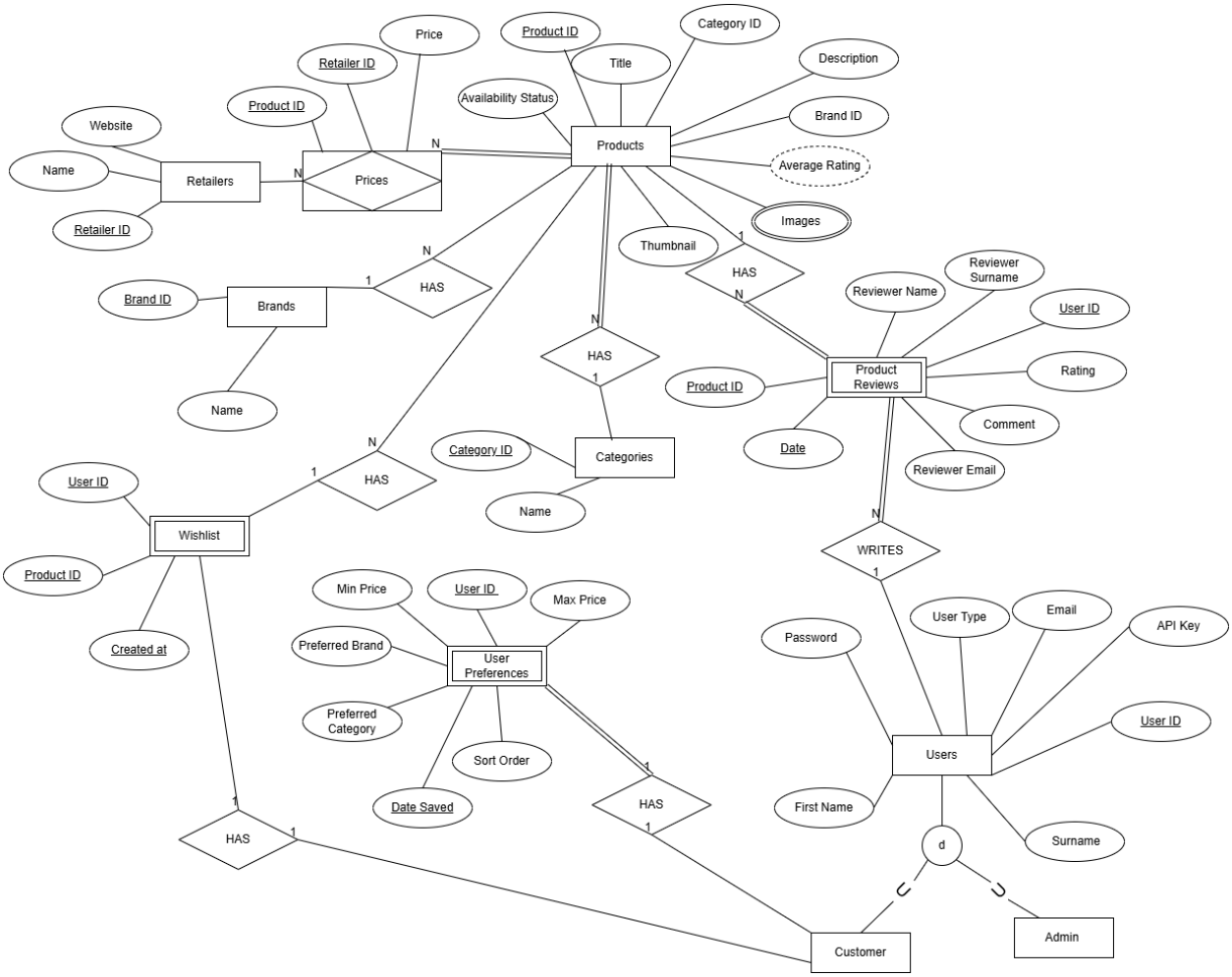
Group 404 NOT FOUND: K. Baloyi (u22697609), Z. Zwane (u23533413), T. Seaba (u23657350), K. Kgodumo (u22686348), B. Rulumente (u23547252), K. Asiedu (u18100156)

TASK 2: (E)ER-Diagram

Key Entities and Relationships:

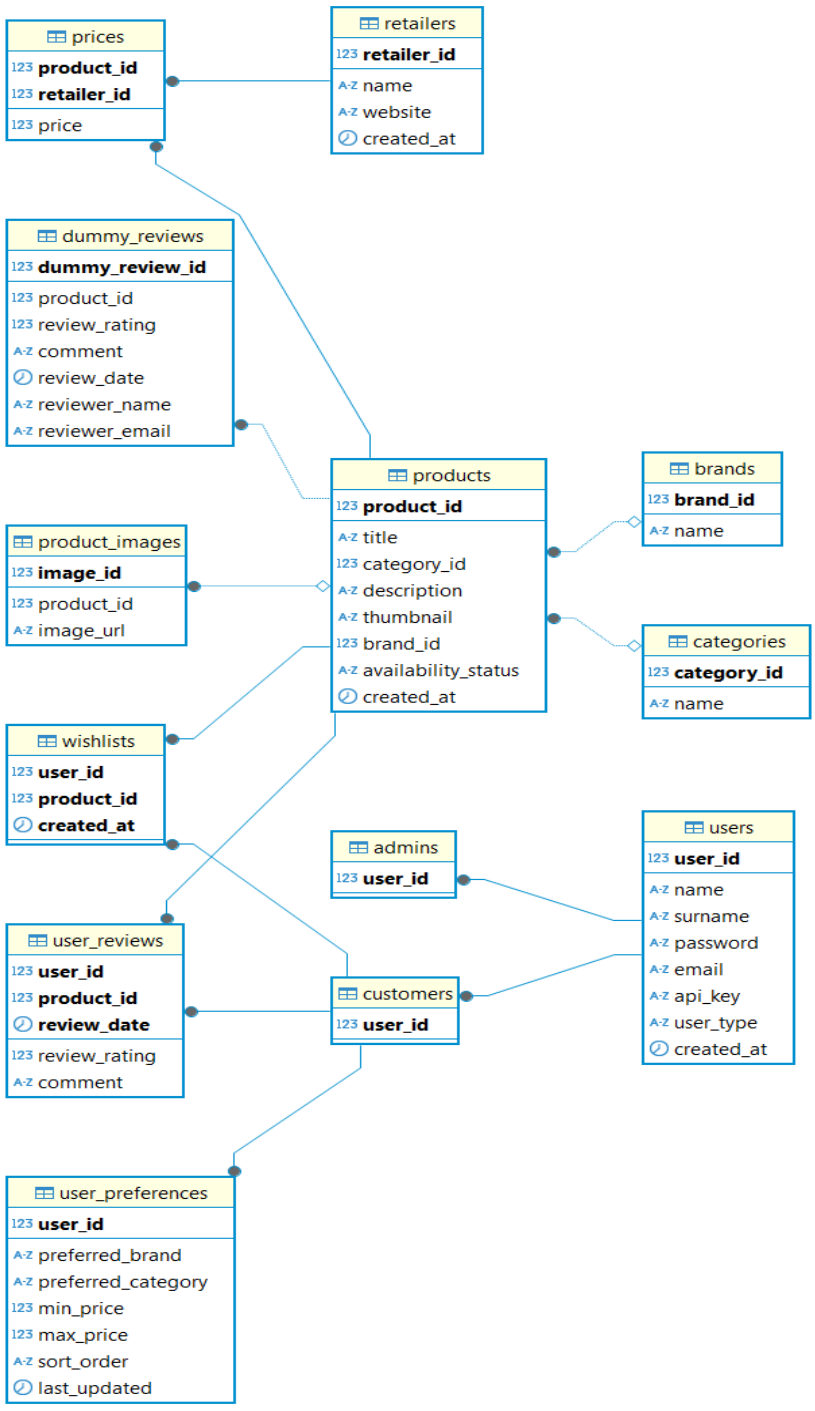
1. **Core Entities**:
    - Users (supertype) with subtypes: Customers and Admins
    - Products (connected to Brands and Categories)
    - Retailers
    - Reviews (both User Reviews and Dummy Reviews)
2. **Assumptions**:
    - A product can have multiple images (1:N relationship)
    - A product can be sold by multiple retailers (M:N via Prices junction table)
    - Users can have only one preference set (1:1 relationship)
    - Wishlist is a M:N relationship between users and products
3. **Iterations**:
    - **First Iteration**: Basic entities without subtypes
    - **Second Iteration**: Added user subtypes (Customer/Admin)
    - **Final Iteration**: Added weak entities (reviews, prices) and proper relationship cardinalities

Group 404 NOT FOUND: K. Baloyi (u22697609), Z. Zwane (u23533413), T. Seaba (u23657350), K. Kgodumo (u22686348), B. Rulumente (u23547252), K. Asiedu (u18100156)



Group 404 NOT FOUND: K. Baloyi (u22697609), Z. Zwane (u23533413), T. Seaba (u23657350), K. Kgodumo (u22686348), B. Rulumente (u23547252), K. Asiedu (u18100156)

TASK 4: RELATIONAL SCHEMA



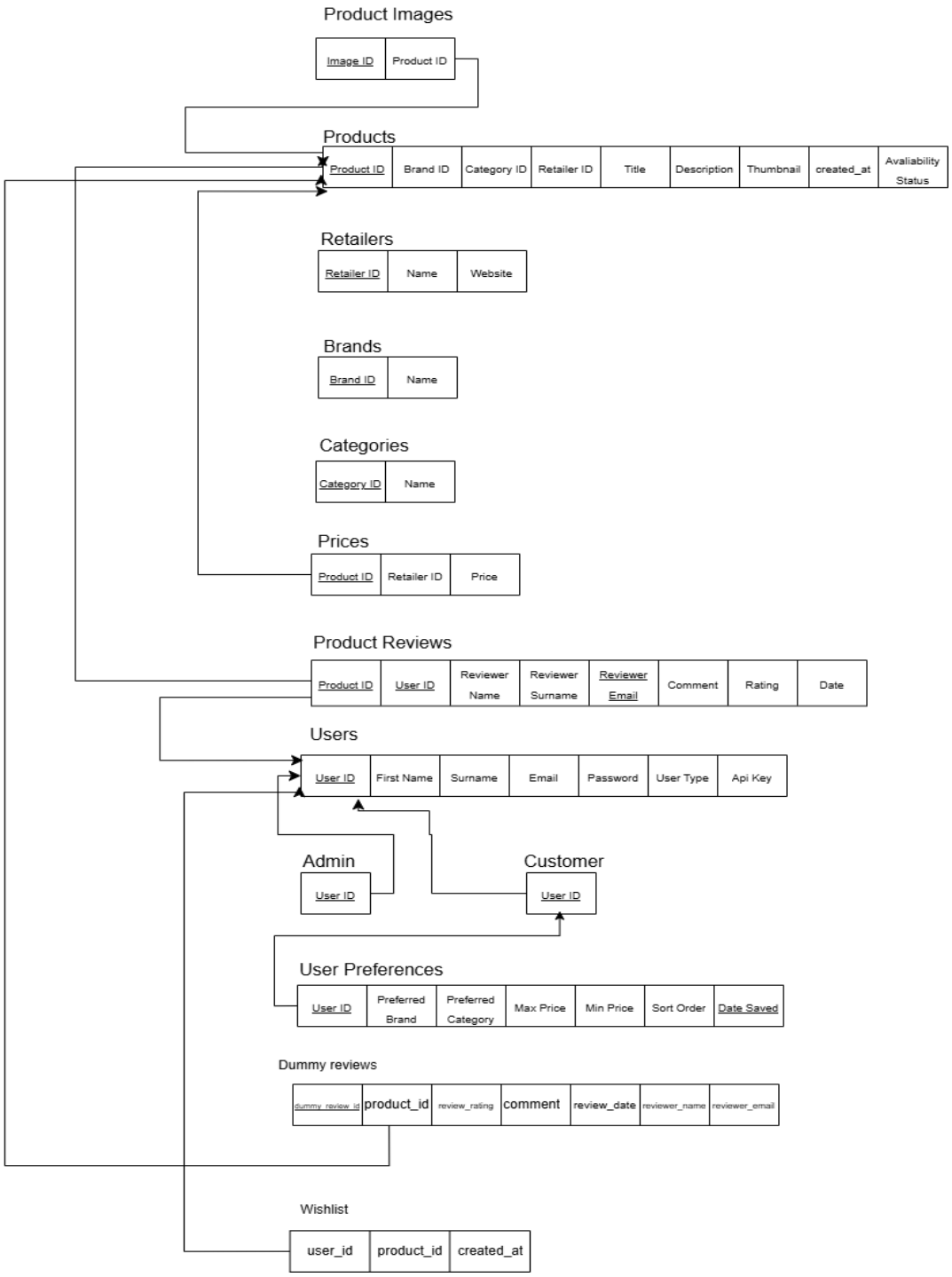
Group 404 NOT FOUND: K. Baloyi (u22697609), Z. Zwane (u23533413), T. Seaba (u23657350), K. Kgodumo (u22686348), B. Rulumente (u23547252), K. Asiedu (u18100156)

TASK 3: (E)ER-DIAGRAM TO RELATIONAL MAPPING

Conversion Steps:

1. **Regular Entities**:
    - Users → users table
    - Products → products table
    - Retailers → retailers table
2. **Specialization/Generalization**:
    - Implemented as separate tables with user_id as FK
    - admins and customers tables reference users
3. **Many-to-Many Relationships**:
    - Prices table connects products and retailers
    - Wishlists table connects users and products
4. **Weak Entities**:
    - product_images depends on products
    - All reviews depend on products

Group 404 NOT FOUND: K. Baloyi (u22697609), Z. Zwane (u23533413), T. Seaba (u23657350), K. Kgodumo (u22686348), B. Rulumente (u23547252), K. Asiedu (u18100156)



TASK 6: SAMPLE DATA

Data quality is crucial because it ensures that testing and demonstrations reflect real-world scenarios as closely as possible. For our **CompareIt** project, we carefully selected and prepared our sample data with these considerations in mind.

Data Source: DummyJSON

To populate our database with product and retailer information, we used **DummyJSON** — a publicly available API providing realistic, structured dummy data including products, prices, brands, and categories. DummyJSON offers a diverse range of product types and details that closely mimic real e-commerce platforms, which helps us simulate genuine user experiences.

Using DummyJSON allowed us to:

- Quickly gather a broad spectrum of relevant product data without manual entry.
- Ensure consistent data formatting, including images, descriptions, and pricing.
- Obtain sample user reviews and ratings that reflect typical shopping behavior.

Data Entry Methods

The data was imported into our MySQL database primarily through automated scripts. These scripts fetch data directly from the DummyJSON API and format it appropriately for insertion into our relational schema. This automated approach guarantees accuracy, reduces manual errors, and simplifies updates.

In addition to scripted imports, some manual entries were made to add custom products and special cases to test specific functionalities, such as filtering and sorting edge cases.

Quality and Relevance

We curated the sample data to:

- Include various categories and brands for comprehensive filtering and comparison.
- Represent a wide range of price points to validate sorting features.
- Include realistic quantities of user ratings and reviews to enhance credibility.
- Maintain data consistency and integrity through validation checks during import.

This careful approach ensures the application behaves realistically, providing meaningful search results and user interactions during demos and testing.

Group 404 NOT FOUND: K. Baloyi (u22697609), Z. Zwane (u23533413), T. Seaba (u23657350),
K. Kgodumo (u22686348), B. Rulumente (u23547252), K. Asiedu (u18100156)

TASK 7: Analysis and Optimisation

Throughout the development of **CompareIt**, we continuously analyzed system behavior and looked for ways to optimize performance and usability. This helped us create a smoother, smarter, and more responsive application for users and admins alike.

Project Analysis

Early testing highlighted a need to help users quickly access fresh, relevant product data. To address this, we introduced a created_at attribute for each product record. This timestamp tracks when the product data was last added or updated in the system.

This allowed us to:

Display the latest deals or newly listed items prominently.

Enable sorting by freshness, so users can prioritize recent data.

Provide admins insights into data recency and trigger updates if needed.

Optimisation Techniques

To ensure optimal performance, especially as the product list grows, we applied several backend and frontend optimisations:

Indexed search fields in the database (e.g., product name, category) for faster query performance.

Lazy loading of product images to reduce initial page load times.

Client-side caching of previously fetched data to avoid redundant requests.

Pagination to minimize strain on the server and improve user navigation through large product lists.

Interpretation of Results

By tracking usage patterns (e.g., which filters are most used or which categories are frequently visited), we were able to make data-driven UI tweaks. These insights led to:

Improved default sorting based on user preferences.

Highlighting popular brands dynamically.

Better layout decisions for mobile responsiveness.

Overall, our optimisation efforts not only enhanced the system's technical performance but also contributed to a more intuitive and enjoyable user experience.

SQL Injection Prevention

Security was a top priority. We made sure all SQL queries are protected against injection attacks by:

- **Using prepared statements** with parameter binding
- **Validating and sanitizing all user inputs**
- Actively testing edge cases, such as entering malicious SQL in search fields

Example of a prepared statement in PHP:

```
$stmt = $pdo->prepare("SELECT * FROM products WHERE brand = ?");

$stmt->execute([$brand]);
```

This ensures that even if a user inputs a SQL snippet like '; DROP TABLE products; --, it won't be executed.

Complex SQL Queries

To improve filtering and sorting, we used **nested and joined SQL queries** that enable:

- Multi-condition product searches (e.g., price range + brand + category)
- Aggregated views of products by brand or retailer
- Dynamic sorting (lowest price, newest product, most popular)

These queries were optimized for performance, and tested against edge cases to ensure they return accurate, relevant results every time.

Example:

```
"SELECT
    p.product_id,
    p.title AS product_name,
    COUNT(*) AS save_count,
    COUNT(DISTINCT w.user_id) AS unique_users
FROM
    wishlists w
JOIN
    products p ON w.product_id = p.product_id
WHERE
    w.created_at >= DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY)
GROUP BY
    w.product_id
ORDER BY
    save_count DESC
LIMIT 5;"
```

BONUS MARKS

Data Visualisation Dashboard

To enhance the analytical capabilities of CompareIt, we developed a basic dashboard to visualize key insights and improve transparency within the system. This feature provides valuable context for both users and administrators.:

- **Dashboard**

    - Overview of key system metrics
    - Quick links to recent activity or reports
- **Products**
    - Manage the product catalog
    - Add, update, or delete product listings
    - View price comparisons across retailers
- **Retailers**
    - Manage retailer profiles and integrations
    - Track product offerings per retailer
- **Users**
    - Monitor user accounts
    - Enable or disable access
    - View registered users and activity logs

Security Features Implemented

As part of improving the robustness and security of the backend API, we conducted a **basic security audit** and implemented multiple security best practices. Below are the enhancements and their justification.

1. Secure Password Hashing with Salt

Why?

Storing plaintext or weakly hashed passwords is a major vulnerability. A secure system must protect user credentials against breaches and dictionary attacks.

What we did:

- We generate a **cryptographically secure salt** using random_bytes(8) and convert it to hexadecimal.
- Passwords are hashed using the **SHA-256** algorithm *combined with the salt*.
- The final stored format is hash:salt, e.g., 8c6976e5b5410415bde908bd4dee15dfb16c2bd0ad:salt12345678

Code example:

```
$salt = bin2hex(random_bytes(8));

$hashedPassword = hash('sha256', $data['password'] . $salt) . ':' . $salt;
```

Verification on login:

- Password is rehashed using the same salt and compared using hash_equals() to mitigate timing attacks.

2. Login Attempt Rate-Limiting

Why?
To prevent brute-force attacks where attackers try many passwords rapidly.

What we did:

- Introduced an in-database **rate-limiting mechanism**:
  - Track failed login attempts and timestamps per email.
  - Temporarily block login if the threshold is exceeded (e.g., 5 attempts within 5 minutes).

Implementation highlights:

```
// If > 5 failed attempts within 5 mins, block login
if ($attemptCount >= 5 && $sinceLastAttempt < 300) {
    throw new Exception('Too many failed attempts. Please try again later.', 429);
}
```

3. Monitoring via Error and Audit Logging

Why?
Logging is crucial for diagnosing security incidents and maintaining traceability.

What we did:

- All errors in registration and login are logged (optional file-based logging or DB-based logging).
- Failed login attempts are **logged with timestamps and IP address** for monitoring and analysis.

4. Session Management with API Keys

Why?
Proper session handling ensures that authenticated users are verified securely in future requests.

What we did:

- Each user receives a **secure, random 64-character API key** on registration/login.
- This API key is stored securely in the DB and returned to the client for use in authenticated requests.
- API keys are generated with random_bytes(32) ensuring high entropy and resistance to guessing.

Example:

```
$apiKey = bin2hex(random_bytes(32)); // 64-character hex string
```