoImpl customerDao;
3      @Mock private ItemDaoImpl itemDao;
4      @Mock private TransactionService transactionService;
5      @InjectMocks private CustomerService customerService;
6
7      private Customer mockCustomer;
8      private Item mockItem1;
9
10     @BeforeEach
```

```
11      public void setUp() {
12          MockitoAnnotations.openMocks(this);
13          mockCustomer = new Customer(2L, new User(1L, "...", "Jhon", "
    Doe", "..."), 100.0);
14          mockItem1 = new Item(30L, "Soda", "...", 1, 10, 1.50, 3,
    ItemType.Bottle);
15      }
16
17      @Test
18      @DisplayName("Test buyItem method success with single item")
19      void testBuyItemSuccessSingleItem() {
20          // Arrange: Prepara i dati e il comportamento dei mock
21          when(itemDao.getItemById(30L)).thenReturn(mockItem1);
22          when(transactionService.createTransaction(any(Transaction.
    class))).thenReturn(expectedTransaction);
23
24          // Act: Esegui il metodo da testare
25          Transaction result = customerService.buyItem(1L, List.of(30L)
    );
26
27          // Assert: Verifica il risultato e le interazioni con i mock
28          assertNotNull(result);
29          assertEquals(9, mockItem1.getQuantity());
30          verify(itemDao, times(1)).updateItem(mockItem1);
31          verify(customerDao, times(1)).updateCustomer(mockCustomer);
32          verify(transactionService, times(1)).createTransaction(any(
    Transaction.class));
33      }
34  }
```

Listing 2: Esempio semplificato di test unitario per il metodo `buyItem` con JUnit 5 e Mockito.

8.1.2 Test di Integrazione

I test di integrazione sono stati implementati per verificare la corretta interazione tra i vari livelli dell'applicazione, in particolare tra il Service Layer, il DAO Layer e il database. Per questi test, è stato impiegato un database in-memory **H2**, che ha permesso l'esecuzione dell'intero stack di persistenza in un ambiente controllato e ad alta velocità. Questi test sono fondamentali per identificare problematiche non rilevabili dai test di unità, quali errori nelle annotazioni di mapping JPA e gestione delle transazioni. La validazione di tali interazioni in un ambiente controllato, prima del deployment, costituisce un passaggio critico per la robustezza del sistema.

8.1.3 Analisi della Copertura del Codice

Per monitorare l'efficacia e la completezza della suite di test, è stato integrato il plugin **JaCoCo** in Maven. L'esecuzione dei test di unità e di integrazione ha permesso di ottenere una **copertura del codice superiore all'80%**. L'analisi della copertura, inoltre, supporta l'identificazione di "codice morto" (porzioni di codice mai eseguite) e assicura che i percorsi logici critici, inclusi quelli relativi alla gestione degli errori, siano stati adeguatamente testati, contribuendo al miglioramento continuo della qualità.

smartVend

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.smartvend.app		0%		0%	28	28	144	144	7	7	1	1
com.smartvend.app.dao.impl		82%		36%	30	97	91	434	2	60	0	11
com.smartvend.app.services		90%		88%	31	201	37	394	6	66	1	9
com.smartvend.app.controllers		81%		80%	7	34	9	56	4	21	0	4
com.smartvend.app.db		57%		83%	3	8	8	20	2	5	0	1
Total	1.110 of 3.937	71%	118 of 411	71%	99	368	289	1.048	21	159	2	26

Figura 18: Riepilogo della copertura del codice per package (generato da JaCoCo).

8.2 Supporto di Intelligenza Artificiale (IA) nello Sviluppo Software

Durante il processo di sviluppo del progetto sono stati usati vari strumenti basati su intelligenza artificiale, mirati a velocizzare il lavoro. Sono stati usati i seguenti strumenti GitHub Copilot (per il suggerimento in-line del codice), GPT-4.1, Claude 4 Sonnet e Gemini 2.5.

GitHub Copilot è stato impiegato per l'automazione di compiti ripetitivi e a basso livello, come la generazione di *boilerplate code* e il supporto al *refactoring*.

Parallelamente, è stata adottata una **strategia di specializzazione degli LLM**, assegnando a ciascun modello il compito per cui, durante il lavoro, ha dimostrato un'efficacia maggiore. La ripartizione dei compiti è stata la seguente:

- **GPT-4.1 per l'Analisi Visuale:** Durate le fasi iniziali di progettazione dell'architettura, abbiamo provato a usare GPT-4.1 per analizzare il nostro UML e porre alcuni dubbi riguardo a determinate scelte. Purtroppo però non era in grado di comprendere a pieno il significato delle varie relazioni tra i vari componenti. Una soluzione parzialmente efficace è stata convertire il diagramma in un formato testuale ***Mermaid*** che ha consentito di ricevere risposte migliori, ma spesso non corrette.
- **Claude 4 Sonnet come Assistente alla Programmazione:** Per la scrittura e l'ottimizzazione del codice. Non è stato impiegato per generare intere funzionalità, ma piuttosto come un "co-pilota" per aiutarci a superare blocchi implementativi e ottimizzare frammenti di codice specifici. Questo approccio ci ha permesso di velocizzare lo sviluppo mantenendo il pieno controllo sulla qualità e l'architettura del software.
- **Gemini come Analista di Progetto:** Abbiamo scoperto che il ruolo più efficace per Gemini nel nostro workflow era quello di analista. Abbiamo osservato che era particolarmente rapido nel sintetizzare informazioni provenienti da fonti diverse, come la nostra

documentazione interna e articoli tecnici esterni, fornendo risposte chiare a quesiti ingegneristici. La sua velocità lo ha reso anche uno strumento eccellente per validare al volo idee architetturali durante le sessioni di brainstorming.

L'adozione di tale approccio, avvenuta sempre sotto stretta supervisione umana, ha permesso di sfruttare al meglio le capacità di ciascuna IA.

8.3 Test dei Casi d'Uso

Questa sezione elenca i test scritti per ciascun caso d'uso principale. Per ogni caso d'uso, vengono identificati e numerati i test che contribuiscono alla sua verifica.

4.1.1 User Login

1. `UserServiceTest.loginWithValidCredentials`
2. `UserControllerTest.login_ValidCredentials_CallsUserServiceAndReturnsUser`
3. `UserServiceTest.loginWithInvalidPassword`
4. `UserServiceTest.loginWithNonExistingEmail`
5. `UserControllerTest.login_NullEmail_ThrowsException`
6. `UserControllerTest.login_EmptyEmail_ThrowsException`
7. `UserControllerTest.login_NullPassword_ThrowsException`
8. `UserControllerTest.login_EmptyPassword_ThrowsException`

4.1.2 User Sign up

1. `UserServiceTest.signUpWithValidData`
2. `UserControllerTest.signUp_ValidUser_CallsUserServiceAndReturnsUser`
3. `UserServiceTest.signUpWithExistingEmail`
4. `UserServiceTest.signUpWithMissingEmail`
5. `UserControllerTest.signUp_NullUser_ThrowsException`

4.2.1 Buy Item

1. `CustomerServiceTest.testBuyItemSuccessSingleItem`
2. `CustomerServiceTest.testBuyItemSuccessMultipleItems`
3. `CustomerControllerTest.testBuyItem`
4. `CustomerServiceTest.testBuyItemInsufficientBalance`
5. `CustomerServiceTest.testBuyItemOutOfStock`
6. `CustomerServiceTest.testBuyItemItemNotFound`
7. `CustomerServiceTest.testBuyItemConnectionNotFound`
8. `CustomerServiceTest.testBuyItemCustomerNotFound`

4.2.2 Connect to Vending Machine

1. `CustomerServiceTest.testConnectSuccess`
2. `ConnectionDaoImplTest.unit_createConnection_persistsConnection`
3. `ConnectionDaoImplTest.integration_CRUD_flow` (parte CREATE)
4. `CustomerControllerTest.testConnect`
5. `CustomerServiceTest.testConnectCustomerNotFound`
6. `CustomerServiceTest.testConnectMachineNotFound`
7. `ConnectionDaoImplTest.unit_createConnection_throwsIfUserNotFound`
8. `ConnectionDaoImplTest.unit_createConnection_throwsIfMachineNotFound`

4.2.3 Recharge Balance

1. `CustomerServiceTest.testUpdateBalanceSuccess`
2. `CustomerControllerTest.testRechargeBalance`
3. `TransactionServiceTest.testCreateTransactionValid`
4. `TransactionDaoImplTest.integration_CRUD_flow` (parte CREATE)
5. `CustomerServiceTest.testUpdateBalanceUserNotFound`
6. `CustomerServiceTest.testUpdateBalanceInsufficientBalance`
7. `TransactionServiceTest.testCreateTransactionInvalid`

4.3.1 Finish Task

1. `WorkerServiceTest.testChangeTaskStatusSuccess`
2. `WorkerControllerTest.taskCompleted_returnsTrue_whenTaskIsCompletedSuccessfully`
3. `TaskDaoImplTest.updateTask_mergesTask`
4. `TaskDaoImplTest.integration_CRUD_flow` (parte UPDATE)
5. `WorkerServiceTest.testChangeTaskStatusWithTaskNotFound`
6. `WorkerServiceTest.testChangeTaskStatusWithNullStatus`
7. `WorkerServiceTest.testChangeTaskStatusWithCompletedTask`
8. `WorkerControllerTest.taskCompleted_returnsFalse_whenTaskCompletionFails`
9. `WorkerControllerTest.taskCompleted_throwsException_whenTaskIdIsInvalid`

4.4.1 View Analytics

1. `AdminServiceTest.testGetUsersReturnsList`
2. `AdminServiceTest.testGetCustomersReturnsList`
3. `AdminServiceTest.testGetMachinesReturnsList`
4. `AdminServiceTest.testGetWorkersReturnsList`
5. `UserDaoImplTest.findAll_empty`
6. `CustomerDaoImplTest.findAll_returnsListWithCustomers`
7. `ConcreteVendingMachineDaoImplTest.findAll_oneItem`
8. `ItemDaoImplTest.getInventoryItems_returnsList`
9. `AdminServiceTest.testGetUsersThrowsIfDaoNull`
10. `AdminServiceTest.testGetCustomersThrowsIfDaoNull`
11. `AdminServiceTest.testGetMachinesThrowsIfDaoNull`
12. `AdminServiceTest.testGetWorkersThrowsIfDaoNull`
13. `UserDaoImplTest.findAll`

4.4.2 Create New Vending Machine

1. `AdminServiceTest.testCreateMachineReturnsCreatedMachine`
2. `ConcreteVendingMachineDaoImplTest.createMachine_persists`
3. `ConcreteVendingMachineDaoImplTest.integration_CRUD_andQueries`
4. `AdminServiceTest.testCreateMachineThrowsIfMachineNull`
5. `AdminServiceTest.testCreateMachineThrowsIfDaoNull`

8.4 Codebase

Il codice sorgente del progetto è disponibile pubblicamente su GitHub. È possibile consultarlo, scaricarlo ed eventualmente contribuire allo sviluppo al seguente indirizzo:
https://github.com/matteominin/smartVend