Final Project - A C2C Second-hand Trading Platform Database

0. Team Infomation

Name	Student ID	Department/Year
邱元廣	113550025	CS/117
謝亞序	113550109	CS/117
陳泓淯	113550015	CS/117

1. Introduction

Our application, Second-Hand Item Trading & Exchange Platform, is a web-based C2C (Consumer-to-Consumer) marketplace designed to address the issue of resource waste and the high cost of new goods for students. The system allows users to list items for sale or exchange, manage personal wishlists, and communicate directly through an integrated messaging system. The core functions include secure user authentication, item listing with image uploads, robust search/filtering, and a structured transaction management system to ensure clear trade status tracking.

2. Design Motivation

- **Problem addressed:** Students often have useful items (textbooks, furniture) they no longer need, while others seek these items at lower prices. Existing social media groups lack structured searching, transaction history, and the needed in messaging.
- **Suitability:** Our system provides a centralized database that stores items, tracks availability in real-time, and archives conversation history between specific buyers and sellers, making the process more efficient than unstructured platforms.
- **Need for a database:** A database is essential to maintain data persistence (user accounts, item details), referential integrity (linking transactions to specific users and items), and concurrency control (preventing two users from buying the same item simultaneously).

3. Database Design

users

Column Name	Data Type	Constraints
user_id	SERIAL	PRIMARY KEY
username	VARCHAR(50)	NOT NULL, UNIQUE
email	VARCHAR(100)	NOT NULL, UNIQUE
password_hash	VARCHAR(255)	NOT NULL

Column Name	Data Type	Constraints
address	VARCHAR(255)	
is_active	BOOLEAN	DEFAULT TRUE
join_date	TIMESTAMP WITH TIME ZONE	DEFAULT CURRENT_TIMESTAMP

phones

Column Name	Data Type	Constraints
phone_id	SERIAL	PRIMARY KEY
user_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
phone_number	VARCHAR(20)	NOT NULL

items

Column Name	Data Type	Constraints
item_id	SERIAL	PRIMARY KEY
title	VARCHAR(100)	NOT NULL
description	TEXT	
condition	VARCHAR(50)	NOT NULL
owner_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
post_date	TIMESTAMP WITH TIME ZONE	DEFAULT CURRENT_TIMESTAMP, INDEX
price	INTEGER	INDEX
exchange_type	BOOLEAN	DEFAULT FALSE
status	BOOLEAN	DEFAULT TRUE
desired_item	VARCHAR(100)	
category	INTEGER	NOT NULL, REFERENCES categories(category_id), INDEX
total_images	INTEGER	DEFAULT 0

item_images

Column Name	Data Type	Constraints
image_id	SERIAL	PRIMARY KEY
item_id	INTEGER	NOT NULL, REFERENCES items(item_id), INDEX

Column Name	Data Type	Constraints
image_data_name	VARCHAR(255)	NOT NULL

wishlist

Column Name	Data Type	Constraints
wishlist_id	SERIAL	PRIMARY KEY
user_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
item_id	INTEGER	NOT NULL, REFERENCES items(item_id), INDEX
added_date	TIMESTAMP WITH TIME ZONE	DEFAULT CURRENT_TIMESTAMP

transactions

Column Name	Data Type	Constraints
transaction_id	SERIAL	PRIMARY KEY
item_id	INTEGER	NOT NULL, REFERENCES items(item_id), INDEX
buyer_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
seller_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
transaction_date	TIMESTAMP WITH TIME ZONE	DEFAULT CURRENT_TIMESTAMP, INDEX
status	VARCHAR(50)	DEFAULT 'pending'
completion_date	TIMESTAMP WITH TIME ZONE	

messages

Column Name	Data Type	Constraints
message_id	SERIAL	PRIMARY KEY
sender_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
receiver_id	INTEGER	NOT NULL, REFERENCES users(user_id), INDEX
content	TEXT	NOT NULL
sent_at	TIMESTAMP WITH TIME ZONE	DEFAULT CURRENT_TIMESTAMP, INDEX
is_read	BOOLEAN	DEFAULT FALSE
item_id	INTEGER	REFERENCES items(item_id), INDEX

Schema Changes from Milestone 2:

1. Modified `category` table

Table: `categories`

Change Type: `Modified`

Rationale: Transitioned to a dynamic enumeration logic to reduce operational friction. Missing categories are now automatically handled by the backend during item creation, ensuring referential integrity and BCNF compliance without administrative overhead.

2. change `item_images` table

Table: `item_images`

Change Type: `Modify`

Rationale: We changed the `image_data` column to `image_data_name` to store the image file names instead of binary data. This change improves database performance and scalability by offloading image storage to a dedicated file storage system or CDN, reducing database size and improving query performance.

Index Design Rationale

We added indexes to specific columns to optimize query performance based on the application's access patterns defined in the API routes:

1. **Foreign Keys**: Columns like `owner_id`, `category`, `item_id`, `user_id`, `buyer_id`, `seller_id`, `sender_id`, and `receiver_id` are frequently used in `JOIN` operations and `WHERE` clauses to link related data (e.g., finding all items by a user, getting messages for a conversation). Indexing these columns speeds up these lookups significantly.
2. **Sorting**: Columns `price`, `post_date`, `transaction_date`, and `sent_at` are used in `ORDER BY` clauses (e.g., sorting items by price or date, listing transactions or messages chronologically). Indexes on these columns allow the database to retrieve sorted results efficiently without performing a full table scan and sort.
3. **Filtering**: While `status` and `exchange_type` are also used for filtering, they have low cardinality (few unique values), so indexes might be less effective depending on data distribution. However, columns like `price` and dates have high cardinality and are good candidates for indexing to speed up range queries or specific value lookups.

Design Justification: Normalization and Performance

We have chosen to analyze our database design through the lens of **Normalization (BCNF)** and its trade-offs with performance.

BCNF Compliance

Our database schema is largely designed to adhere to **Boyce-Codd Normal Form (BCNF)**.

- **Multi-valued Attributes**: We handled multi-valued attributes like phone numbers and images by creating separate tables (`phones`, `item_images`) linked by foreign keys, rather than storing them as delimited strings or arrays, satisfying First Normal Form (1NF) and enabling proper indexing and querying.

Performance Trade-offs and Denormalization

While strict normalization eliminates redundancy, we introduced a deliberate **denormalization** in the `items` table for performance reasons:

- **The `total_images` Attribute**: The `items` table contains a `total_images` column. Strictly speaking, this is a derived attribute dependent on the count of related rows in the `item_images` table. Storing this violates BCNF because it introduces redundancy; the information is already available by querying `item_images`.
- **Justification**: This trade-off was made to optimize the **Read** performance of the item feed. The "browse items" page is the most frequently accessed part of the application. By storing `total_images`, we can display the number of images on the product card (e.g., "iPhone 13 (5 images)") using a simple `SELECT` from the `items` table, avoiding a costly `JOIN` and `GROUP BY/COUNT` operation on the `item_images` table for every single item in the list. This significantly reduces the database load for read-heavy workloads, which is a standard practice discussed in performance tuning lectures when the cost of maintaining consistency (updating `total_images` on insert/delete) is outweighed by the read efficiency.

4. Data Source

Because there is no publicly available dataset that fits our specific schema and requirements, we generated synthetic data using **Python scripts**. These scripts create user profiles, item listings, categories, transactions, and messages to populate our database for testing and demonstration purposes. The data generation process ensures that the relationships between entities are maintained according to our database design.

5. Application with Database

5.1 Target Users & Design Customization

The primary target users are **University Students** who want to trade second-hand items. The system is designed to support a dual-role model where every user can act as both a buyer and a seller seamlessly.

- **Buyers**:
  - **Needs**: Efficiently find specific items (textbooks, electronics) at low prices and ensure the seller is responsive.
  - **Design Customization**: We implemented a **Search and Filter Bar** (by keyword, price, category) on the main page to facilitate quick discovery. A **Wishlist** view allows buyers to track items they are interested in.
- **Sellers**:
  - **Needs**: Easily list items and manage incoming purchase requests.
  - **Design Customization**: An **Upload Interface** simplifies the listing process. The **Transaction Dashboard** distinguishes between "Sales" (where the user is the seller) and "Purchases" (where the user is the buyer), allowing sellers to manage order status (e.g., marking a transaction as completed).

5.2 Functionalities

1. **User Authentication**: Register, Login, and Profile Management (update address, phone).
2. **Marketplace**: Browse items, Search by keyword, Filter by category, Sort by price/date.

3. **Item Management**: Post new items with images, Edit item details, Delete items.
4. **Transaction System**: Initiate purchase requests, View transaction history, Update transaction status (Pending -> Completed).
5. **Messaging**: Send and receive messages related to specific items/transactions.
6. **Wishlist**: Add/Remove items from a personal wishlist.

5.3 SQL Queries

Our application uses **Raw SQL** via SQLAlchemy's `text()` construct for all database operations. Below are the key queries used:

User Management

- **Create User**:

```
-- Insert core user data
INSERT INTO users (username, email, password_hash, address, is_active,
join_date)
VALUES (:username, :email, :password_hash, :address, true, now())
RETURNING user_id
```

- **Get User Profile**:

```
-- Fetch basic info
SELECT * FROM users WHERE user_id = :user_id
```

- **Update User**:

```
-- Update core attributes
UPDATE users SET email = :email WHERE user_id = :id
UPDATE users SET address = :address WHERE user_id = :id
```

Item Operations

- **List Items (with Search & Sort)**:

```
-- Market feed with dynamic filtering and ordering
SELECT * FROM items WHERE status = true
AND (title LIKE :search OR description LIKE :search)
ORDER BY price ASC  -- or DESC, or post_date
```

- **Create Item**:

```
-- Insert item and record its post_date
INSERT INTO items (title, description, condition, owner_id, price,
exchange_type, status, desired_item, category, total_images, post_date)
VALUES (:title, :description, :condition, :owner_id, :price, :exchange_type,
true, :desired_item, :category, :total_images, now())
RETURNING item_id, post_date
```

- **Update Item**:

```
UPDATE items
SET title = :title, description = :description, condition = :condition,
    price = :price, exchange_type = :exchange_type, desired_item =
:desired_item
WHERE item_id = :item_id
```

- **Delete Item**:

```
DELETE FROM item_images WHERE item_id = :item_id;
DELETE FROM items WHERE item_id = :item_id;
```

Transactions

- **Create Transaction**:

```
INSERT INTO transactions (item_id, buyer_id, seller_id, transaction_date,
status)
VALUES (:item_id, :buyer_id, :seller_id, now(), 'pending')
RETURNING transaction_id, transaction_date
```

- **Get User Transactions**:

```
SELECT * FROM transactions
WHERE buyer_id = :user_id OR seller_id = :user_id
ORDER BY transaction_date DESC
```

- **Update Transaction Status**:

```
UPDATE transactions
SET status = :status, completion_date = :cdate
WHERE transaction_id = :tid
RETURNING *
```

Messaging

- **Get Conversation List**:

```
-- Fetch unique chat partners with their latest item context
SELECT DISTINCT ON (u.user_id)
    u.user_id, u.username,
    i.item_id, i.title as item_title,
    img.image_data_name as item_image
FROM users u
JOIN messages m ON (u.user_id = m.sender_id OR u.user_id = m.receiver_id)
LEFT JOIN items i ON m.item_id = i.item_id
LEFT JOIN item_images img ON i.item_id = img.item_id
WHERE (m.sender_id = :user_id OR m.receiver_id = :user_id)
  AND u.user_id != :user_id
```

5.4 Database Connection

The application connects to the database using **FastAPI** and **SQLAlchemy**.

1. **Connection Engine**: We use `sqlalchemy.create_engine` to establish a connection pool to the PostgreSQL database. The connection string is constructed from environment variables (User, Password, DB Name).
2. **Session Management**: We implemented a `get_db()` dependency function that yields a database session (`SessionLocal`) for each HTTP request. This ensures that each request has its own isolated transaction scope and the connection is properly closed after the request is processed.
3. **Execution**: Although we use SQLAlchemy for connection management, we strictly use `db.execute(text("SQL..."))` to run the raw SQL queries listed above, bypassing the ORM layer to demonstrate direct SQL usage.

5.5 Deployment

We deployed our frontend Svelte application on **Cloudflare Pages**, accessible at `db.trashcode.dev`. The backend is hosted on a **Raspberry Pi** located in a dormitory. To expose the backend to the public internet and bypass NAT restrictions, we utilized **Cloudflare Tunnel**, making the API accessible at `db_api.trashcode.dev`. We chose to use distinct subdomains for the frontend and backend instead of path-based routing to fully leverage **Cloudflare's CDN** capabilities for improved load times and performance.