CS 348 Project: PocketTrader

Milestone 3: Final Report

[Link to GitHub Repo]

All SQL queries, views, triggers, and indices can be found in our Github Repo.

R1 - Application high-level description

PocketTrader is a small inventory/collection manager for TCG (trading card game) collectors. The app maintains a catalog of cards (cardID, name, pack/set, rarity, type, imageURL, etc.) and allows authenticated users browse and filter the catalog, add one or more copies of a card to their personal collection, add cards to their public wishlist, and have the opportunity to find trades with other uses (i.e. identify what cards are on different users' collections and wishlist, then arrange a suitable trade, such that one or more players get cards on their wishlist). Typical users are players and collectors who want to track what they own and quickly find cards by pack, rarity, or type. Administrators are the app maintainers (us), who seed and manage the database.

As of milestone 1, we have implemented the ability to view/filter cards, and to add cards to the user's personal collection.

R2 - System support description

The frontend is built with Next.js (React) and Tailwind CSS, served on port 3000. The backend is a Flask (Python) app that exposes REST endpoints (ports: backend 5000) and executes SQL queries stored in app/backend/sql/ (one .sql file per query). The database is MySQL (5.7 in the provided Docker Compose), which holds tables like Card, User, and Collection. Docker Compose is used to orchestrate the three services (db, backend, frontend) so it runs consistently across OSes (containers run on Linux via Docker Desktop on Windows/macOS).

R3 - Database with sample dataset

We will use a Python script to fetch card metadata from the public TCGDEX Pocket API (https://tcgdex.dev/tcg-pocket). The script should normalize fields (cardID, name, packName, rarity, type, imageURL). We produce an artifact, which is a SQL seed file (e.g., app/database/migrations/init.sql) with INSERTs for Card, User, and sample Collection rows. For our sample data, we'll only include a small set of ~25 cards. When we create our full production database, we can use or extend this same script to gather ALL existing cards.

Here is our sample Card instance:

cardID	name	packName	rarity	type	imageURL
A1-001	Bulbasaur	Mewtwo	1D	Grass	https://assets.tcgdex.net/en/tcgp/A1/001/high.webp
A1-002	Ivysaur	Mewtwo	2D	Grass	https://assets.tcgdex.net/en/tcgp/A1/002/high.webp
A1-003	Venusaur	Mewtwo	3D	Grass	https://assets.tcgdex.net/en/tcgp/A1/003/high.webp
A1-004	Venusaur ex	Mewtwo	4D	Grass	https://assets.tcgdex.net/en/tcgp/A1/004/high.webp
A1-033	Charmander	Charizard	1D	Fire	https://assets.tcgdex.net/en/tcgp/A1/033/high.webp
A1-034	Charmeleon	Charizard	2D	Fire	https://assets.tcgdex.net/en/tcgp/A1/034/high.webp
A1-035	Charizard	Charizard	3D	Fire	https://assets.tcgdex.net/en/tcgp/A1/035/high.webp
A1-036	Charizard ex	Charizard	4D	Fire	https://assets.tcgdex.net/en/tcgp/A1/036/high.webp
A1-053	Squirtle	Pikachu	1D	Water	https://assets.tcgdex.net/en/tcgp/A1/053/high.webp
A1-054	Wartortle	Pikachu	2D	Water	https://assets.tcgdex.net/en/tcgp/A1/054/high.webp
A1-055	Blastoise	Pikachu	3D	Water	https://assets.tcgdex.net/en/tcgp/A1/055/high.webp
A1-056	Blastoise ex	Pikachu	4D	Water	https://assets.tcgdex.net/en/tcgp/A1/056/high.webp
A1-094	Pikachu	Pikachu	1D	Lightning	https://assets.tcgdex.net/en/tcgp/A1/094/high.webp
A1-096	Pikachu ex	Pikachu	4D	Lightning	https://assets.tcgdex.net/en/tcgp/A1/096/high.webp
A1-129	Mewtwo ex	Mewtwo	4D	Psychic	https://assets.tcgdex.net/en/tcgp/A1/129/high.webp
A1-227	Bulbasaur	Mewtwo	1S	Grass	https://assets.tcgdex.net/en/tcgp/A1/227/high.webp
A1-230	Charmander	Charizard	1S	Fire	https://assets.tcgdex.net/en/tcgp/A1/230/high.webp
A1-232	Squirtle	Pikachu	1S	Water	https://assets.tcgdex.net/en/tcgp/A1/232/high.webp
A1-266	Erika	Charizard	2S	Trainer	https://assets.tcgdex.net/en/tcgp/A1/266/high.webp
A1-267	Misty	Pikachu	2S	Trainer	https://assets.tcgdex.net/en/tcgp/A1/267/high.webp
A1-268	Blaine	Charizard	2S	Trainer	https://assets.tcgdex.net/en/tcgp/A1/268/high.webp
A1-280	Charizard ex	Charizard	3S	Fire	https://assets.tcgdex.net/en/tcgp/A1/280/high.webp
A1-281	Pikachu ex	Pikachu	3S	Lightning	https://assets.tcgdex.net/en/tcgp/A1/281/high.webp
A1-282	Mewtwo ex	Mewtwo	3S	Psychic	https://assets.tcgdex.net/en/tcgp/A1/282/high.webp
A1-285	Pikachu ex	Shared	C	Lightning	https://assets.tcgdex.net/en/tcgp/A1/285/high.webp
A1-286	Mewtwo ex	Shared	C	Psychic	https://assets.tcgdex.net/en/tcgp/A1/286/high.webp

Here is our sample User instance:

userID	username	passwordHash	dateJoined
1	trainer	password123	2025-10-20 13:30:00
2	Bob	password456	2025-10-20 13:30:00
3	Chloe	password789	2025-10-20 13:30:00
4	a	ca978112ca1	2025-10-20 13:30:00

Here is our sample Collection instance:

userID	cardID	quantity	dateAcquired
1	A1-033	2	2025-10-22 02:22:36
1	A1-055	1	2025-10-22 02:22:36
1	A1-096	1	2025-10-22 02:22:36
2	A1-053	1	2025-10-22 02:22:36
2	A1-002	2	2025-10-22 02:22:36
2	A1-003	1	2025-10-22 02:22:36
3	A1-001	1	2025-10-22 02:22:36

Here is our sample Wishlist instance:

```
+--------+--------+---------------------+
| userID | cardID | dateAcquired        |
+--------+--------+---------------------+
|      1 | A1-053 | 2025-10-22 02:22:36 |
|      1 | A1-001 | 2025-10-22 02:22:36 |
|      2 | A1-033 | 2025-10-22 02:22:36 |
|      3 | A1-033 | 2025-10-22 02:22:36 |
+--------+--------+---------------------+
```

R4 - Database with production dataset

Our production dataset is sourced from the public TCGDEX Pocket API (https://tcgdex.dev/tcg-pocket), which provides comprehensive metadata for all Pokémon TCG Pocket cards. A Python script fetches card data from the API for the first expansion pack (A1 series), including all cards from the Mewtwo, Charizard, and Pikachu packs. The script normalizes API responses to our schema format, mapping fields to our Card table structure (cardID, name, packName, rarity, type, imageURL), while cleaning inconsistencies and validating rarity enumerations. This process produces the SQL seed file: init-prod.sql, containing INSERT statements for all 732 Card records and production-ready User accounts with properly hashed passwords, Collection entries, and Wishlist entries showing cards that users are actively seeking to acquire. This separation allows for independent management of card data and user data.
This dataset allows the application to demonstrate full functionality, including the ability for users to browse available cards, track their collections, maintain wishlists, and identify potential trade partners based on their needs. The database can be populated by executing the generated init-prod.sql file, against a fresh MySQL instance, creating a production-ready state with realistic user activity and card distribution patterns.

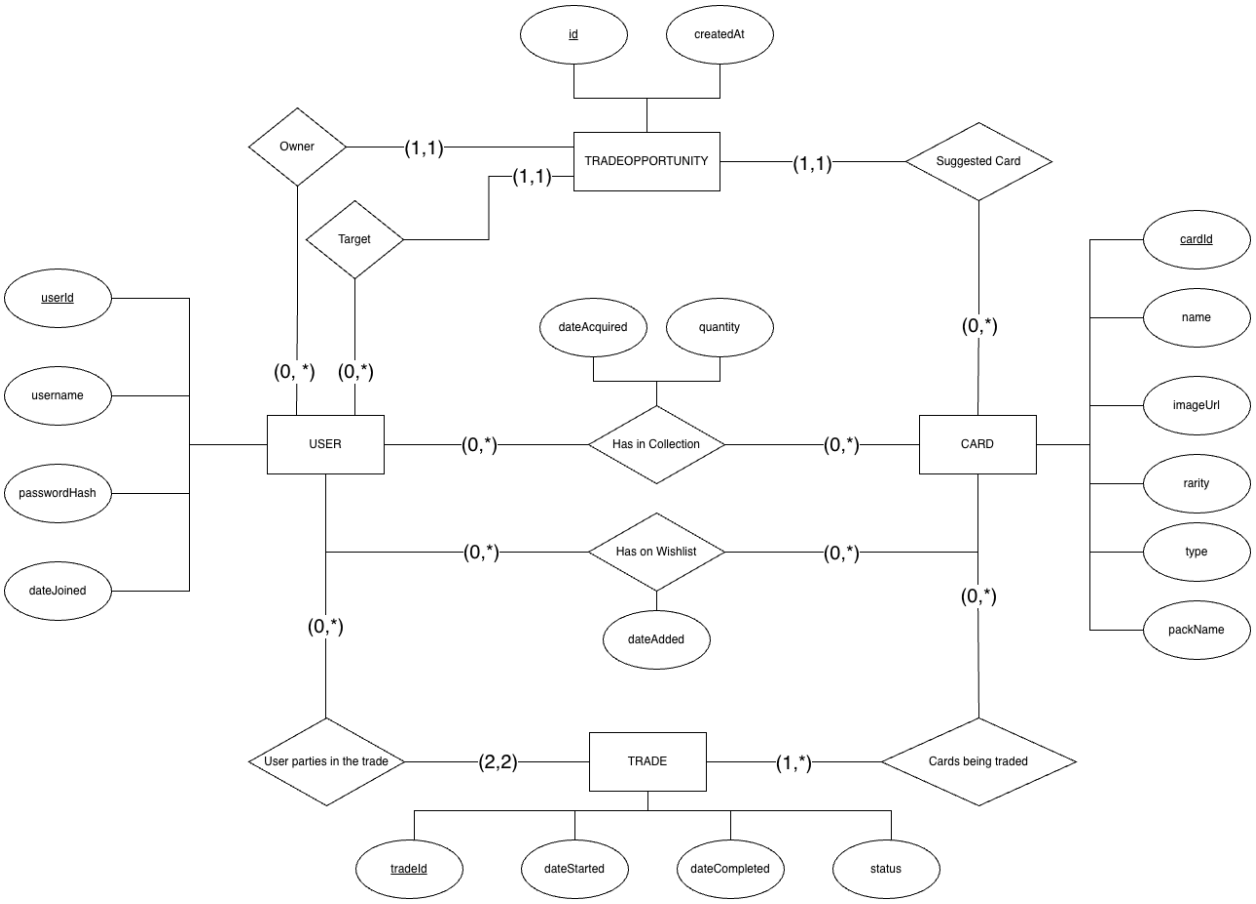
R5 - Design database schema

R5a - Assumptions

1. Each user has a unique user ID and username
2. Each Pokemon card has a unique card ID
3. Card rarity levels follows Pokemon TCG Pocket rarity levels (1-4 diamond, 1-2 star)
4. All cards have defined types (fire, water, grass, etc.)
5. A user can own multiple copies of the same card, tracked by a quantity field. Users can only trade cards they currently own
6. User can add multiple cards to their wishlist. The same card can appear on multiple users' wishlists.
7. A trade involves exactly 2 users and 2 cards
8. Both cards in trade should have the same rarity level
9. Trade statuses are: "pending", "accepted", "rejected", "cancelled", "completed"
    a. Only completed trades affect user collections
10. When trade is completed, card quantities are automatically updated in both users' collections

11. Users should not be able to delete their account if they have pending trades
12. Date fields should use format YYYY-MM-DD HH:MI:SS

R5b - E/R diagram



R5c - Relational Data Model

User (<u>userID</u>, username, passwordHash, dateJoined)

- PK: (<u>userID</u>)

Card (<u>cardID</u>, name, imageUrl, rarity, type, packName)

- PK: (<u>cardID</u>)

Collection (<u>userID</u>, <u>cardID</u>, dateAcquired, quantity)

- PK: (<u>userID</u>, <u>cardID</u>)
- FK: userId references User(userID)
- FK: cardId references Card(cardID)

Wishlist (<u>userId</u>, <u>cardId</u>, dateAdded)

- PK: (<u>userId</u>, <u>cardId</u>)
- FK: userId references User(userID)
- FK: cardId references Card(cardID)

Trade (<u>tradeId</u>, initiatorID, recipientID, status, createdBy, confirmedBy, dateStarted, dateCompleted)

- PK: (<u>tradeID</u>)

- FK: initiatorID references User(userID)
- FK: recipientID references User(userID)
- FK: createdBy references User(userID)
- FK: confirmedBy references User(userID)

TradeCard (id, tradeID, fromUserID, toUserID, cardID)

- PK: (id)
- FK: tradeId references Trade(tradeID)
- FK: fromUserID references User(userID)
- FK: toUserID references User(userID)
- FK: cardID references Card(cardID)

TradeOpportunity(id, ownerID, targetID, cardID, createdAt)

- PK: (id)
- FK: ownerID references User(userID)
- FK: targetID references User(userID)
- FK: cardID references Card(cardID)

R6 - Basic Feature/Functionality 1: Browse & Filter my collection

R6-a:

This feature allows users to browse and filter their cards. The user is any logged-in player.

Flow: User selects optional filters (rarity/type/text search) → Clicking search renders a table of matching cards with quantities → Clicking a row opens a card detail panel.

R6-b:

Query template:

```
-- :userId, :rarityOpt, :typeOpt, :packOpt, :nameSearchOpt are parameters
SELECT c.cardID, c.name, c.rarity, c.type, col.quantity
FROM COLLECTION col
JOIN CARD c ON c.cardID = col.cardID
WHERE col.userID = :userId
  AND (:rarityOpt IS NULL OR c.rarity = :rarityOpt)
  AND (:typeOpt   IS NULL OR c.type   = :typeOpt)
  AND (:packOpt   IS NULL OR c.packName = :packOpt)
  AND (:nameSearchOpt IS NULL OR c.name LIKE CONCAT('%%', :nameSearchOpt,
'%%'))
ORDER BY c.rarity, c.name;
```

Sample run:

```
SELECT c.cardID, c.name, c.rarity, col.quantity
FROM COLLECTION col
JOIN CARD c ON c.cardID = col.cardID
WHERE col.userID = 1 AND c.rarity = '1D' AND c.type = 'Fire'
ORDER BY c.name;
```

Expected output:

```
cardID | name       | rarity | quantity
-------+------------+--------+---------
A1-033 | Charmander | 1D     | 2
```

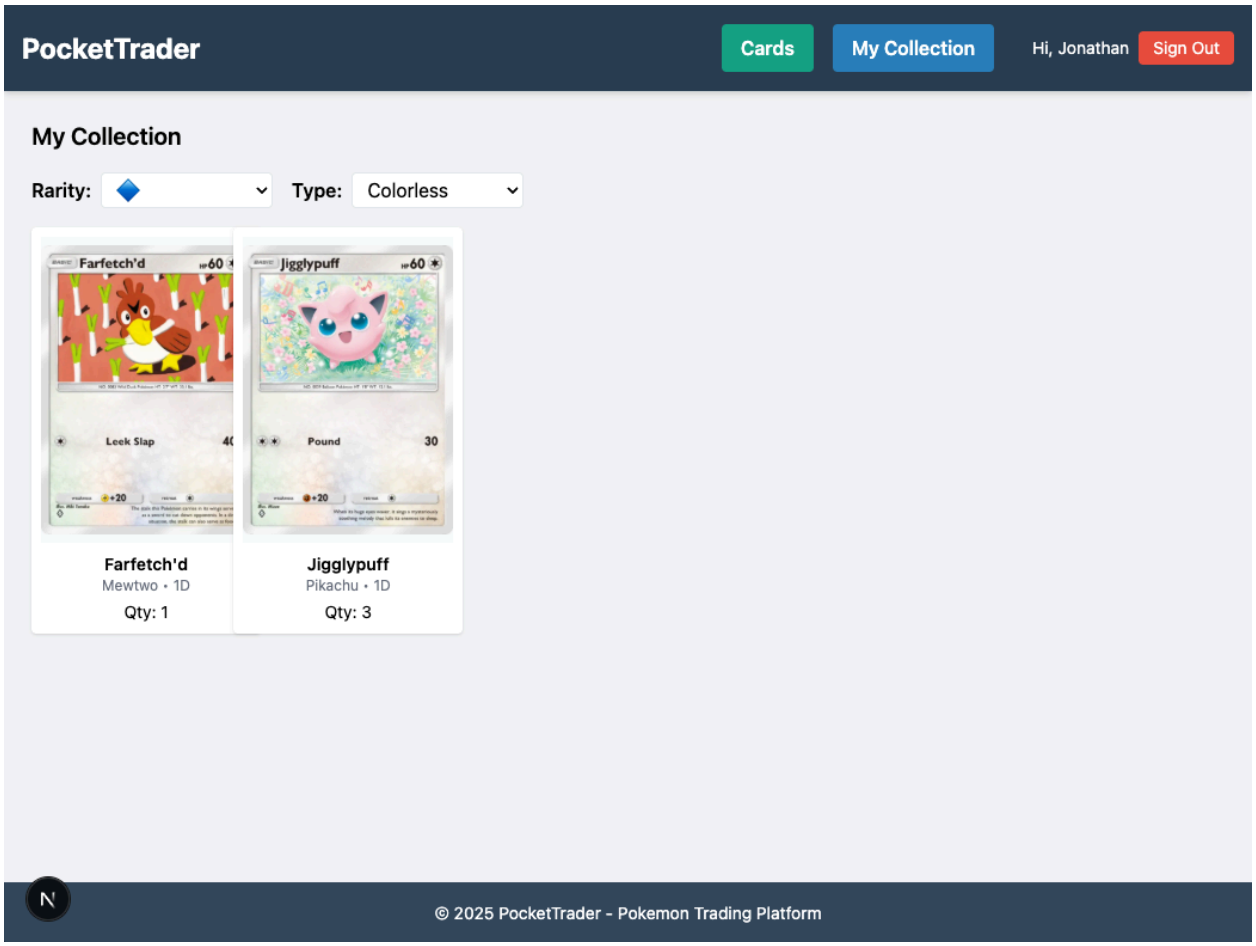
R6-c:

In app/database/schema.sql, the primary key for collection is already userId, cardId

```
CREATE INDEX idx_collection_user_card ON COLLECTION (userID, cardID);
```

The query always filters by col.userID = :userId and then joins on col.cardID = c.cardID. The primary key on Collection(userID, cardID) already gives us an index on (userID, cardID), so no additional index is needed.

R6-d:



As we can see in the test above, the rarity and type are correctly filtered.

Test-case description: Sign in or sign up, click on the “My Collection” button, filter by rarity and/or by type.

R7 - Basic Feature/Functionality 2: Add a card to my collection

R7-a:

This feature allows a user to add a card to their collection. The user is any logged-in user.

Flow: The user navigates to the “Add to Collection” page. They search for a Pokémon card by name, enter a quantity, and click “Add Card.” If the card already exists in their collection, we increase the

quantity. Otherwise, we add a new record. The app then displays a confirmation message and updates the visible collection table.

R7-b:

Query template:

```
INSERT INTO COLLECTION(userID, cardID, quantity, dateAcquired)
VALUES (:userID, :cardID, :quantity, NOW())
ON DUPLICATE KEY UPDATE quantity = quantity + VALUES(quantity);
```

Sampled query:

```
INSERT INTO COLLECTION(userID, cardID, quantity, dateAcquired)
VALUES (1, 'A1-035', 1, '2025-10-20 13:30:00')
ON DUPLICATE KEY UPDATE quantity = quantity + VALUES(quantity);

-- Verify
SELECT userID, cardID, quantity FROM COLLECTION WHERE userID=1 AND
cardID='A1-035';
```

Expected output:

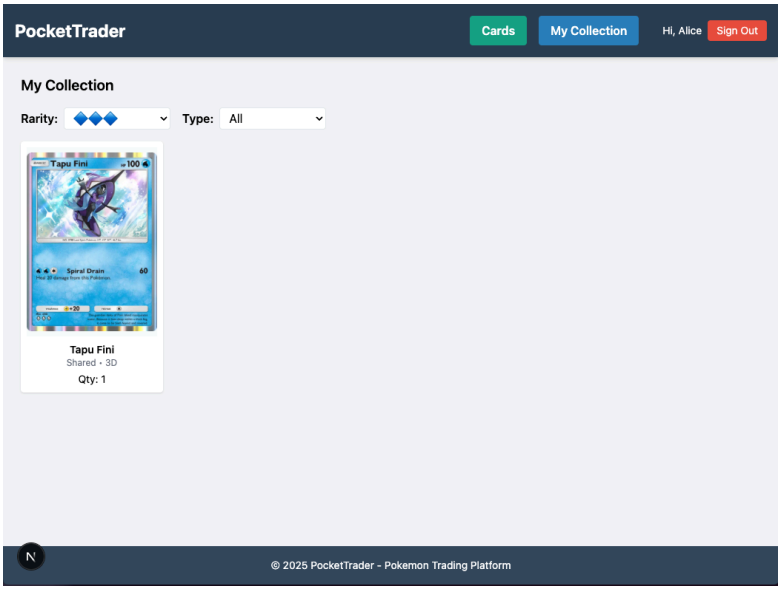
```
userID | cardID | quantity
-------+--------+---------
1      | A1-035  | 1
```

R7-c

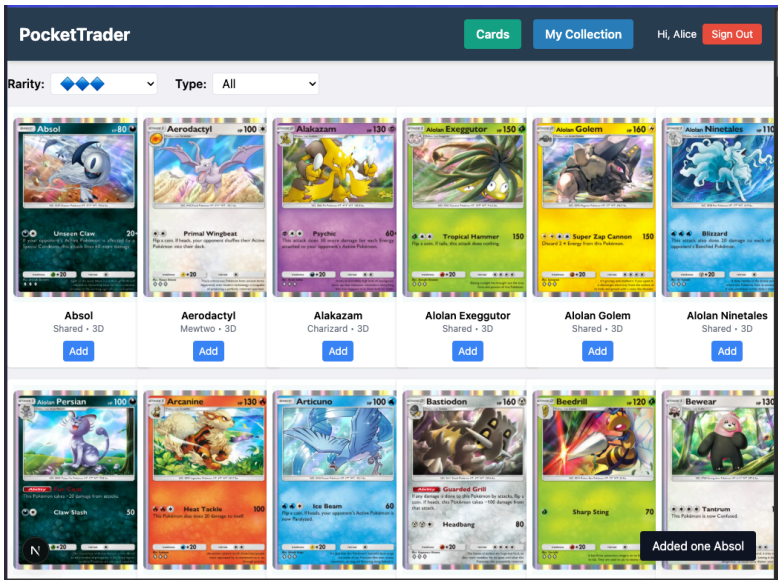
The logic of this query for production data remains unchanged from the query for the sample data.
This insert checks if the duplicate key (userId, cardId) is already present. Since (userId, cardId) is already the primary key of Collection, this cannot be further optimized by an index. Adding an index on (userId, cardId) would be redundant.

R7-d

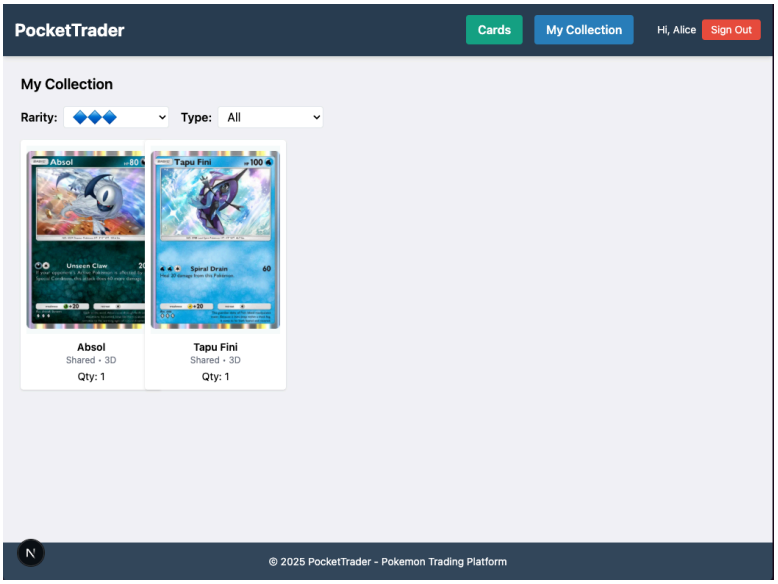
Before adding Absol to collection, this is the My collection page:



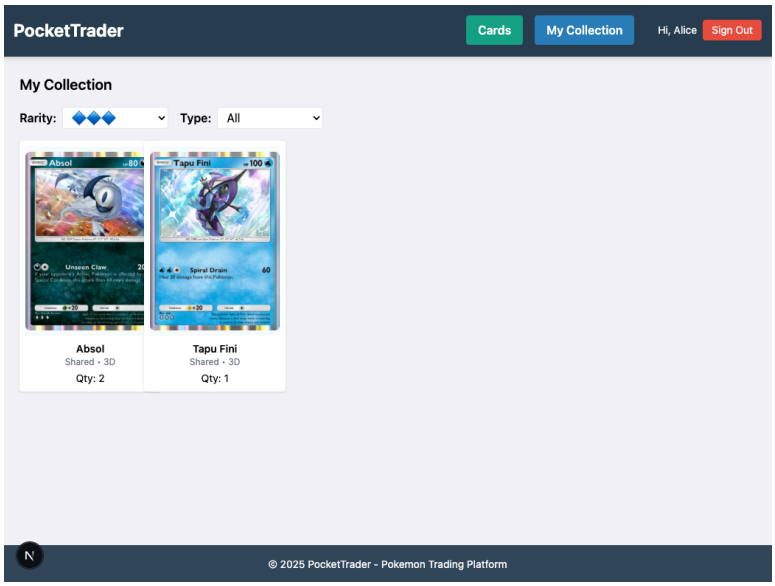
Adding Absol to collection on the Cards page



After adding Absol to collection, the My Collection page shows:



After adding another Absol to collection, the My Collection page shows:



R8 - Basic Feature/Functionality 3: Wishlist availability

R8-a:

> Let a logged-in user view which other users can likely spare a card on their wishlist (owners who have extra copies)
>
> Flow: On the Wishlist page the user can open a specific wishlist card and request "Who can trade this?" The backend returns a list of owners (other users) who have quantity > 1 of that card and the quantity they have. The frontend shows those users in a modal/list.

R8-b:

Query template:

```
-- :userId is the viewer
SELECT col.userID AS ownerID,
       u.username,
       col.quantity
FROM Collection col
JOIN User u ON u.userID = col.userID
WHERE col.cardID = :cardId
  AND col.userID <> :userId
  AND col.quantity > 1
ORDER BY u.username;
```

Sampled Query:

```
SELECT col.userID AS ownerID,
       u.username,
       col.quantity
FROM Collection col
JOIN User u ON u.userID = col.userID
WHERE col.cardID = 'A1-001'
  AND col.userID <> 1
  AND col.quantity > 1
ORDER BY u.username;
```

Expected output:

```
+---------+----------+----------+
| ownerID | username | quantity |
+---------+----------+----------+
| 9       | ash      | 3        |
| 12      | misty    | 2        |
+---------+----------+----------+
```

R8-c:

We add the index

```
CREATE INDEX idx_collection_card_qty_user ON Collection (cardID, quantity, userID);
```

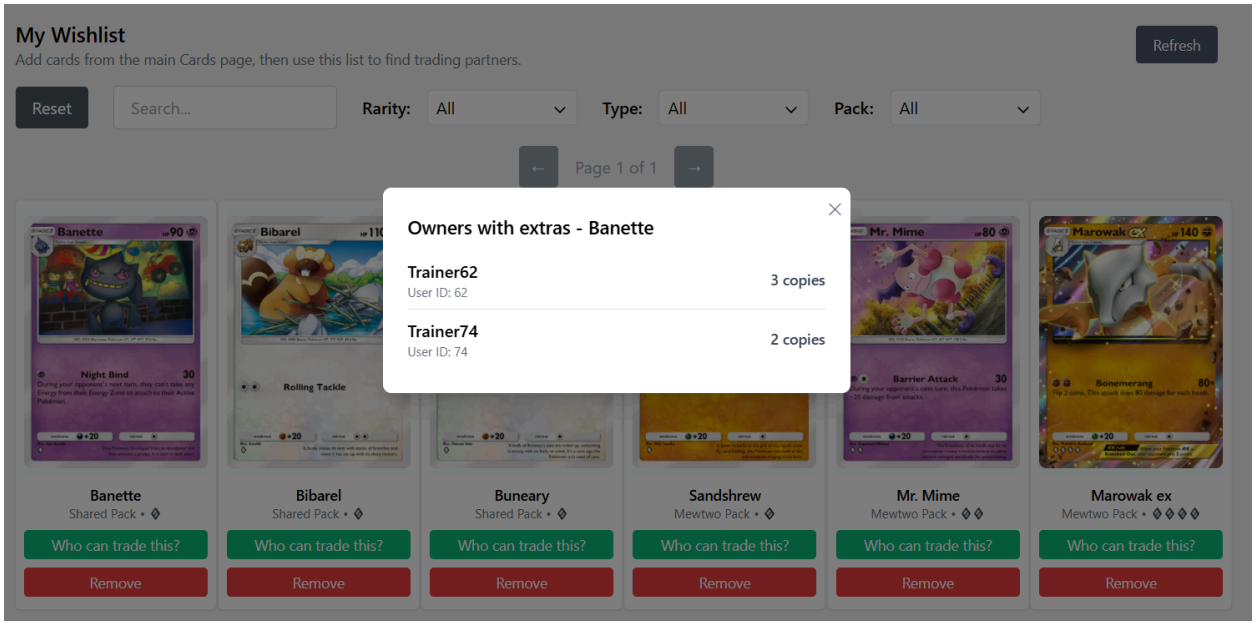
The index begins with cardID, allowing MySQL to directly jump to all rows matching a specific card. Adding quantity next filters out low-quantity rows efficiently using index ranges. Including userID allows

the join condition to be satisfied without an extra lookup.

To measure the improvement:

1. Ran EXPLAIN before/after adding the index.
    - Before: full or large index scan on Collection using the PK.
    - After: index seek on idx_collection_card_qty_user with dramatically fewer scanned rows.
2. Executed the query on the production dataset and recorded execution time in test-production.sql / test-production.out.
3. Verified that sorting now uses the User(username) index rather than performing a filesort.

R8-d:



R9 - Basic Feature/Functionality 4: Find mutual trade matches

R9-a:

This feature allows a user A to find users B satisfying A wants X, B has X, B wants Y, and A has Y, with X and Y having the same rarity. The user is any logged-in user on the Find Matches page.

Flow: The user clicks Find Mutual Matches. The result cards show: Partner, They give → I want, I give → They want. Only pairs where both cards share the same rarity are shown.

R9-b:

Query template:

```sql
-- :me is the logged-in userId
SELECT
  other.userID           AS partnerID,
  other.username         AS partnerName,
  wantMine.cardID        AS iWant_cardID,
  c1.name                AS iWant_name,
  c1.rarity              AS rarity_required,
  wantTheirs.cardID      AS theyWant_cardID,
  c2.name                AS theyWant_name
FROM USER me
JOIN USER other ON other.userID <> me.userID
-- I want a card the other owns
JOIN WISHLIST w_me        AS wantMine   ON wantMine.userID = me.userID
JOIN COLLECTION col_other               ON col_other.userID = other.userID AND col_other.cardID = wantMine.cardID AND col_other.quantity > 0
JOIN CARD c1                            ON c1.cardID = wantMine.cardID
-- They want a card I own
JOIN WISHLIST w_other     AS wantTheirs ON wantTheirs.userID = other.userID
JOIN COLLECTION col_me                  ON col_me.userID = me.userID AND col_me.cardID = wantTheirs.cardID AND col_me.quantity > 0
JOIN CARD c2                            ON c2.cardID = wantTheirs.cardID
-- Rarity must match
WHERE me.userID = :me
  AND c1.rarity = c2.rarity
ORDER BY other.username, c1.name, c2.name;
```

Sampled query:

```sql
SELECT
  other.userID           AS partnerID,
  other.username         AS partnerName,
  wantMine.cardID        AS iWant_cardID,
  c1.name                AS iWant_name,
  c1.rarity              AS rarity_required,
  wantTheirs.cardID      AS theyWant_cardID,
  c2.name                AS theyWant_name
FROM USER me
JOIN USER other ON other.userID <> me.userID
JOIN WISHLIST wantMine  ON wantMine.userID = me.userID
JOIN COLLECTION col_other
  ON col_other.userID = other.userID
 AND col_other.cardID = wantMine.cardID
 AND col_other.quantity > 0
JOIN CARD c1 ON c1.cardID = wantMine.cardID
JOIN WISHLIST wantTheirs ON wantTheirs.userID = other.userID
JOIN COLLECTION col_me
  ON col_me.userID = me.userID
 AND col_me.cardID = wantTheirs.cardID
 AND col_me.quantity > 0
JOIN CARD c2 ON c2.cardID = wantTheirs.cardID
WHERE me.userID = 1
  AND c1.rarity = c2.rarity
ORDER BY other.username, c1.name, c2.name;
```

Expected output:

```
partnerID | partnerName | iWant_cardID | iWant_name | rarity_required | theyWant_cardID | theyWant_name
----------+-------------+--------------+------------+-----------------+-----------------+--------------
2         | Bob         | A1-053       | Squirtle   | 1D              | A1-033          | Charmander
3         | Chloe       | A1-001       | Bulbasaur  | 1D              | A1-033          | Charmander
```

R9-c:

The optimizer can choose a join order like:

1. Start from my collection (col_me) → small set of cards I own.
2. For each owned card, jump into Wishlist via idx_wishlist_card_user to find all users who want that card (wantTheirs).
3. Then join to their Collection (col_other) to see if they own something from my wishlist, etc.

We add this index to schema.sql:

```
CREATE INDEX idx_wishlist_card_user ON Wishlist(cardID, userID);
```

Without this index, to find "who wants card X?" the DB has to:

- Either scan the whole Wishlist table, or
- Use the PK (userID, cardID) in a suboptimal way (it's not ordered by cardID first).

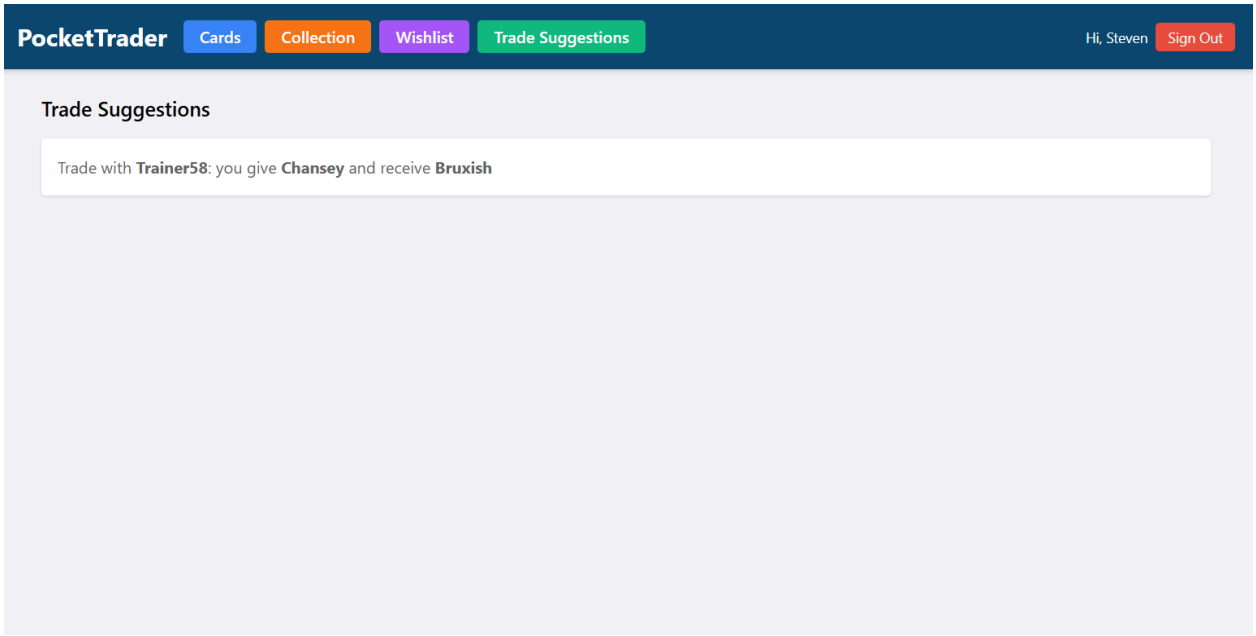
So this index is the main R9-specific tuning.
To measure the improvement, we compare plans and timings with and without the index:

```
# With tuning (default)
mysql ... -e "EXPLAIN ANALYZE
SELECT
  other.userID           AS partnerID,
  other.username         AS partnerName,
  wantMine.cardID        AS iWant_cardID,
  c1.name                AS iWant_name,
  c1.rarity              AS rarity_required,
  wantTheirs.cardID      AS theyWant_cardID,
  c2.name                AS theyWant_name
FROM User other
JOIN Wishlist wantMine  ON wantMine.userID = 1
JOIN Collection col_other
  ON col_other.userID = other.userID
 AND col_other.cardID = wantMine.cardID
 AND col_other.quantity > 0
JOIN Card c1           ON c1.cardID = wantMine.cardID
JOIN Wishlist wantTheirs
  ON wantTheirs.userID = other.userID
JOIN Collection col_me
  ON col_me.userID = 1
 AND col_me.cardID = wantTheirs.cardID
 AND col_me.quantity > 0
JOIN Card c2           ON c2.cardID = wantTheirs.cardID
WHERE other.userID <> 1
```

```
  AND c1.rarity = c2.rarity
ORDER BY other.username, c1.name, c2.name;" app_db

# Simulate no tuning by ignoring the new index
mysql ... -e "EXPLAIN ANALYZE
SELECT ...
FROM User other
JOIN Wishlist wantMine  ON wantMine.userID = 1
JOIN Collection col_other
  ON col_other.userID = other.userID
 AND col_other.cardID = wantMine.cardID
 AND col_other.quantity > 0
JOIN Card c1           ON c1.cardID = wantMine.cardID
JOIN Wishlist wantTheirs IGNORE INDEX (idx_wishlist_card_user)
  ON wantTheirs.userID = other.userID
JOIN Collection col_me
  ON col_me.userID = 1
 AND col_me.cardID = wantTheirs.cardID
 AND col_me.quantity > 0
JOIN Card c2           ON c2.cardID = wantTheirs.cardID
WHERE other.userID <> 1
  AND c1.rarity = c2.rarity
ORDER BY other.username, c1.name, c2.name;" app_db
```

We then compare the rows and timing columns in EXPLAIN ANALYZE. With idx_wishlist_card_user, MySQL probes Wishlist by cardID and touches far fewer rows; ignoring it forces a less selective plan and higher execution time on the production dataset.

R9-d:



R10 - Sign up and Login

R10-a:

This feature lets users sign up and login to the platform.

Flow: Sign up and log in buttons appear on the platform if the user is not already signed in. When each one of the buttons is pressed, a form opens up. For signup, the passwords are checked if they match, the username is checked to see if it doesn't exist already, and the columns of the user table are filled out. Although database constraints will prevent duplicate usernames, a query to check will provide a better error message. For login, the username and password hash are compared with the stored one.

R10-b

Signup uniqueness check query template:

```
SELECT 1 FROM User WHERE username = :username;
```

Signup uniqueness check query example:

```
SELECT 1 FROM User WHERE username = 'a';
```

Signup uniqueness check query expected output:

```
1
```

Signup insert query template:

```
INSERT INTO User(username, passwordHash, dateJoined)
VALUES (:username, :passwordHash, NOW());
```

Signup insert query example:

```
INSERT INTO User(username, passwordHash, dateJoined)
VALUES (
    'a',
    'ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb',
    '2025-10-20 13:30:00'
);
```

Login query template:

```
SELECT passwordHash FROM User WHERE username = :username;
```

Login query example:

```
SELECT passwordHash FROM User WHERE username = 'a';
```

Login query expected output:

```
passwordHash
----------------------------------------------------------------
ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
```

R10-c

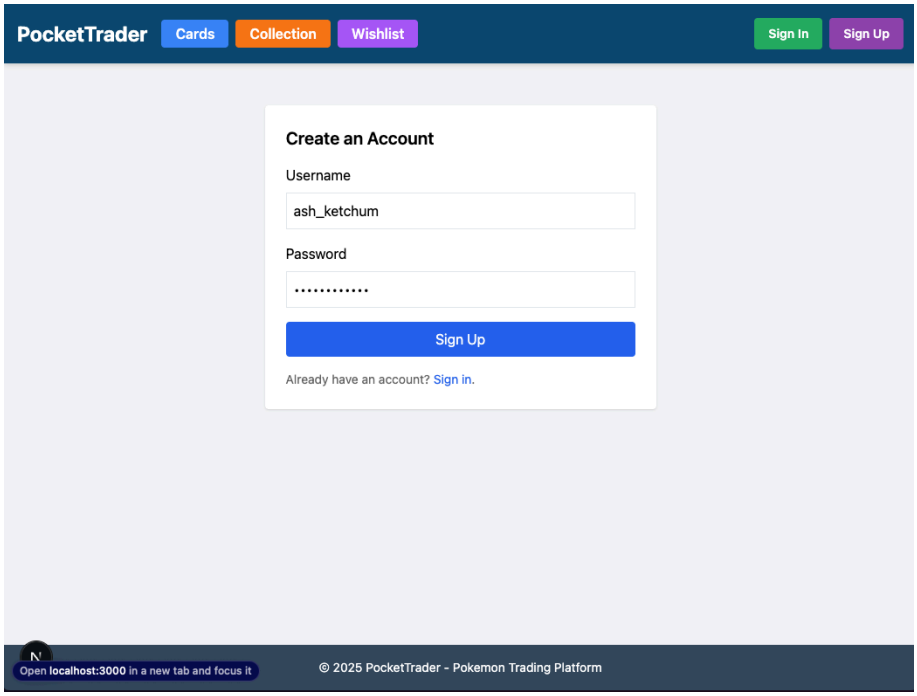
Both the signup uniqueness check and the login query filter by username.
To optimize these lookups, we already use a unique index on User(username). In our schema this is provided by the UNIQUE constraint:

```
username VARCHAR(50) UNIQUE NOT NULL
```

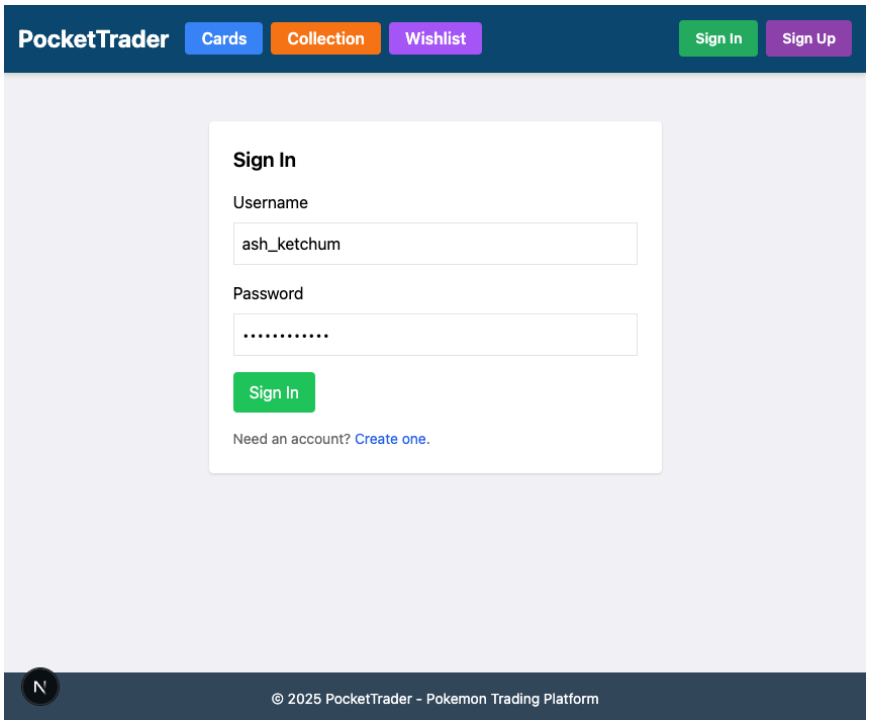
Hence, this is already efficient.

R10-d:

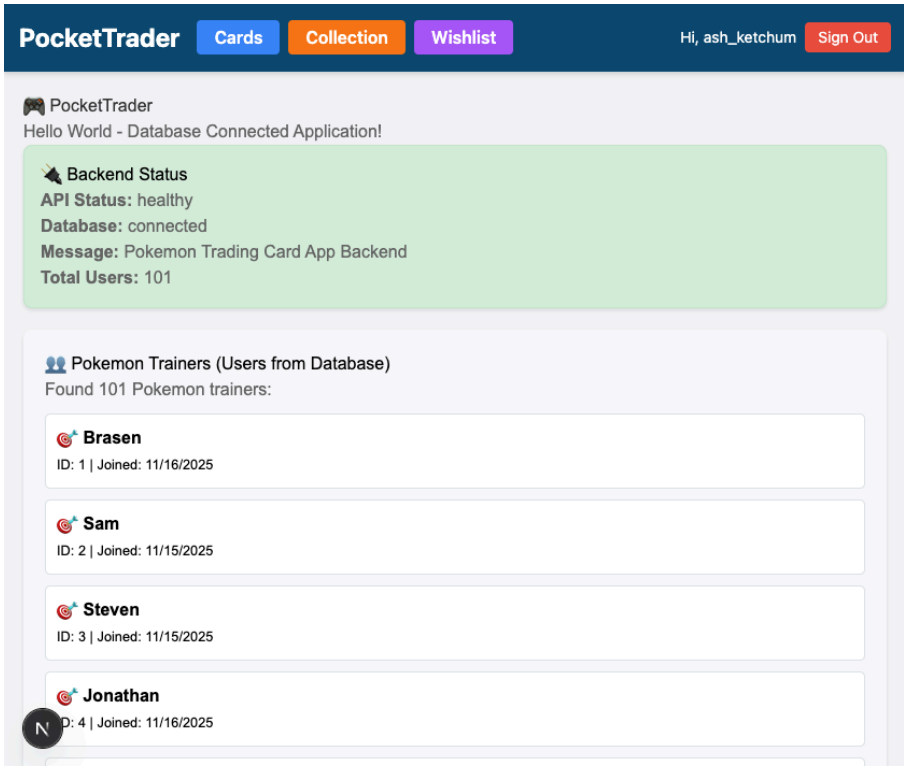
Sign up page:



Sign in page:



Home page (Logged in):



R11 - Trigger-based notifications for when a card becomes tradable

This feature automatically notifies users when trading opportunities arise. Whenever a user acquires a second copy of a card (making them able to trade it), the system immediately identifies all users who have that card on their wishlist and creates notification records. This eliminates the need for users to search for trade partners every time they acquire duplicates manually.

Flow: When a User adds a card to their collection, if the quantity reaches 2 or more, a Trigger automatically inserts records into the TradeOpportunity table for all users who want that card.
On the Wishlist page, affected users see a “New Opportunities” section showing who can now trade specific cards to them.

Table creation query:

```
-- TradeOpportunity table: records potential trade matches when a user has 2+ copies
CREATE TABLE IF NOT EXISTS TradeOpportunity (
    ownerID INT NOT NULL,
    targetID INT NOT NULL,
    cardID VARCHAR(50) NOT NULL,
    createdAt DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (ownerID, targetID, cardID),
    FOREIGN KEY (ownerID) REFERENCES User(userID) ON DELETE CASCADE,
    FOREIGN KEY (targetID) REFERENCES User(userID) ON DELETE CASCADE,
    FOREIGN KEY (cardID) REFERENCES Card(cardID) ON DELETE RESTRICT
);
```

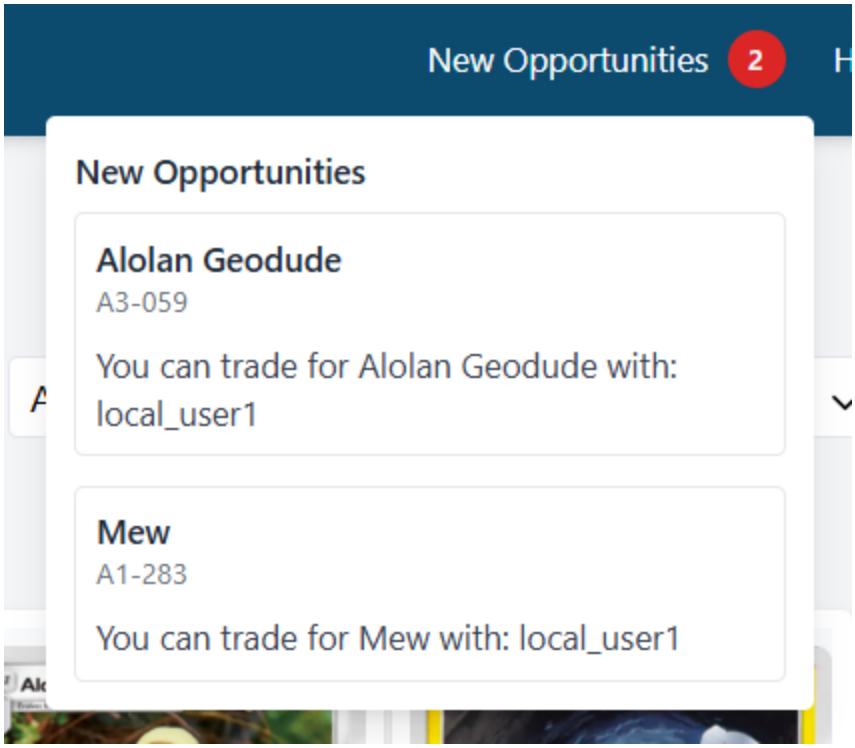
Triggers for automatic notification creation:

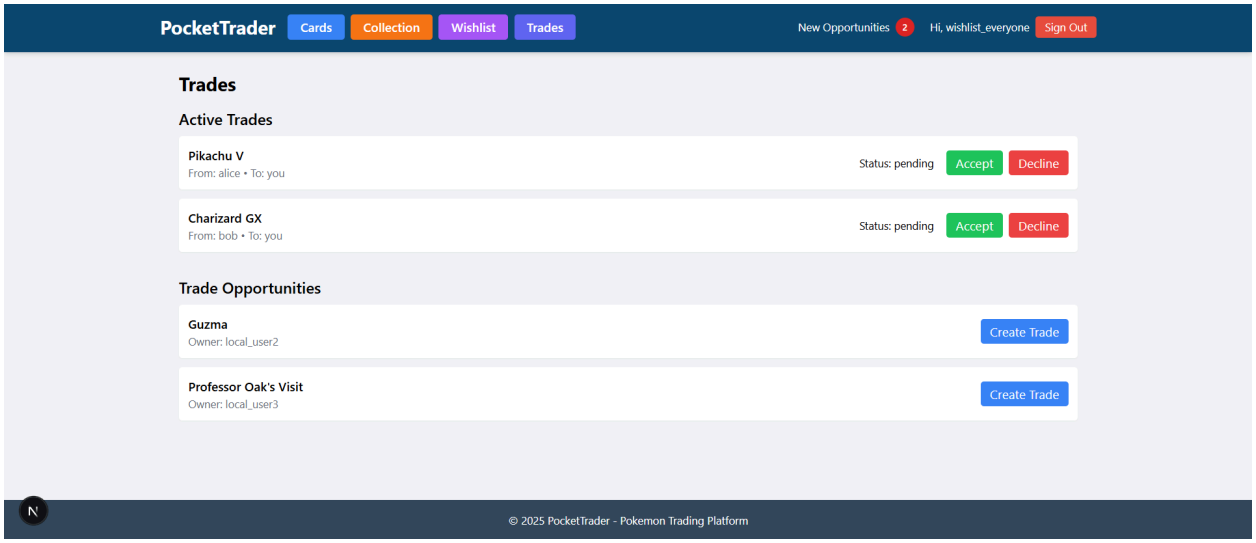
```
DELIMITER $$
CREATE TRIGGER trg_collection_after_insert
AFTER INSERT ON Collection
FOR EACH ROW
BEGIN
    IF NEW.quantity >= 2 THEN
        INSERT IGNORE INTO TradeOpportunity (ownerID, targetID, cardID)
        SELECT NEW.userID, w.userID, NEW.cardID
        FROM Wishlist w
        WHERE w.cardID = NEW.cardID
            AND w.userID <> NEW.userID;
    END IF;
END$$

CREATE TRIGGER trg_collection_after_update
AFTER UPDATE ON Collection
FOR EACH ROW
BEGIN
    IF OLD.quantity < 2 AND NEW.quantity >= 2 THEN
        INSERT IGNORE INTO TradeOpportunity (ownerID, targetID, cardID)
        SELECT NEW.userID, w.userID, NEW.cardID
        FROM Wishlist w
        WHERE w.cardID = NEW.cardID
            AND w.userID <> NEW.userID;
    END IF;
END$$
DELIMITER ;
```

Query template to fetch opportunities for a user:

```
-- Get trade opportunities for a target user
SELECT
    t.ownerID,
    u.username AS ownerName,
    t.cardID,
    c.name AS cardName,
    t.createdAt
FROM TradeOpportunity t
JOIN User u ON u.userID = t.ownerID
JOIN Card c ON c.cardID = t.cardID
WHERE t.targetID = :targetId
ORDER BY t.createdAt DESC;
```

This feature was tested on the production dataset containing 700+ cards and multiple Users with populated Collection and Wishlist tables. The trigger mechanism uses `INSERT IGNORE` to prevent duplicate notifications.





R12 - Active Trades View

The active trades view combines information from all trade-related databases to show an easier-to-read view of what active trades exist for the user

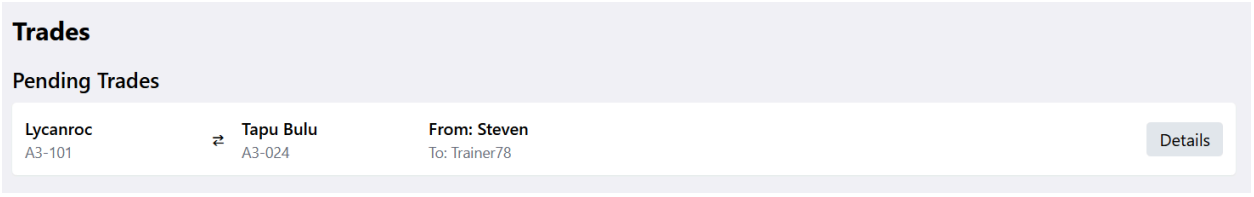
```
DROP VIEW IF EXISTS active_trades_view;
CREATE VIEW active_trades_view AS
```

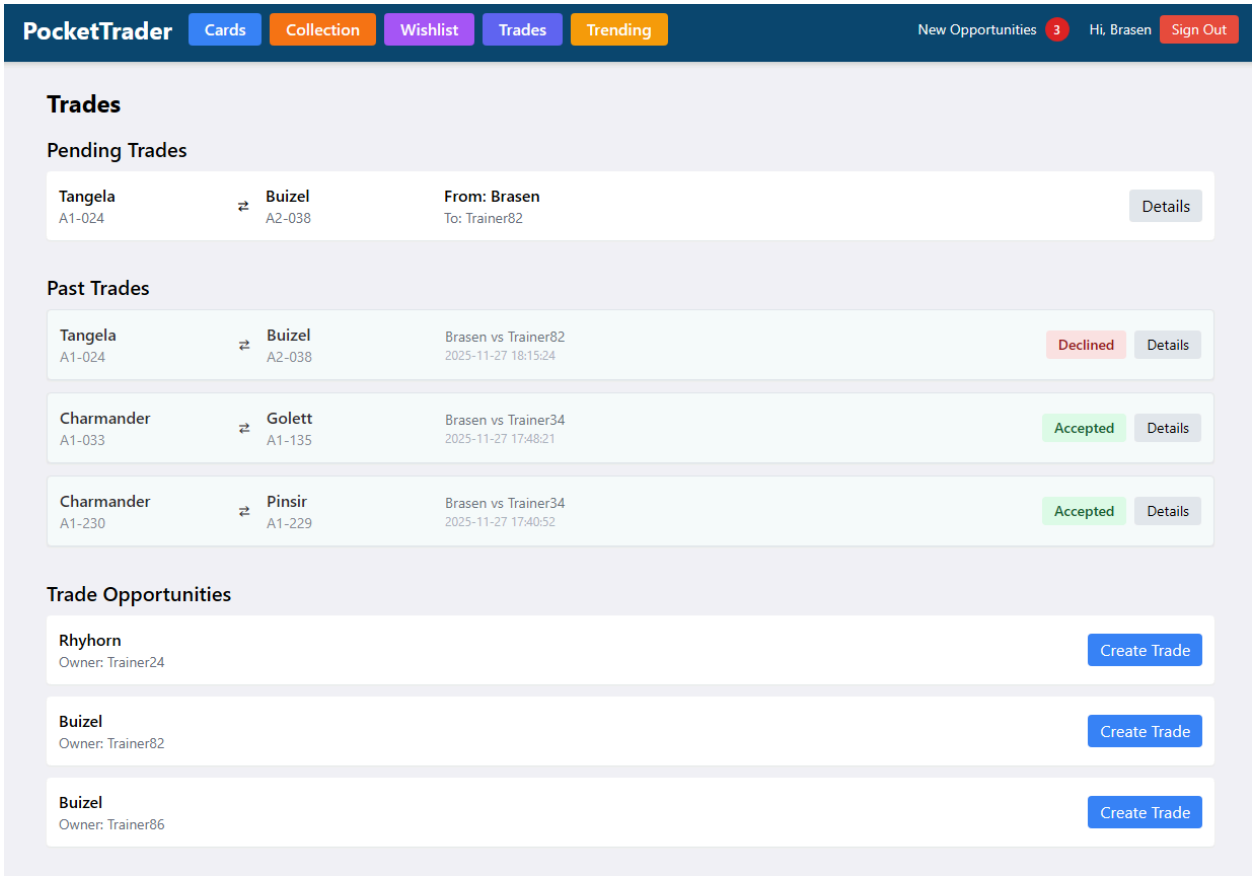
```
SELECT
  t.tradeID,
  t.initiatorID AS initiatorID,
  t.recipientID AS responderID,
  (SELECT tc.cardID FROM Tradecard tc WHERE tc.tradeID = t.tradeID AND tc.fromUserID =
t.initiatorID LIMIT 1) AS cardOfferedByUser1,
  (SELECT c1.name FROM Card c1 WHERE c1.cardID = (SELECT tc.cardID FROM Tradecard tc WHERE
tc.tradeID = t.tradeID AND tc.fromUserID = t.initiatorID LIMIT 1)) AS
cardOfferedByUser1Name,
  (SELECT c1.imageURL FROM Card c1 WHERE c1.cardID = (SELECT tc.cardID FROM Tradecard tc
WHERE tc.tradeID = t.tradeID AND tc.fromUserID = t.initiatorID LIMIT 1)) AS
cardOfferedByUser1Image,
  (SELECT tc.cardID FROM Tradecard tc WHERE tc.tradeID = t.tradeID AND tc.fromUserID =
t.recipientID LIMIT 1) AS cardOfferedByUser2,
  (SELECT c2.name FROM Card c2 WHERE c2.cardID = (SELECT tc.cardID FROM Tradecard tc WHERE
tc.tradeID = t.tradeID AND tc.fromUserID = t.recipientID LIMIT 1)) AS
cardOfferedByUser2Name,
  (SELECT c2.imageURL FROM Card c2 WHERE c2.cardID = (SELECT tc.cardID FROM Tradecard tc
WHERE tc.tradeID = t.tradeID AND tc.fromUserID = t.recipientID LIMIT 1)) AS
cardOfferedByUser2Image,
  t.status,
  t.createdBy,
  t.confirmedBy,
  t.dateCompleted,
  t.dateStarted
FROM Trade t;
```

To ensure data integrity of the status column in the Trade table, a trigger is added to make sure new rows all have the pending status.

```
DROP TRIGGER IF EXISTS trg_trade_before_insert$$
CREATE TRIGGER trg_trade_before_insert
BEFORE INSERT ON Trade
FOR EACH ROW
BEGIN
    SET NEW.status = 'pending';
END$$
```

The active trades table is used in the trades table, where pending trades related to the user are shown.





R13 - Atomic Active Trade Execution

This feature automatically executes confirmed trades atomically, ensuring both users' collections and wishlists are updated simultaneously without risk of partial completion. When both users confirm a trade, a single database transaction removes sent cards from collections, adds received cards, removes fulfilled wishlist items, and cleans up the trade record (all done atomically).

Flow: When User 2 confirms a pending trade, the confirmed column updates from FALSE to TRUE. The `AFTER UPDATE` trigger fires, and the transaction begins. Remove card1 from user1's collection, remove card2 from user2's collection. Add card2 to user1's collection, add card1 to user2's collection. Remove both cards from respective wishlists if present. Delete completed trade from the ActiveTrades table. Transaction commits, and both users see updated collections instantly.

Before a trade is processed, we verify that the trade is valid:

```
SELECT rarity FROM Card WHERE cardID = %s;
```

```
SELECT quantity FROM Collection WHERE userID = %s AND cardID = %s;
```

```
-- Check whether a given cardID is part of any pending trade
SELECT t.tradeID
FROM Trade t
JOIN Tradecard tc ON tc.tradeID = t.tradeID
WHERE t.status = 'pending' AND tc.cardID = %s
LIMIT 1;
```

Then we create a trade:

```
INSERT INTO Trade (initiatorID, recipientID, status, dateStarted, createdBy)
VALUES (%s, %s, 'pending', NOW(), %s);
```

Then we have triggers to process the trade. The BEFORE trigger enforces invariants and cancels the operation early with a clear error when preconditions fail. The AFTER trigger performs the actual state changes (the atomic swap) once the update has been accepted and committed.

```
DROP TRIGGER IF EXISTS trg_trade_before_update$$
CREATE TRIGGER trg_trade_before_update
BEFORE UPDATE ON Trade
FOR EACH ROW
BEGIN
    DECLARE cnt1 INT DEFAULT 0;
    DECLARE cnt2 INT DEFAULT 0;
    DECLARE v_card1 VARCHAR(50);
    DECLARE v_card2 VARCHAR(50);

    -- Only validate transitions to accepted
    IF OLD.status = 'pending' AND NEW.status = 'accepted' THEN
        -- confirmedBy must be provided and must be one of the participants
        IF NEW.confirmedBy IS NULL THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Trade: confirmedBy must
be provided when confirming';
        END IF;
        IF NOT (NEW.confirmedBy = OLD.initiatorID OR NEW.confirmedBy =
OLD.recipientID) THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Trade: confirmedBy must
be one of the participants';
        END IF;

        -- The confirmer cannot be the same as the creator
        IF NEW.confirmedBy = OLD.createdBy THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Trade: creator cannot
confirm their own trade';
        END IF;
```

```
        -- Get representative cards (single-card-per-side assumption)
        SELECT tc.cardID INTO v_card1 FROM Tradecard tc WHERE tc.tradeID =
OLD.tradeID AND tc.fromUserID = OLD.initiatorID LIMIT 1;
        SELECT tc.cardID INTO v_card2 FROM Tradecard tc WHERE tc.tradeID =
OLD.tradeID AND tc.fromUserID = OLD.recipientID LIMIT 1;

        -- Check user ownership
        SELECT COUNT(*) INTO cnt1 FROM Collection WHERE userID =
OLD.initiatorID AND cardID = v_card1 AND quantity >= 1;
        SELECT COUNT(*) INTO cnt2 FROM Collection WHERE userID =
OLD.recipientID AND cardID = v_card2 AND quantity >= 1;
        IF cnt1 = 0 THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Trade: initiator does
not have required card at confirmation time';
        END IF;
        IF cnt2 = 0 THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Trade: recipient does
not have required card at confirmation time';
        END IF;
    END IF;
END$$
```

```
DROP TRIGGER IF EXISTS trg_trade_after_update$$
CREATE TRIGGER trg_trade_after_update
AFTER UPDATE ON Trade
FOR EACH ROW
BEGIN
    DECLARE v_card1 VARCHAR(50);
    DECLARE v_card2 VARCHAR(50);

    IF OLD.status = 'pending' AND NEW.status = 'accepted' THEN
        -- fetch the one card IDs per participant
        SELECT tc.cardID INTO v_card1 FROM Tradecard tc WHERE tc.tradeID = NEW.tradeID AND
tc.fromUserID = NEW.initiatorID LIMIT 1;
        SELECT tc.cardID INTO v_card2 FROM Tradecard tc WHERE tc.tradeID = NEW.tradeID AND
tc.fromUserID = NEW.recipientID LIMIT 1;

        -- decrement initiator's card
        UPDATE Collection SET quantity = quantity - 1 WHERE userID = NEW.initiatorID AND
cardID = v_card1;
        -- increment/add to recipient
        INSERT INTO Collection (userID, cardID, quantity) VALUES (NEW.recipientID, v_card1,
1)
            ON DUPLICATE KEY UPDATE quantity = quantity + 1;
```

```
        -- decrement recipient's card
        UPDATE Collection SET quantity = quantity - 1 WHERE userID = NEW.recipientID AND
cardID = v_card2;
        -- increment/add to initiator
        INSERT INTO Collection (userID, cardID, quantity) VALUES (NEW.initiatorID, v_card2,
1)
            ON DUPLICATE KEY UPDATE quantity = quantity + 1;

        -- Remove received cards from wishlists
        DELETE FROM Wishlist WHERE userID = NEW.recipientID AND cardID = v_card1;
        DELETE FROM Wishlist WHERE userID = NEW.initiatorID AND cardID = v_card2;

    END IF;
END$$
```

The trigger is treated as atomic (implicitly) by the database to ensure the data is consistent.

R14 - Indexes For Optimization

Features/Goal: Speed up three core flows: (1) “Who can trade this wishlist card?” (R8) and (2) “Find mutual trade matches” (R9), as well as our advanced features for processing trades.
How it functions: We added targeted secondary indexes so MySQL can seek by card first, then filter quantities/users, avoiding full PK scans.
Indexes (queries):

```
-- Collection: find owners by card quickly
CREATE INDEX idx_collection_cardID ON Collection(cardID);
CREATE INDEX idx_collection_card_qty_user ON Collection(cardID, quantity,
userID);

-- Wishlist: find wishlist owners quickly
CREATE INDEX idx_wishlist_cardID ON Wishlist(cardID);
CREATE INDEX idx_wishlist_card_user ON Wishlist(cardID, userID);

-- Trade: filter by status and lookup trades for a user
CREATE INDEX idx_trade_status ON Trade(status);
CREATE INDEX idx_trade_initiator_recipient ON Trade(initiatorID, recipientID);

-- Tradecard: joins from Trade -> Tradecard and lookups by fromUser
CREATE INDEX idx_tradecard_tradeID ON Tradecard(tradeID);
CREATE INDEX idx_tradecard_fromUserID ON Tradecard(fromUserID);
CREATE INDEX idx_tradecard_toUserID ON Tradecard(toUserID);
CREATE INDEX idx_tradecard_cardID ON Tradecard(cardID);

-- TradeOpportunity: fast lookup of opportunities for a target user
CREATE INDEX idx_tradeopportunity_target ON TradeOpportunity(targetID);
```

Index Name	Table	Column(s)	Covering?	Primary Purpose	Typical Query Pattern
idx_collection_cardID	Collection	(cardID)	❌	Fast lookup of users owning a given card (R8, R9, R11)	`SELECT userID FROM Collection WHERE cardID = ?`
idx_collection_card_qty_user	Collection	(cardID, quantity, userID)	✅ for quantity filters	Filters by card + minimum quantity while returning userID directly (R8, R9, R11)	`SELECT userID FROM Collection WHERE cardID = ? AND quantity >= 2`
idx_wishlist_cardID	Wishlist	(cardID)	❌	Find wishlist holders for a card (R8, R9, R11)	`SELECT userID FROM Wishlist WHERE cardID = ?`
idx_wishlist_card_user	Wishlist	(cardID, userID)	✅	Covering lookup for "who wants card X" (R8, R9)	`SELECT userID FROM Wishlist WHERE cardID = ?`
idx_trade_status	Trade	(status)	❌	Scans for trades by activity state (esp. `'pending'`) (R12, R13)	`SELECT * FROM Trade WHERE status='pending'`
idx_trade_initiator_recipient	Trade	(initiatorID, recipientID)	❌	Fetch trades involving specific users (R12, R13)	`SELECT * FROM Trade WHERE initiatorID=? OR recipientID=?`
idx_tradecard_tradeID	Tradecard	(tradeID)	Partial	Joins Trade → Tradecard efficiently (R12, R13)	`SELECT cardID FROM Tradecard WHERE tradeID=?`
idx_tradecard_fromUserID	Tradecard	(fromUserID)	❌	Finds cards a user is offering in a trade (R12, R13)	`SELECT cardID FROM Tradecard WHERE tradeID=? AND fromUserID=?`

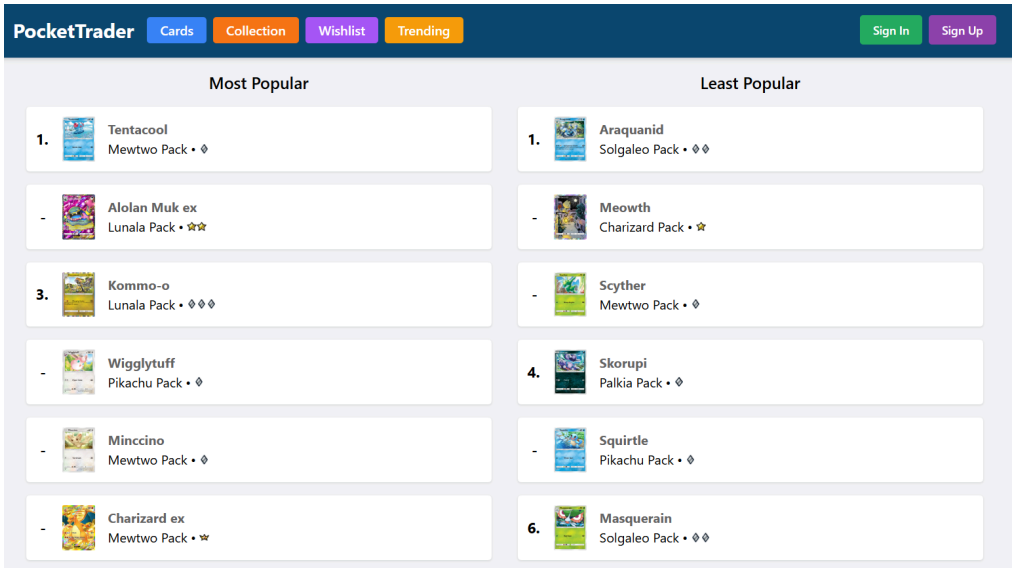
idx_tradecard_toUserID	Tradecard	(toUserID)	❌	Recipient-based Tradecard retrieval (R12, R13)	`SELECT * FROM Tradecard WHERE toUserID=?`
idx_tradecard_cardID	Tradecard	(cardID)	❌	Ensures a card isn't already involved in a pending trade (R11, R13)	`SELECT tradeID FROM Tradecard WHERE cardID=?`
idx_tradeopportunity_target	TradeOpportunity	(targetID)	Optional	Fast retrieval of opportunities directed to a user (R11, R12)	`SELECT ownerID, cardID FROM TradeOpportunity WHERE targetID=?`

R15 - Card Value Ranking with Dynamic Views

Every card has a "market value" based on supply and demand. Cards that many users want (high wishlist count) but few users can trade (low spare quantity) are considered more valuable. This feature creates a live-updated view that ranks all cards by their trade value.

```
DROP VIEW IF EXISTS market_trends_view;
CREATE VIEW market_trends_view AS
SELECT
    c.cardID,
    c.name,
    c.rarity,
    c.packName,
    c.imageURL,
    COUNT(DISTINCT w.userID) AS demand,
    COUNT(DISTINCT col.userID) AS supply,
    (COUNT(DISTINCT w.userID) - COUNT(DISTINCT col.userID)) AS trend
FROM Card c
LEFT JOIN Wishlist w ON c.cardID = w.cardID
LEFT JOIN Collection col ON c.cardID = col.cardID
GROUP BY c.cardID, c.name, c.rarity, c.packName, c.imageURL;
```

From a page on the website, the user can see a ranking of the top 10 most popular and least popular cards. Their respective set is also listed, so the user can see the most optimal pack to pull for to find trades. Here is how the feature looks on the production database:



R16 - Members

Jonathan Polina:

- Feature documentation and query design
- Production database seed generation (732 cards from TCGDEX API)
- Password hashing, wishlist page, trade prospects functionality

Steven Wu:

- Frontend implementation (cards, collections, wishlists, trending, matches, trades pages)
- Trending feature queries and R12 trigger implementation
- Advanced features implementation and documentation

Brasen Xu:

- Database schema design (E/R diagram, relational model)
- Sample and production dataset creation
- SQL feature queries and advanced features implementation
- Bug fixes for trading functionality

Samuel Zheng:

- R1-R3 project proposal
- Setting up backend and frontend project (Flask + Next + MySQL + Docker)
- Advanced features for trading (schema design, queries, views, triggers, and indices), including creating trades/processing trades/finding trade opportunities
- Backend and frontend implementation for cards, collections, wishlists, trading
- Sign up/login authentication system