**TraceIt**

**Database Design Document**

**V 4.0**

**By**

|  |  |  |
| --- | --- | --- |
| 1 | Usman Ghani | Roll No 24 |
| 2 | Muhammad Shariq | Roll No 39 |
| 3 | Rayan Badar | Roll No 18 |



**Department of Computer Sciences**

**Namal University Mianwali, Pakistan**

**Submission Date: 2nd June, 2025**

**REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Approved by** |
| 16/06/25 | V 3.0 | This version includes updates based on the approved Relational Schema creation Milestone submitted earlier, ERD, covering complete SQL Scripts and all other deliverables. | **Miss Asiya Batool** |
| 31/05/25 | V 2.0 | This version includes updates based on the approved ERD creation Milestone submitted earlier, covering complete Relational Schema design and all other deliverables. | **Miss Asiya Batool** |
| 18/04/25 | V 1.0 | This version includes updates based on the approved project proposal submitted earlier, covering complete ERD design, data dictionary, relationships, and in-text citations. | **Miss Asiya Batool** |

**TABLE OF CONTENTS**

[CHAPTER 1: PROJECT OVERVIEW 6](#_Toc200992429)

[1.1. INTRODUCTION 6](#_Toc200992430)

[1.2. PROBLEM STATEMENT 6](#_Toc200992431)

[1.3. PROJECT OBJECTIVES 6](#_Toc200992432)

[1.4. DOCUMENT OBJECTIVES 6](#_Toc200992433)

[CHAPTER 2: DETAILED DATABASE DESIGN 7](#_Toc200992434)

[2.1 ENTITY [3] 7](#_Toc200992436)

[2.2 DATA DICTIONARY [3] 7](#_Toc200992437)

[2.2.1 USER 7](#_Toc200992438)

[2.2.2 ADMIN 8](#_Toc200992439)

[2.2.3 ITEM 8](#_Toc200992440)

[2.2.4 REPORT 9](#_Toc200992441)

[2.2.5 VERIFICATION 9](#_Toc200992442)

[2.2.6 MATCHING\_LOG 9](#_Toc200992443)

[2.2.7 NOTIFICATION 10](#_Toc200992444)

[2.2.8 FEEDBACK 10](#_Toc200992445)

[2.2.9 CLAIM 10](#_Toc200992446)

[2.2.10 ITEM\_RETURN\_LOG 11](#_Toc200992447)

[2.3 RELATIONSHIPS 11](#_Toc200992448)

[2.4 ENTITY RELATIONSHIP DIAGRAM [3] 12](#_Toc200992449)

[CHAPTER 3: LOGICAL DATABASE DESIGN 13](#_Toc200992450)

[3.1 RELATIONAL SCHEMA 13](#_Toc200992451)

[3.2 FUNCTIONAL DEPENDENCIES 14](#_Toc200992452)

[3.3 NORMALIZATION 15](#_Toc200992453)

[3.3.1 STEP-BY-STEP NORMALIZATION PROCESS 16](#_Toc200992454)

[3.3.2 FINAL 3NF RELATIONS 16](#_Toc200992455)

[CHAPTER 4: PHYSICAL DATABASE DESIGN 17](#_Toc200992456)

[4.1 STRUCTURE OF THE TABLES 17](#_Toc200992457)

[4.1.1 ADMIN 17](#_Toc200992458)

[4.1.2 CLAIM 17](#_Toc200992459)

[4.1.3 FEEDBACK 17](#_Toc200992460)

[4.1.4 ITEM 18](#_Toc200992461)

[4.1.5 ITEM\_RETURN\_LOG 18](#_Toc200992462)

[4.1.6 MATCHING\_LOG 19](#_Toc200992463)

[4.1.7 USER 19](#_Toc200992464)

[4.1.8 NOTIFICATION 19](#_Toc200992465)

[4.1.9 VERIFICATION 20](#_Toc200992466)

[4.1.10 REPORT 20](#_Toc200992467)

[4.2 DATA SAMPLES INSIDE TABLES 20](#_Toc200992468)

[4.2.1 USER 21](#_Toc200992469)

[4.2.2 ITEM 21](#_Toc200992470)

[4.2.3 ADMIN 21](#_Toc200992471)

[4.2.4 ITEM\_RETURN\_LOG 22](#_Toc200992472)

[4.2.5 MATCHING\_LOG 22](#_Toc200992473)

[4.2.6 REPORT 23](#_Toc200992474)

[4.2.7 VERIFICATION 23](#_Toc200992475)

[4.2.8 NOTIFICATION 23](#_Toc200992476)

[4.2.9 CLAIM 24](#_Toc200992477)

[4.2.10 FEEDBACK 24](#_Toc200992478)

[4.3 QUERIES RESULTS 24](#_Toc200992479)

[CHAPTER 5: INTERFACE DESIGN 29](#_Toc200992480)

[5.1 LANGUAGE/FRAMEWORK 29](#_Toc200992481)

[5.1.1 KOTLIN: THE BACKBONE OF ANDROID LOGIC 29](#_Toc200992482)

[5.1.2 JETPACK COMPOSE: MODERN UI FRAMEWORK 29](#_Toc200992483)

[5.1.3 MYSQL INTEGRATION 30](#_Toc200992484)

[5.2 DATABASE CONNECTIVITY 30](#_Toc200992485)

[5.2.1 ARCHITECTURE OVERVIEW 30](#_Toc200992486)

[5.2.2 TECHNOLOGY STACK 30](#_Toc200992487)

[5.2.3 PHP BACKEND EXAMPLE 30](#_Toc200992488)

[5.2.4 KOTLIN CLIENT SIDE 31](#_Toc200992489)

[5.2.5 EXAMPLE 32](#_Toc200992490)

[5.2.6 ERROR HANDLING AND SECURITY 33](#_Toc200992491)

[5.3 STORED PROCEDURES AND FUNCTIONS: 33](#_Toc200992492)

[5.3.1 STORED PROCEDURE 1: InsertReport 33](#_Toc200992493)

[5.3.2 STORED PROCEDURE 2: LogItemReturn 34](#_Toc200992494)

[5.3.3 STORED FUNCTION 1: GetAverageFeedbackRating 34](#_Toc200992495)

[5.3.4 STORED FUNCTION 2 GetUserFeedbackCount 35](#_Toc200992496)

[5.3.5 ENHANCEMENT FOR GUI CLIENT 35](#_Toc200992497)

[5.4 INTERFACES 36](#_Toc200992498)

[5.4.1 SPLASH SCREEN 36](#_Toc200992499)

[5.4.2 ONBOARDING SCREENS 37](#_Toc200992500)

[5.4.3 AUTHENTICATION INTERFACE 38](#_Toc200992501)

[5.4.4 USER INTERFACE MODULE 39](#_Toc200992502)

[5.4.5 ADMIN CONTROL DASHBOARD 44](#_Toc200992503)

[CHAPTER 6: CONCLUSION 49](#_Toc200992504)

[6.1 LESSONS LEARNED 49](#_Toc200992505)

[6.1.1 TECHNICAL LESSONS 49](#_Toc200992506)

[6.1.2 PROJECT MANAGEMENT LESSONS 49](#_Toc200992507)

[6.1.3 TEAMWORK AND COMMUNICATION 49](#_Toc200992508)

[6.1.4 REAL WORLD APPLICATION INSIGHT 50](#_Toc200992509)

[6.2 CHALLENGES AND SOLUTIONS 50](#_Toc200992510)

[6.2.1 DATABASE CONNECTIVITY (MySQL Integration) 50](#_Toc200992511)

[6.2.2 DESIGNING OF RELATIONAL SCHEMA 50](#_Toc200992512)

[6.2.3 HANDLING ASYNCHRONOUS OPERATIONS 51](#_Toc200992513)

[6.2.4 UI STATE MANAGEMENT 51](#_Toc200992514)

[6.2.5 COORDINATION BETWEEN MULTIPLE CONTRIBUTORS 51](#_Toc200992515)

[6.2.6 LIMITED TESTING TIME FOR END TO END FLOW 51](#_Toc200992516)

[6.3 FUTURE WORK AND IMPROVEMENTS 52](#_Toc200992517)

[6.3.1 ROLE BASED ACCESS CONTROL (RBAC) 52](#_Toc200992518)

[6.3.2 REAL TIME NOTIFICATIONS 52](#_Toc200992519)

[6.3.3 MAP-BASED ITEM LOCATION 52](#_Toc200992520)

[6.3.4 QR CODE AND RFID SUPPORT 52](#_Toc200992521)

[6.3.5 IMAGE SIMILARITY MATCHING USING AI 52](#_Toc200992522)

[6.3.6 MULTILINGUAL SUPPORT 53](#_Toc200992523)

[6.3.7 OFFLINE DATA ENTRY AND SYNCRONIZATION 53](#_Toc200992524)

[6.4 FINAL THOUGHTS 53](#_Toc200992525)

[CHAPTER 7: REFERENCES 54](#_Toc200992526)

[7.1 APPENDIX: AI PROMPTS 54](#_Toc200992527)

# PROJECT OVERVIEW

## INTRODUCTION

The **TraceIt** Lost & Found Management System is a mobile application backed by a cloud database, designed to digitize the lost-and-found process within institutions like universities or offices. It enables users to report lost or found items, automates match suggestions using algorithm-based logic, and allows administrators to verify and approve item claims. The system promotes transparency, improves recovery rates, and streamlines the item tracking process by eliminating traditional, inefficient, paper-based methods [1] [2].

## PROBLEM STATEMENT

Lost and found items are frequently mismanaged [1] due to traditional manual logging systems that are inefficient, error-prone, and non-centralized. These systems lead to unclaimed items, delayed recoveries, and miscommunication. The lack of real-time notifications, proper verification, and data-driven tracking impairs the overall efficiency. The desired future state is a cloud-based mobile system that allows users to report, search, and recover items, while enabling administrators to verify and manage the overall process securely and efficiently [2].

## PROJECT OBJECTIVES

The primary objective of **TraceIt** is to digitize and optimize the lost-and-found process by leveraging modern database management, automation, and secure authentication methods.

The key objectives include:

1. **Providing a centralized cloud-based database** for storing and managing lost and found item records.
2. **Enabling users to report lost items** and search for found objects through a user-friendly mobile application.
3. **Providing an automated matching mechanism** that suggests potential item matches based on entered details.
4. **Providing admins with verification tools** to validate reports and ensure authenticity.
5. **Integrating real-time notifications** to alert users about potential item matches.

## DOCUMENT OBJECTIVES

This document aims to:

1. **Define** the core database entities and their relationships.
2. **Provide** a detailed data dictionary for each table.
3. **Describe** the logical structure of the ERD based on system functionalities.
4. **Act as a** guide for database implementation and future modifications.
5. **Ensure** clarity, consistency, and maintainability of the data model.

# DETAILED DATABASE DESIGN



## ENTITY [3]

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Entity Name** | **Description** |
| 01 | USER | Represents individuals who use the app to report or recover items. |
| 02 | ADMIN | Authorized personnel responsible for verifying item matches and reports. |
| 03 | ITEM | Represents a physical object reported as lost or found. |
| 04 | REPORT | Contains lost or found reports submitted by users or admins. |
| 05 | VERIFICATION | Records the admin’s decision for verifying item claims. |
| 06 | MATCHING\_LOG | Logs system-suggested item matches with a score and review status. |
| 07 | NOTIFICATION | Stores alerts sent to users/admins regarding match suggestions or actions. |
| 08 | FEEDBACK | Captures user feedback for item experience or system usage. |
| 09 | CLAIM | Shows the process of Claim by an user. |
| 10 | ITEM\_RETURN\_LOG | Stores the data of returning the item to user. |

## DATA DICTIONARY [3]

### **USER**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | UserID | Integer | Primary Key | Unique identifier for user. |
| 02 | Name | String | Not Null | Full name of the user. |
| 03 | Email | String | Unique | Email address. |
| 04 | PasswordHash | String | Not Null | Hashed password for security. |
| 05 | ContactInfo | String | Optional | Phone number or other contact. |

### **ADMIN**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | AdminID | Integer | Primary Key | Unique identifier for the admin. |
| 02 | Name | String | Not Null | Full name of the admin. |
| 03 | Email | String | Unique, Not Null | Admin’s email for login/notifications. |
| 04 | PasswordHash | String | Not Null | Encrypted password for authentication. |
| 05 | ContactInfo | String | Optional | Contact number of the admin. |

### **ITEM**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Attribute** | **Data Type** | **Constraint** | **Description** |
| 01 | ItemID | Integer | Primary Key | Unique identifier. |
| 02 | Category | String | Not Null | Type/category of item. |
| 03 | Description | String | Optional | Detailed explanation of the item. |
| 04 | Color | String | Optional | Primary color of the item. |
| 05 | Brand | String | Optional | Brand/manufacturer of the item. |
| 06 | Size | String | Optional | Item size or dimensions. |
| 07 | ItemCondition | String | Optional | Physical condition of the item. |
| 08 | Material | String | Optional | Material composition (e.g., leather). |
| 09 | TagNumber | String | Not Null | Unique label, barcode, or tag (optional). |
| 10 | Location | String | Not Null | Where the item was reported. |
| 11 | DateReported | DateTime | Not Null | Timestamp of report. |
| 12 | Status | String | Not Null | 'Returning', 'matched', ‘Resolved’ etc. |
| 13 | ImageURL | String | Optional | Link to item image (optional). |
| 14 | ItemType | String | Not Null | Lost or found. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | ReportID | Integer | Primary Key | Unique ID for each report. |
| 02 | ItemID | Integer | --- | References the reported item. |
| 03 | AdminID | Integer | --- | ID of Admin who made the report. |
| 04 | UserID | Integer | --- | ID of user who made report. |
| 05 | ReportedAt | DateTime | Not Null | Timestamp of when the report was filed. |

### **REPORT**

### **VERIFICATION**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | VerificationID | Integer | Primary Key | Unique ID for each verification request. |
| 02 | ClaimID | Integer | --- | References the claim item under verification. |
| 03 | AdminID | Integer | --- | Admin responsible for verifying the item. |
| 04 | VerifiedAt | DateTime | Optional | Timestamp of verification action. |
| 05 | Status | String | Not Null | ‘Pending’, ‘Approved’, or ‘Rejected’. |

### **MATCHING\_LOG**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | MatchID | Integer | Primary Key | Unique ID for each AI-based item match. |
| 02 | LostItemID | Integer | --- | Refers to the item reported as lost. |
| 03 | FoundItemID | Integer | --- | Refers to the item reported as found. |
| 04 | MatchScore | Float | Not Null | AI-calculated score of similarity. |
| 05 | MatchedAt | DateTime | Optional | Timestamp when the match was logged. |
| 06 | Status | String | Not Null | Match status: ‘Approved’, ‘Pending’ or ‘Rejected’. |

### **NOTIFICATION**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | NotificationID | Integer | Primary Key | Unique ID for each notification. |
| 02 | UserID | Integer | --- | ID of the user receiving the notification. |
| 03 | AdminID | Integer | --- | ID of the admin receiving the notification. |
| 04 | Message | String | Not Null | Content of the notification. |
| 05 | Timestamp | DateTime | Not Null | When the notification was generated. |
| 06 | ClaimID | Integer | --- | Information about matching\_log. |
| 07 | MatchID | Integer | --- | Information about items (lost or found). |

### **FEEDBACK**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | FeedbackID | Integer | Primary Key | Unique identifier for each feedback entry. |
| 02 | UserID | Integer | --- | User who gave the feedback. |
| 03 | FeedbackType | String | Not Null | Type: ‘Bug Report’, ‘Feature Request’, etc. |
| 04 | Comment | String | Optional | Text of the feedback. |
| 05 | Rating | Integer | Range 1–5 | Numeric rating provided by the user. |
| 06 | CreatedAt | DateTime | Not Null | Timestamp when feedback was submitted. |

### **CLAIM**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | ClaimID | Integer | Primary Key | Unique ID for each claim by user. |
| 02 | UserID | Integer | --- | ID of the user claiming the item. |
| 03 | MatchID | Integer | --- | ID of the Matching log that matches for specific item. |
| 04 | ClaimedAt | DateTime | Not Null | When the item is claimed, i.e., time. |

### **ITEM\_RETURN\_LOG**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Name** | **Data Type** | **Constraint** | **Description** |
| 01 | ReturnedID | Integer | Primary Key | Unique ID for each item return. |
| 02 | ItemID | Integer | --- | ID of the item being return. |
| 03 | AdminID | Integer | --- | ID of the admin recording the return. |
| 04 | Remarks | String | Not Null | Content of the return. |
| 05 | ReturnedAt | DateTime | Not Null | When the return was done, i.e., time. |
| 06 | Status | String | Not Null | Received or not received status. |
| 07 | UserID | Integer | --- | ID of user who receives item. |

## RELATIONSHIPS

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Participating Entities** | **Relation** | **Business Rule** |
| 01 | User, Report | User creates Report | A user can report multiple items; each report belongs to one user. |
| 02 | Admin, Verification | Admin approves Verification | An admin verifies item claims through the verification module. |
| 03 | Item, Matching\_Log | Item appears in Match | A single item can be part of multiple match logs (lost or found sides). |
| 04 | User, Notification | User receives Notification | A user can receive multiple notifications. |
| 05 | Admin, Notification | Admin receives Notification | An admin can receive multiple notifications. |
| 06 | User, Feedback | User gives Feedback | A user can give feedback on items or system experiences. |
| 07 | User, Item | User claims Items | A user can claim items if found matched. |
| 08 | Admin, Record | Admin Checks Record | An Admin can check records of items claimed by user. |

## ENTITY RELATIONSHIP DIAGRAM [3]

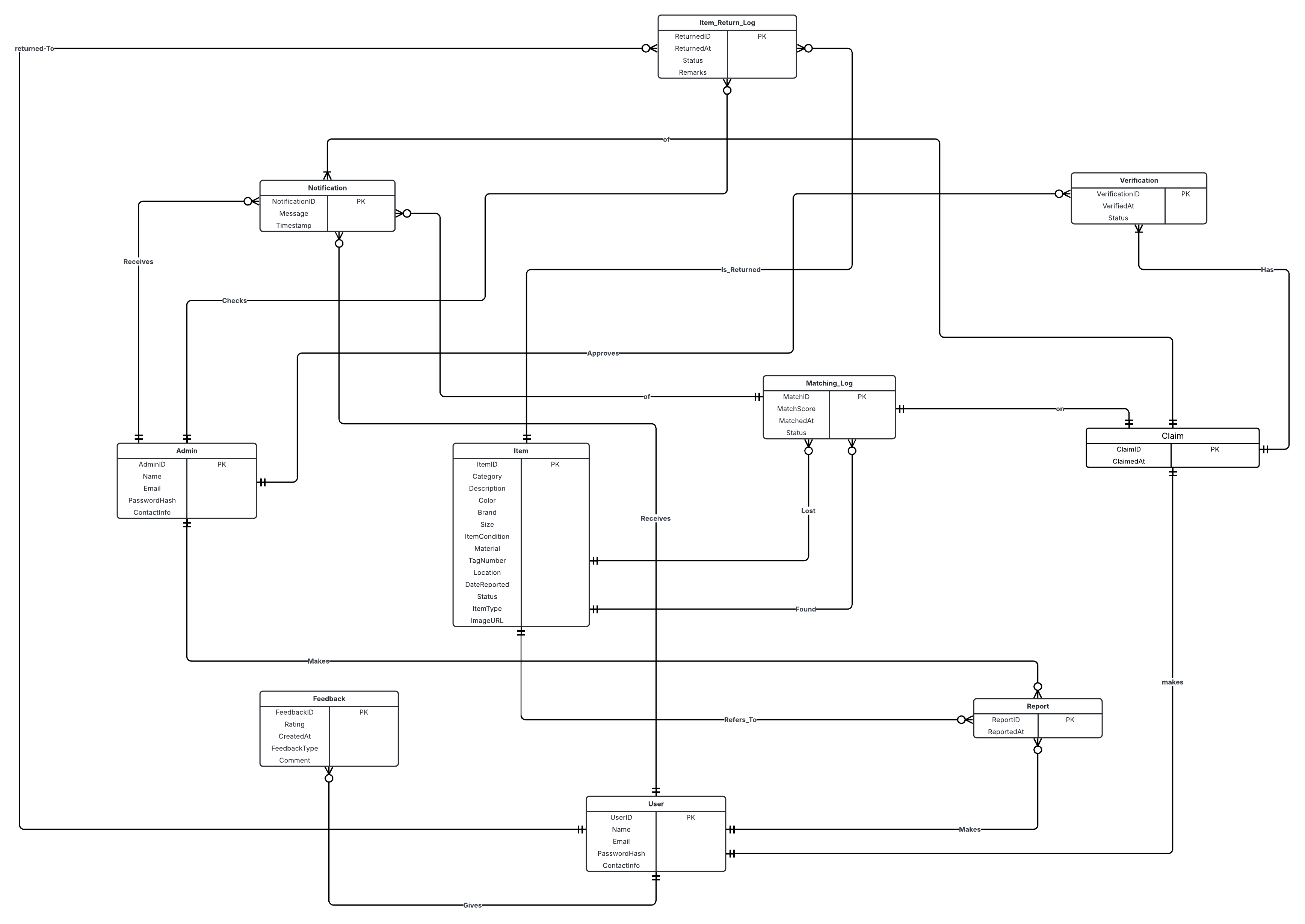


Fig 1.1: Entity Relationship Diagram (Version 2.0)

# CHAPTER 3: LOGICAL DATABASE DESIGN

* 1. **RELATIONAL SCHEMA**

The process of converting our **Entity-Relationship Diagram (ERD)** into a **Relational Schema** involved analyzing each entity, identifying primary keys, and mapping relationships through appropriate Foreign Keys. We ensured that the resulting schema was **fully normalized to Third Normal Form (3NF)** to eliminate redundancy and preserve data integrity.

Each entity in the ERD was translated into a relation (table), with attributes becoming columns and **Primary Keys (PK)** clearly defined. For relationships between entities, **Foreign Keys (FK)** were introduced to maintain referential integrity. Where applicable, **associative entities** were created to handle many-to-many relationships, such as between User, Item, and Claim.

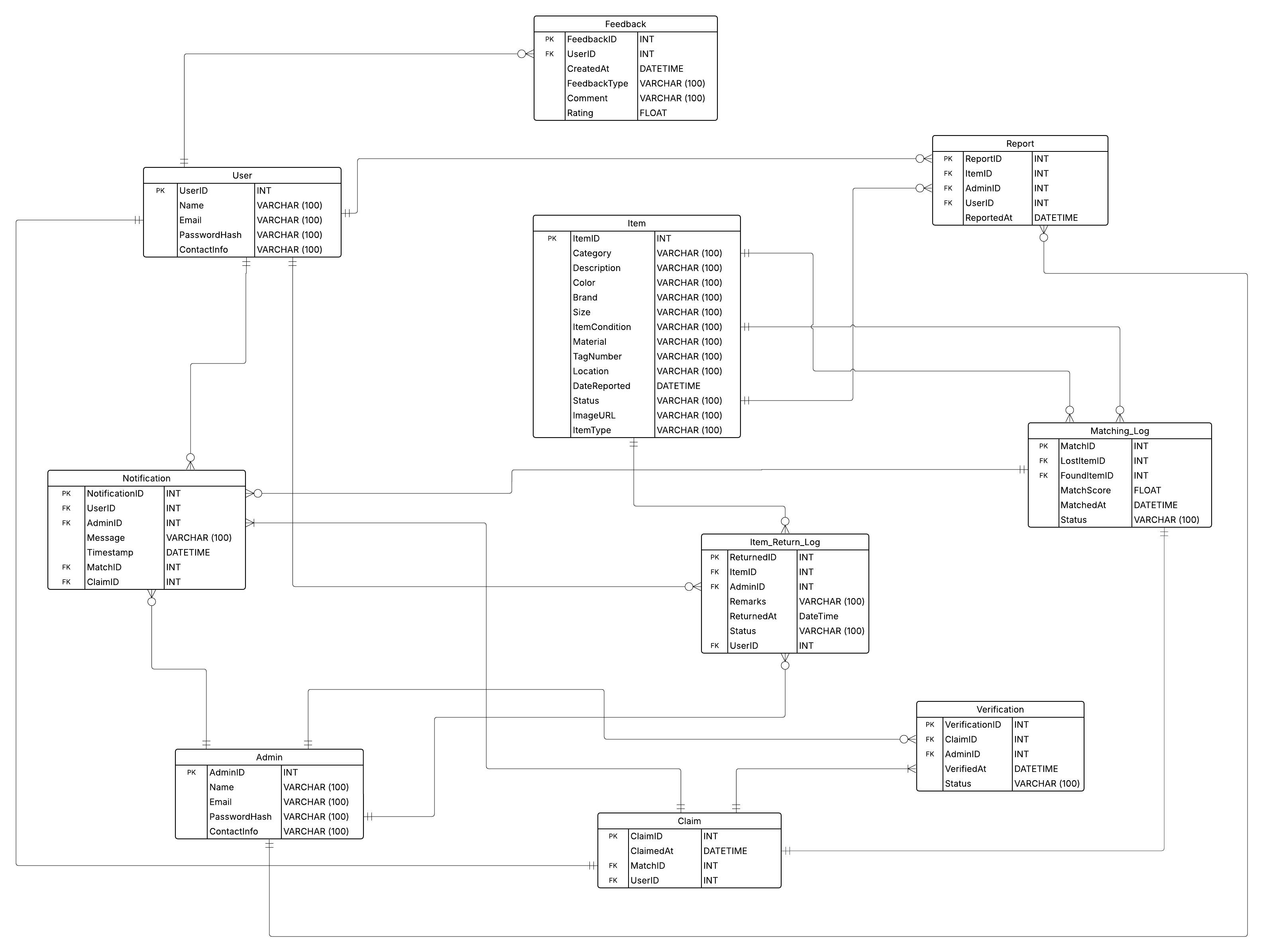


Fig 1.2: Relational Schema Diagram (Version 1.0)

This relational schema effectively captures the **conceptual design** of our system and is ready for implementation in any relational DBMS such as MySQL. It ensures consistency, supports complex queries, and enforces integrity across all interrelated data.

* 1. **FUNCTIONAL DEPENDENCIES**

In a relational database, a functional dependency (FD) exists when the value of one attribute (or a combination of attributes) determines the value of another attribute. Below, we list the functional dependencies for each table in our schema, along with concrete examples to clarify how they work in real-world scenarios.

#### **USER**

1. **Functional Dependencies**:

UserID → Name, Email, PasswordHash, ContactInfo

1. **Example**:

If UserID = 2, we can determine that Name = "Sara", Email = "sara@example.com".

#### **ADMIN**

1. **Functional Dependencies**:

AdminID → Name, Email, PasswordHash, ContactInfO

1. **Example**:

AdminID = 3 determines a unique admin and their details.

#### **ITEM**

1. **Functional Dependencies**:

ItemID → Category, Description, Color, Brand, Size, ItemCondition, Material, TagNumber, Location, DateReported, Status, ImageURL, ItemType

1. **Example**:

If ItemID = 10, we can determine that it's a "Red Nike Bag" reported on "2025-05-12".

#### **FEEDBACK**

1. **Functional Dependencies**:

FeedbackID → UserID, CreateAt, FeedbackType, Comment, Rating

1. **Example**:

FeedbackID = 7 implies UserID = 3, FeedbackType = "General", and Rating = 4.5

#### **REPORT**

1. **Functional Dependencies**:

ReportID → ItemID, AdminID, UserID, ReportedAt

1. **Example**:

ReportID = 5 determines which item was reported, by which user, and to which admin.

#### **NOTIFICATION**

1. **Functional Dependencies**:

NotificationID → UserID, AdminID, ClaimID, MatchID, Message, Timestamp

1. **Example**:

NotificationID = 4 tells us that user UserID = 8 received "Item matched" at 2025-05-15 10:00:00.

#### **MATCHING\_LOG**

1. **Functional Dependencies**:

MatchID → LostItemID, FoundItemID, MatchScore, MatchedAt, Status

1. **Example**:

MatchID = 12 tells us LostItemID = 5, FoundItemID = 11, MatchScore = 0.92.

#### **CLAIM**

1. **Functional Dependencies**:

ClaimID → ClaimedAt, MatchID, UserID

1. **Example**:

ClaimID = 8 tells us the claim was made by UserID = 4 for MatchID = 8 on a specific date.

#### **VERIFICATION**

1. **Functional Dependencies**:

VerificationID → ClaimID, AdminID, VerifiedAt, Status

1. **Example**:

VerificationID = 5 determines that AdminID = 2 verified ClaimID = 5.

#### **ITEM\_RETURN\_LOG**

1. **Functional Dependencies**:

ReturnID → ItemID, AdminID, ReturnedAt, Remarks, Status

1. **Example**:

ReturnID = 3 tells us that ItemID = 7 was returned by AdminID = 1 with remarks "Delivered".

## NORMALIZATION

The Lost & Found Management System (TraceIt) underwent a thorough normalization process using the identified functional dependencies. The goal was to achieve **Third Normal Form (3NF)** for all relations.

### **STEP-BY-STEP NORMALIZATION PROCESS**

#### **First Normal Form (1NF)**

All attributes contain only **atomic** values, and each record is **uniquely identifiable** by a primary key.  
All tables in the schema satisfy 1NF because:

1. There are no repeating groups or arrays.
2. Each column stores a single piece of data.

#### **Second Normal Form (2NF)**

All tables in the schema satisfy 2NF because:

1. Table is in 1NF.
2. All tables use **single-column primary keys.**
3. No **partial dependencies** (i.e., no non-prime attribute depends on a part of a composite key).

#### **Third Normal Form (3NF)**

All tables in the schema satisfy 3NF because:

1. Table is in 3NF.
2. No **transitive dependencies** (i.e., non-key attributes should not depend on other non-key attributes).
3. And all non-key attributes in each table depend only on the primary key.

### **FINAL 3NF RELATIONS**

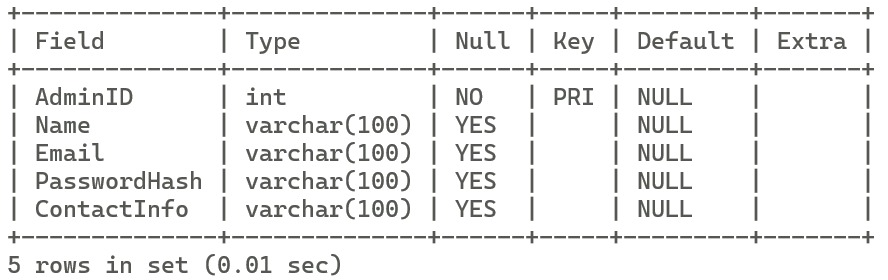
|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Relation Name** | **Primary Key** | **Foreign Keys** |
| 01 | **User** | UserID | ---- |
| 02 | **Admin** | AdminID | ---- |
| 03 | **Item** | ItemID | ---- |
| 04 | **Feedback** | FeedbackID | UserID → User |
| 05 | **Report** | ReportID | UserID → User, AdminID → Admin, ItemID → Item |
| 06 | **Notification** | NotificationID | UserID → User, AdminID → Admin, ClaimID → Claim, MatchID → Matching\_Log |
| 07 | **Matching\_Log** | MatchID | LostItemID → Item, FoundItemID → Item |
| 08 | **Claim** | ClaimID | MatchID → Matching\_Log, UserID → User |
| 09 | **Verification** | VerificationID | ClaimID → Claim, AdminID → Admin |
| 10 | **Item\_Return\_Log** | ReturnID | ItemID → Item, AdminID → Admin, UserID → User |

# CHAPTER 4: PHYSICAL DATABASE DESIGN

* 1. **STRUCTURE OF THE TABLES**

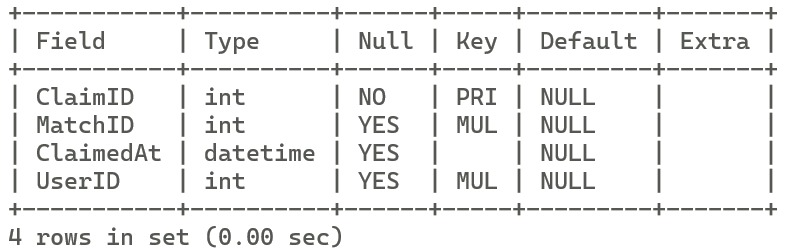
### **ADMIN**

**Query:** Describe Admin;



### **CLAIM**

**Query:** Describe Claim;



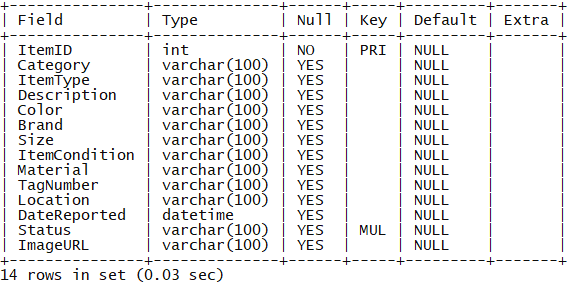
### **FEEDBACK**

**Query:** Describe Feedback;



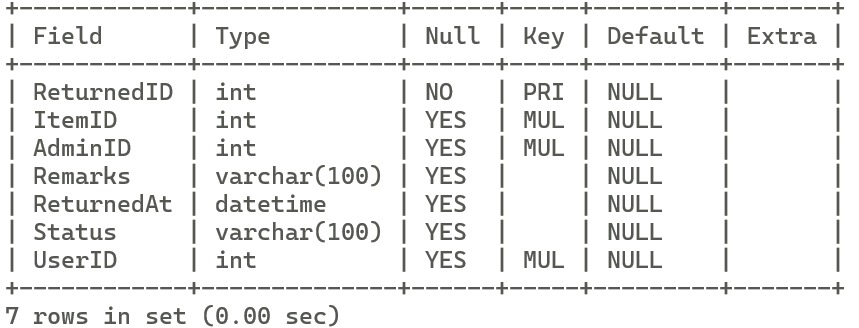
### **ITEM**

**Query:** Describe Item;



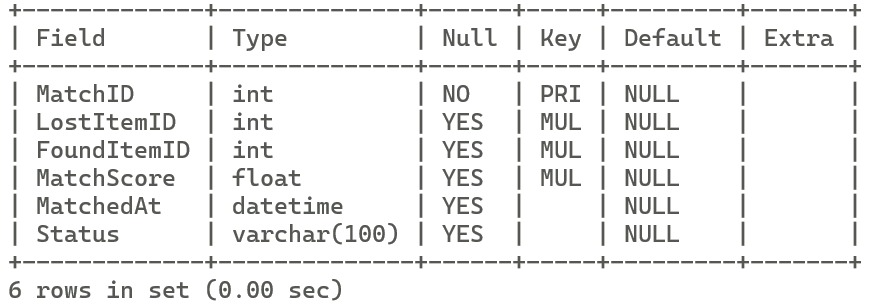
### **ITEM\_RETURN\_LOG**

**Query:** Describe Item\_Return\_Log;



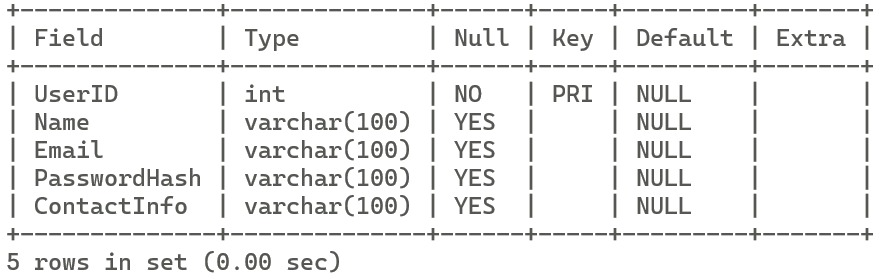
### **MATCHING\_LOG**

**Query:** Describe Matching\_Log;



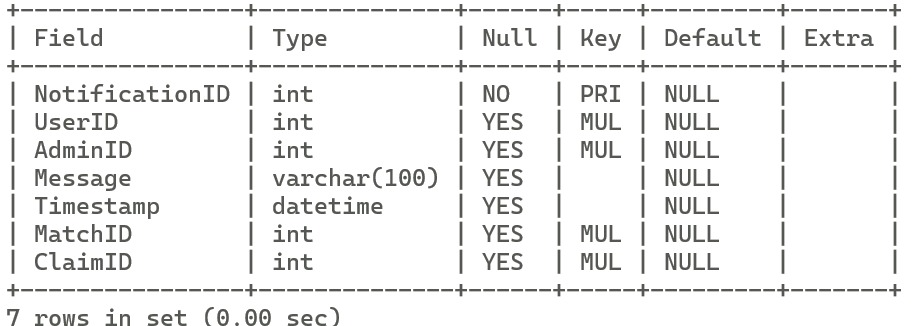
### **USER**

**Query:** Describe User;



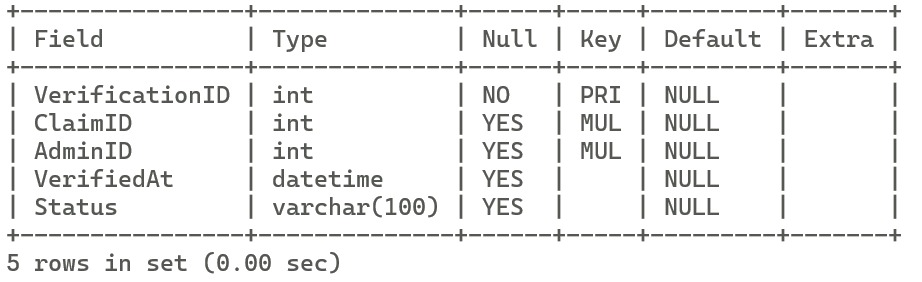
### **NOTIFICATION**

**Query:** Describe Notification;



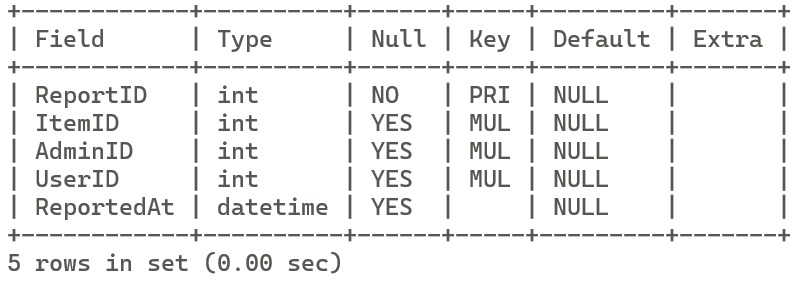
### **VERIFICATION**

**Query:** Describe Verification;



### **REPORT**

**Query:** Describe Report;

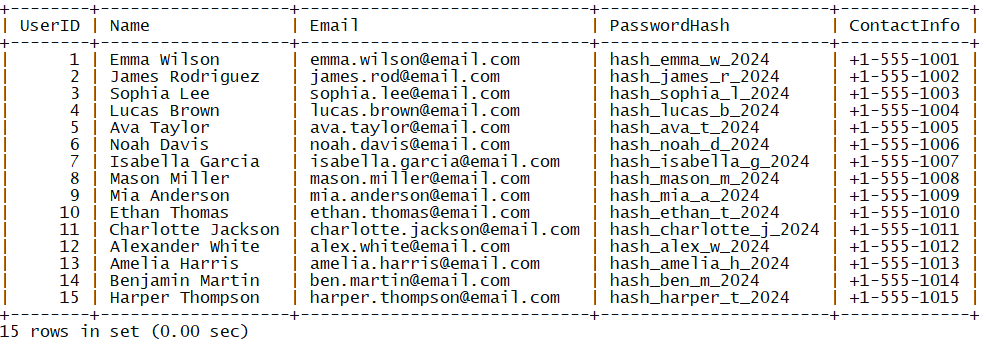


* 1. **DATA SAMPLES INSIDE TABLES**

To confirm that the database has been correctly populated, the following SELECT \* queries were executed on each table. The sample output shows that each table contains at least 15 rows of data, as required.

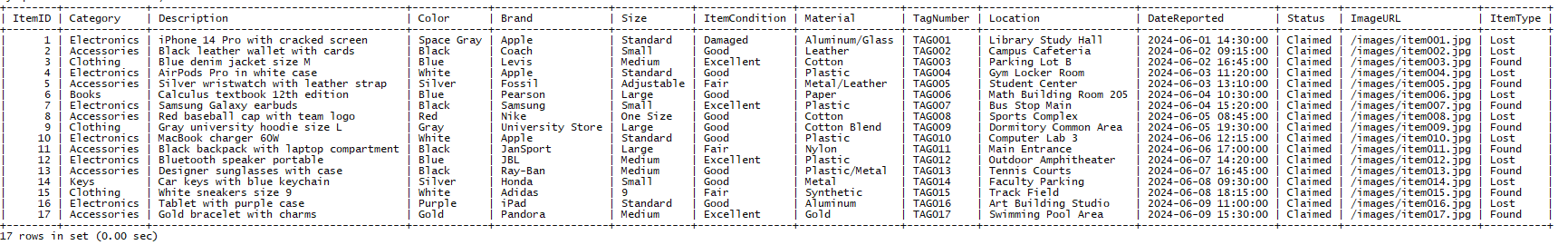
### **USER**

**Query:** Select \* from User;



### **ITEM**

**Query:** Select \* from Item;



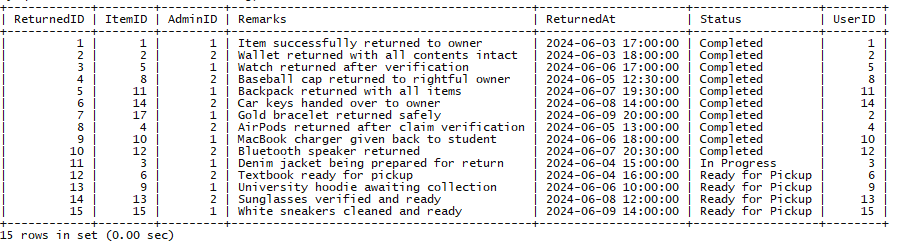
### **ADMIN**

**Query:** Select \* from Admin;



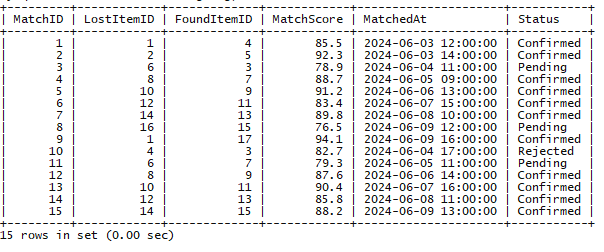
### **ITEM\_RETURN\_LOG**

**Query:** Select \* from Item\_Return\_Log;



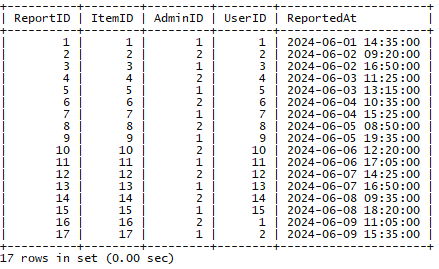
### **MATCHING\_LOG**

**Query:** Select \* from matching\_log;



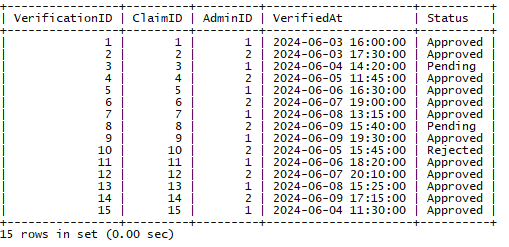
### **REPORT**

**Query:** Select \* from Report Limit 15;



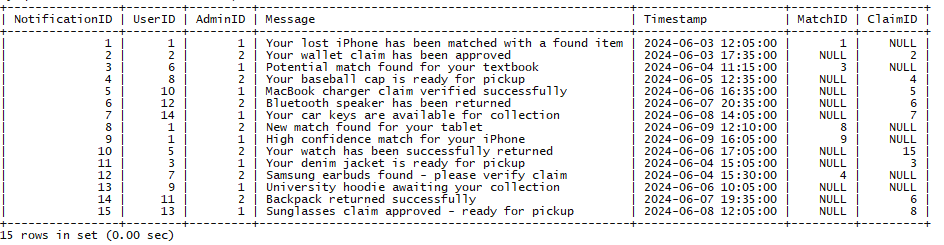
### **VERIFICATION**

**Query:** Select \* from Verification;



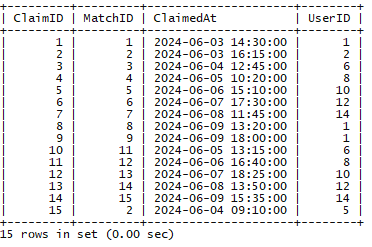
### **NOTIFICATION**

**Query:** Select \* from notification;



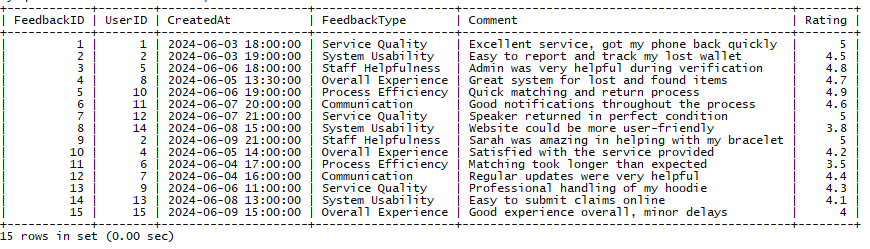
### **CLAIM**

**Query:** Select \* from claim;



### **FEEDBACK**

**Query:** Select \* from feedback;



* 1. **QUERIES RESULTS**

**Use Case 1:** List all lost items reported with user details

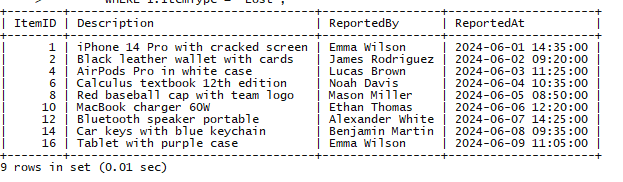
**Query:** SELECT I.ItemID, I.Description, U.Name AS ReportedBy, R.ReportedAt

        FROM Item AS I

        JOIN Report R ON I.ItemID = R.ItemID

        JOIN User U ON R.UserID = U.UserID

        WHERE I.ItemType = 'Lost';



**Use Case 2:** Show details of all claimed items with their matched items

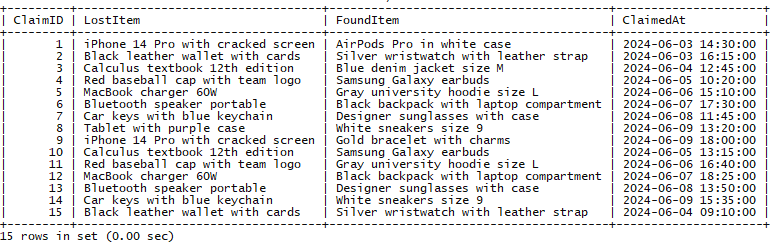
**Query:** SELECT C.ClaimID, L.Description AS LostItem, F.Description AS FoundItem, C.ClaimedAt

        FROM Claim C

        JOIN Matching\_Log M ON C.MatchID = M.MatchID

        JOIN Item L ON M.LostItemID = L.ItemID

        JOIN Item F ON M.FoundItemID = F.ItemID;



**Use Case 3:** List users who have given more than one feedback with average rating

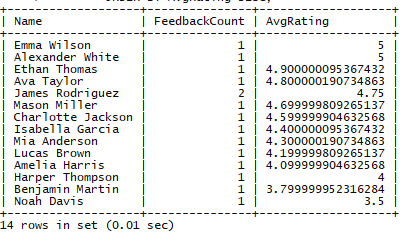
**Query:** SELECT U.Name, COUNT(F.FeedbackID) AS FeedbackCount, AVG(F.Rating) AS

        AvgRating FROM Feedback F

        JOIN User U ON F.UserID = U.UserID

        GROUP BY U.Name

        ORDER BY AvgRating DESC;



**Use Case 4:** Count number of items reported by each user, ordered by count descending.

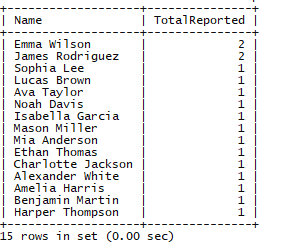
**Query:** SELECT U.Name, COUNT(R.ItemID) AS TotalReported

      FROM User U

        JOIN Report R ON U.UserID = R.UserID

        GROUP BY U.Name

        ORDER BY TotalReported DESC;



**Use Case 5:** List users who claimed items that were verified.

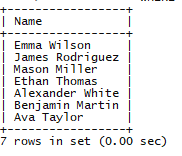
**Query:** SELECT Name FROM User

        WHERE UserID IN (

            SELECT C.UserID FROM Claim C

            JOIN Verification V ON C.ClaimID = V.ClaimID

            WHERE V.Status = 'Approved' );



**Use Case 6:** Show notifications sent by Admins related to verified claims only.

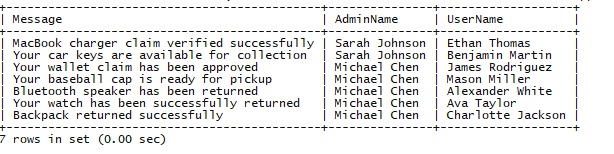
**Query:** SELECT N.Message, A.Name AS AdminName, U.Name AS UserName

        FROM Notification N

        JOIN Admin A ON N.AdminID = A.AdminID

        JOIN User U ON N.UserID = U.UserID

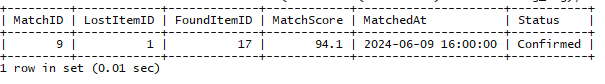
      WHERE N.ClaimID IN (SELECT ClaimID FROM Verification WHERE Status = 'Approved');



**Use Case 7:** Find item(s) with the highest match score.

**Query:** SELECT \* FROM Matching\_Log

        WHERE MatchScore = (SELECT MAX(MatchScore) FROM Matching\_Log);



**Use Case 8:** Show the latest 5 returned items with their users and admins

**Query:** SELECT IRL.ReturnedID, I.Description, U.Name AS ReturnedTo, A.Name AS HandledBy, IRL.ReturnedAt

        FROM Item\_Return\_Log IRL

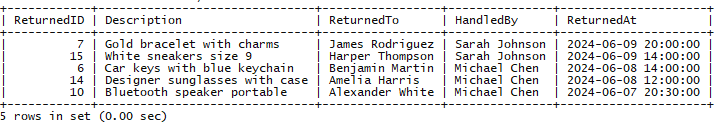
        JOIN Item I ON IRL.ItemID = I.ItemID

        JOIN User U ON IRL.UserID = U.UserID

        JOIN Admin A ON IRL.AdminID = A.AdminID

        ORDER BY IRL.ReturnedAt DESC

        LIMIT 5;



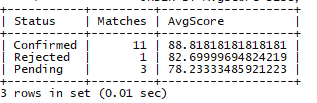
**Use Case 9:** Average match score for each status type in Matching\_Log

**Query:** SELECT Status, COUNT(\*) AS Matches, AVG(MatchScore) AS AvgScore

        FROM Matching\_Log

        GROUP BY Status

        ORDER BY AvgScore DESC;

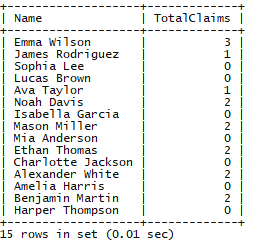


**Use Case 10:** List users with the total number of claims they made (include users with 0 claims)

**Query:** SELECT U.Name, (

            SELECT COUNT(\*) FROM Claim C WHERE C.UserID = U.UserID) AS TotalClaims

        FROM User U;



# CHAPTER 5: INTERFACE DESIGN

## LANGUAGE/FRAMEWORK

The Graphical User Interface (GUI) client of the TraceIt application has been developed using **Kotlin** as the primary programming language, in conjunction with **Jetpack Compose** for designing the user interface. This modern and efficient technology stack was strategically chosen to facilitate the development of a highly responsive, maintainable, and user-friendly Android application. Additionally, the application is integrated with a **MySQL database** to support persistent data storage and robust backend operations.

### **KOTLIN: THE BACKBONE OF ANDROID LOGIC**

Kotlin, officially supported by Google for Android development, provides a concise, expressive, and type-safe syntax that greatly enhances developer productivity. Some of the key features that make Kotlin ideal for this project include:

1. **Null Safety:** Minimizes the risk of runtime exceptions by enforcing null checks at compile time.
2. **Coroutines:** Enable smooth handling of asynchronous tasks, such as database communication, without blocking the main thread.
3. **Java Interoperability:** Seamlessly integrates with existing Java libraries, enhancing flexibility and library access.
4. **Readable and Maintainable Code:** Reduces boilerplate, which accelerates development and debugging.

These features ensure a smooth and bug-resistant implementation of core application logic and data flow between the GUI and the MySQL backend.

### **JETPACK COMPOSE: MODERN UI FRAMEWORK**

Jetpack Compose is Android’s latest UI toolkit that allows developers to build native interfaces using declarative Kotlin code. Its introduction has revolutionized how UIs are designed by eliminating XML layouts and enabling a fully programmatic UI structure.

Key benefits include:

1. **Declarative UI Development:** UI is updated automatically in response to state changes, ensuring better user interaction without manual UI refreshes.
2. **Composable Functions:** Modular and reusable UI components that make code scalable and maintainable.
3. **Live Previews and Faster Prototyping:** Supports rapid development with real-time feedback.
4. **Material Design Integration:** Built-in support for Material UI components enhances visual consistency and usability.

Jetpack Compose’s architecture aligns perfectly with the dynamic needs of TraceIt, particularly when interacting with backend systems to display, update, and manage data records related to lost and found items.

### **MYSQL INTEGRATION**

While the frontend is Android-based, the GUI communicates with a **MySQL database** via API endpoints. This integration enables real-time operations like item registration, report submissions, claim processing, and feedback storage. The GUI sends and retrieves structured data via secure HTTP requests or direct queries, ensuring accurate synchronization with the backend schema.

* 1. **DATABASE CONNECTIVITY**

To enable seamless interaction between the **TraceIt Android GUI client** and the **MySQL database**, a RESTful API layer was developed using **PHP**. This approach abstracts the complexity of direct SQL communication from the mobile client and provides a secure, scalable method to perform CRUD operations on the backend.

### **ARCHITECTURE OVERVIEW**

The connectivity flow is structured as follows:

Kotlin (Android App) ↔ PHP APIs ↔ MySQL Database

### **TECHNOLOGY STACK**

1. **Frontend:** Kotlin (Jetpack Compose UI)
2. **Backend APIs:** PHP (with MySQL or PDO for database operations)
3. **Database:** MySQL

This design ensures loose coupling between client and server logic, allowing independent updates and better security.

### **PHP BACKEND EXAMPLE**

A PHP API to fetch user data from the database:

<?php

header("Content-Type: application/json");

include 'db\_connect.php';

$sql = "SELECT \* FROM User";

$result = mysqli\_query($conn, $sql);

$response = [];

while ($row = mysqli\_fetch\_assoc($result)) {

$response[] = $row;

}

echo json\_encode($response);

mysqli\_close($conn);

?>

**db\_connect.php:**

<?php

$host = "localhost";

$username = "root";

$password = "namal123";

$database = "TraceIt";

$conn = mysqli\_connect($host, $username, $password, $database);

if (!$conn) {

die(json\_encode(["error" => "Database connection failed: " . mysqli\_connect\_error()]));

}

?>

### **KOTLIN CLIENT SIDE**

The Android app uses **Ktor** and **Retrofit** to make HTTP requests to the PHP API. Here's an example using Volley to fetch all users from the /getUsers.php endpoint.

val url = "http://yourserver.com/getUsers.php"

val requestQueue = Volley.newRequestQueue(context)

val jsonArrayRequest = JsonArrayRequest(Request.Method.GET, url, null,

{ response ->

for (i in 0 until response.length()) {

val user = response.getJSONObject(i)

val name = user.getString("Name")

val email = user.getString("Email")

Log.d("User", "Name: $name, Email: $email")}

},

{ error ->

Log.e("VolleyError", "Error: ${error.message}")

})

requestQueue.add(jsonArrayRequest)

### **EXAMPLE**

**PHP API (addItem.php):**

<?php

include 'db\_connect.php';

$category = $\_POST['Category'];

$description = $\_POST['Description'];

$color = $\_POST['Color'];

$status = $\_POST['Status']; // Example: "Matched"

$itemType = $\_POST['ItemType']; // Example: "Lost"

$sql = "INSERT INTO Item (ItemID, Size, ItemCondition, Material, Brand, TagNo., Location, DateReported, ImageURL, Category, Description, Color, Status, ItemType)

VALUES (‘$ItemID’, ‘$Size’, ‘$ItemCondition’, ‘$Material’, ‘$Brand’, ‘$TagNo’., ‘$Location’, ‘$DateReported’, ‘’$ImageURL, '$category', '$description', '$color', '$status', '$itemType')";

if (mysqli\_query($conn, $sql)) {

echo json\_encode(["success" => true]);

} else {

echo json\_encode(["success" => false, "error" => mysqli\_error($conn)]);

}

mysqli\_close($conn);

**Kotlin (using Retrofit):**

interface ApiService {

@FormUrlEncoded

@POST("addItem.php")

suspend fun addItem(

@Field("ItemID") category: String,

@Field("Size") description: String,

@Field("Category") color: String,

@Field("ItemCondition") status: String,

@Field("ItemType") itemType: String

@Field("Material") category: String,

@Field("Brand") description: String,

@Field("Tagno.") color: String,

@Field("Location") status: String,

@Field("DateReported") itemType: String

@Field("ImageURL") category: String,

@Field("Description") description: String,

@Field("Color") color: String,

@Field("Status") status: String,

): Response<ApiResponse>

}

### **ERROR HANDLING AND SECURITY**

This layered architecture ensures that the TraceIt application maintains a **clean separation of concerns**, offers **modularity**, and supports future expansion. The use of PHP for database handling provides simplicity and speed, while Kotlin ensures a responsive and modern user experience on the client side.

1. All PHP responses are in **JSON** format for consistency.
2. PHP handles SQL errors and returns appropriate error messages.
3. Prepared statements or input validation should be implemented to **prevent SQL injection**.
4. API endpoints can be protected via **JWT tokens or session validation**.

## STORED PROCEDURES AND FUNCTIONS:

To improve the maintainability, reusability, and performance of backend database operations in the TraceIt system, **stored procedures and functions** were implemented in **MySQL**. These database-side scripts encapsulate complex queries and business logic, reducing redundancy and centralizing control over critical operations.

### **STORED PROCEDURE 1: InsertReport**

**Purpose:**  
To simplify the process of reporting a lost or found item by encapsulating the logic of inserting a report into the Report table.

**Script:**

DELIMITER //

CREATE PROCEDURE InsertReport (

IN p\_ItemID INT,

IN p\_AdminID INT,

IN p\_UserID INT,

IN p\_ReportedAt DATETIME

)

BEGIN

INSERT INTO Report (ItemID, AdminID, UserID, ReportedAt)

VALUES (p\_ItemID, p\_AdminID, p\_UserID, p\_ReportedAt);

END;

//

DELIMITER ;

**Usage:**  
Used by the PHP API when a user reports a new item, reducing SQL code duplication in the backend.

### **STORED PROCEDURE 2: LogItemReturn**

**Purpose:**  
To record a return transaction in the Item\_Return\_Log when a claimed item is returned.

**Script:**

DELIMITER //

CREATE PROCEDURE LogItemReturn (

IN p\_ItemID INT,

IN p\_AdminID INT,

IN p\_Remarks VARCHAR(100),

IN p\_ReturnedAt DATETIME,

IN p\_Status VARCHAR(100),

IN p\_UserID INT

)

BEGIN

INSERT INTO Item\_Return\_Log (ItemID, AdminID, Remarks, ReturnedAt, Status, UserID)

VALUES (p\_ItemID, p\_AdminID, p\_Remarks, p\_ReturnedAt, p\_Status, p\_UserID);

END;

//

DELIMITER ;

**Usage:**  
Called by the admin panel when an item is officially returned to the user. Helps track return history and status updates.

### **STORED FUNCTION 1: GetAverageFeedbackRating**

**Purpose:**  
To calculate the average rating given by users in the Feedback table.

**Script:**

DELIMITER //

CREATE FUNCTION GetAverageFeedbackRating()

RETURNS FLOAT

DETERMINISTIC

BEGIN

DECLARE avg\_rating FLOAT;

SELECT AVG(Rating) INTO avg\_rating FROM Feedback;

RETURN avg\_rating;

END;

//

DELIMITER ;

**Usage:**  
This function is useful for displaying system-wide user satisfaction metrics in the admin dashboard or analytics.

### **STORED FUNCTION 2 GetUserFeedbackCount**

**Purpose:**  
To return the total number of feedback entries submitted by a particular user.

**Script:**

DELIMITER //

CREATE FUNCTION GetUserFeedbackCount(p\_UserID INT)

RETURNS INT

DETERMINISTIC

BEGIN

DECLARE feedback\_count INT;

SELECT COUNT(\*) INTO feedback\_count FROM Feedback WHERE UserID = p\_UserID;

RETURN feedback\_count;

END;

//

DELIMITER ;

**Usage:**  
Can be used in the user profile section to show how active the user is in submitting feedback or for gamification features.

### **ENHANCEMENT FOR GUI CLIENT**

These stored procedures and functions:

1. **Encapsulate** business logic for consistent behavior.
2. **Minimize backend code** in PHP by pushing logic to the DBMS.
3. Improve **performance** by reducing multiple network round-trips.
4. Provide **reusability** across multiple PHP API endpoints.
5. **Secure** data logic by reducing direct table manipulations in raw SQL.

## INTERFACES

### **SPLASH SCREEN**



Figure 1:1st Splash Screen

**DESCRIPTION:**This is the initial screen users see when launching the application. It provides an engaging animated introduction to the app, featuring dynamic scaling, fading, and subtle glitch effects to create a modern and visually appealing entry point.

### **ONBOARDING SCREENS**

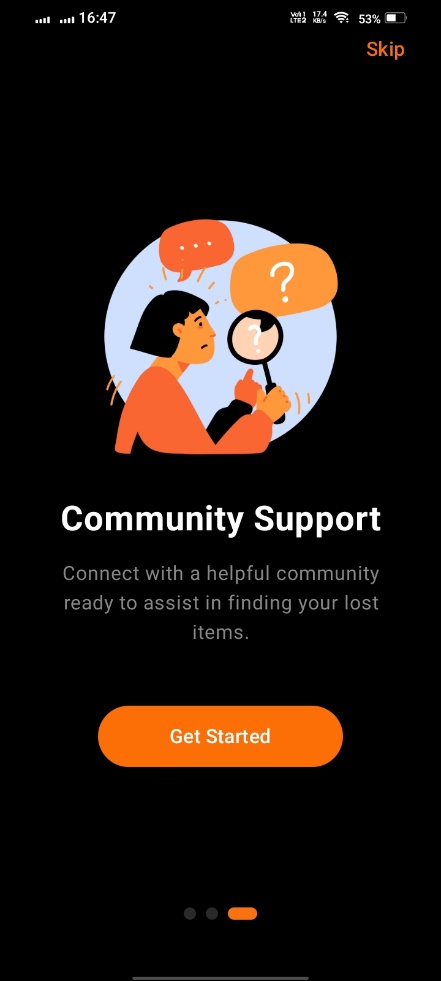


Figure 1:1st Onboarding Screen

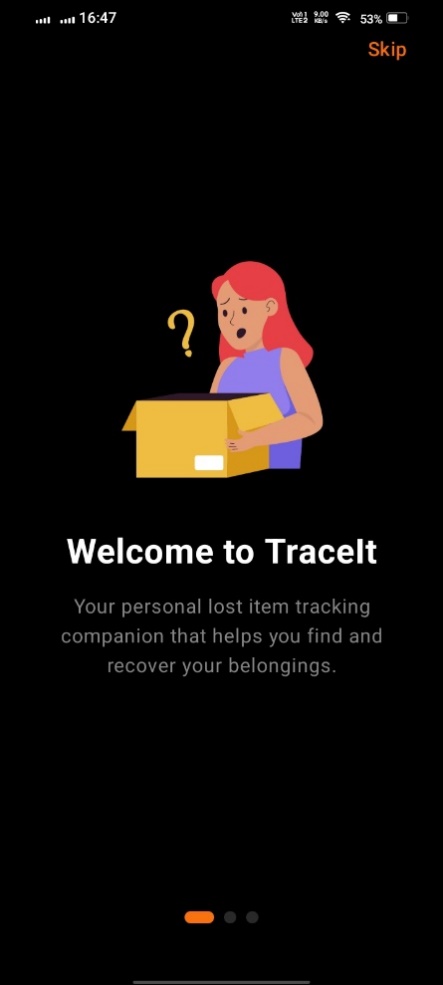


Figure 2: 2nd Onboarding Screen

Figure 3: 3rd Onboarding Screen

**DESCRIPTION:**This is the initial set of screens new users encounter upon their first launch of the application. It guides them through the app's main features and benefits, helping them understand how to use the platform effectively before they proceed to the main functionalities.

### **AUTHENTICATION INTERFACE**

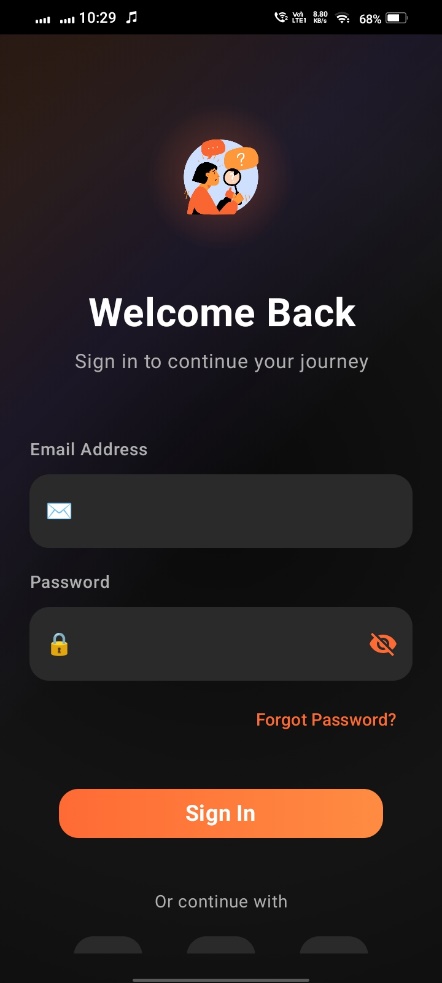


Figure 2: 2nd Authentication Interface

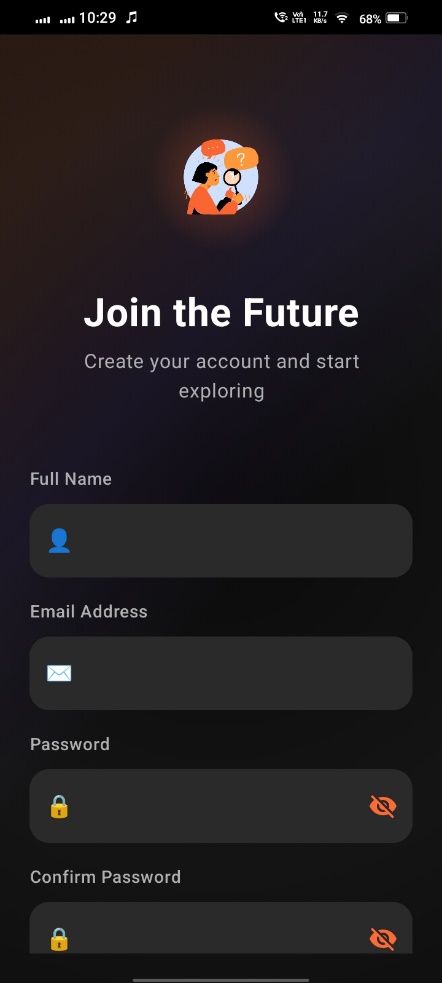


Figure 1: 1st authentication Interface

**DESCRIPTION:**This screen handles all user authentication processes, including user registration (sign-up) and login. It supports various authentication methods, such as email/password, Google Sign-In, and Facebook Sign-In. Additionally, it incorporates a password reset functionality and a mechanism to differentiate and authenticate administrative users.

### **USER INTERFACE MODULE**

#### **Home Screen**

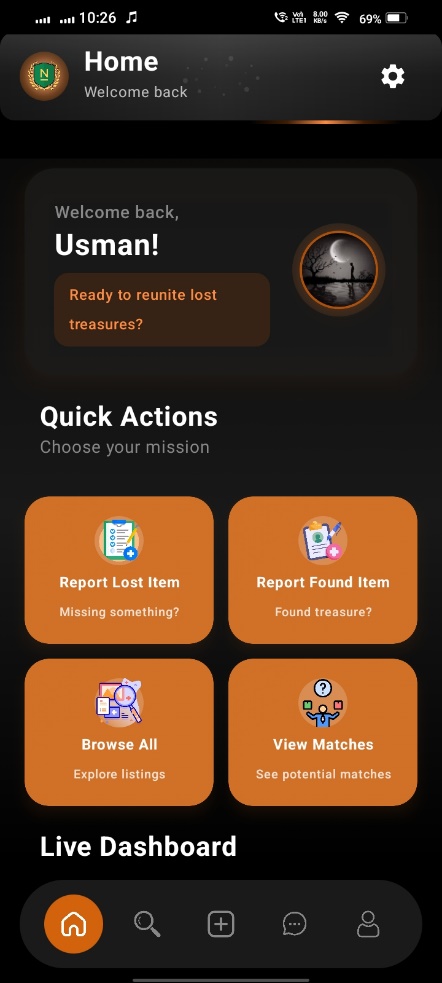


Figure 1: Home Screen

**DESCRIPTION:**This is the main dashboard for regular users, providing quick access to essential features. It might display a personalized feed of relevant lost and found items, recent activity, or notifications, serving as the primary landing page after login.

#### **Explore Screen**

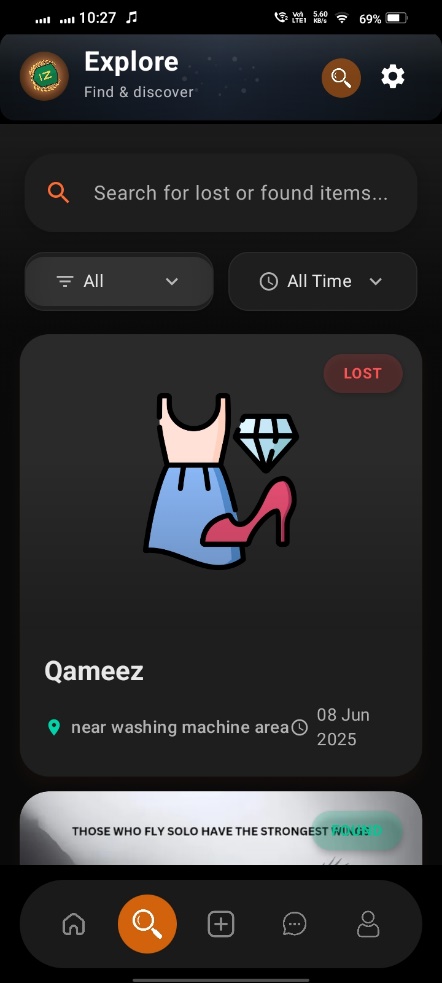


Figure 1: Explore Screen

**DESCRIPTION:**This screen allows users to discover lost and found items. It features search and filter functionalities to help users find specific items based on criteria such as item type, category, location, and date. Users can browse available items and potentially find their lost possessions or report found ones.

#### **Add Item Screen**

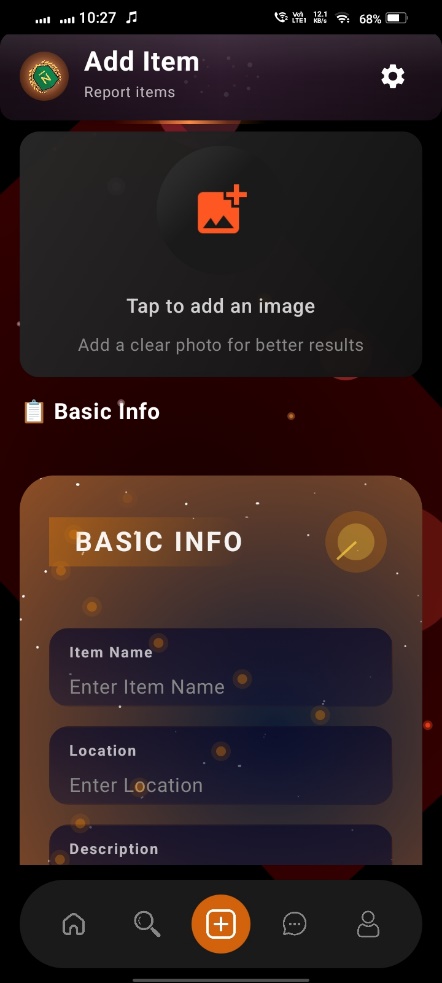


Figure 1: Add Item Screen

**DESCRIPTION:**This screen allows regular users to create new listings for lost or found items. It provides a detailed form for users to input information such as item name, description, location, date, category, color, material, size, brand, tag number, and item condition. It also supports uploading images of the item and includes robust form validation to ensure data accuracy before submission. The screen features animated background elements for a visually engaging user experience.

#### **Chat Screen**

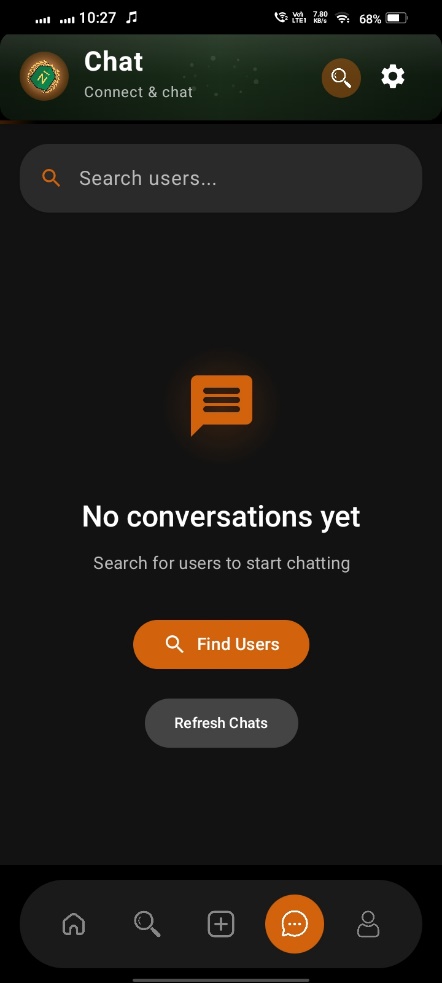


Figure 1: Chat Screen

**DESCRIPTION:**This screen lists all active chat conversations for the current user. It provides an overview of ongoing communications, allowing users to select a chat to view its detailed history and continue the conversation.

#### **Profile Screen**

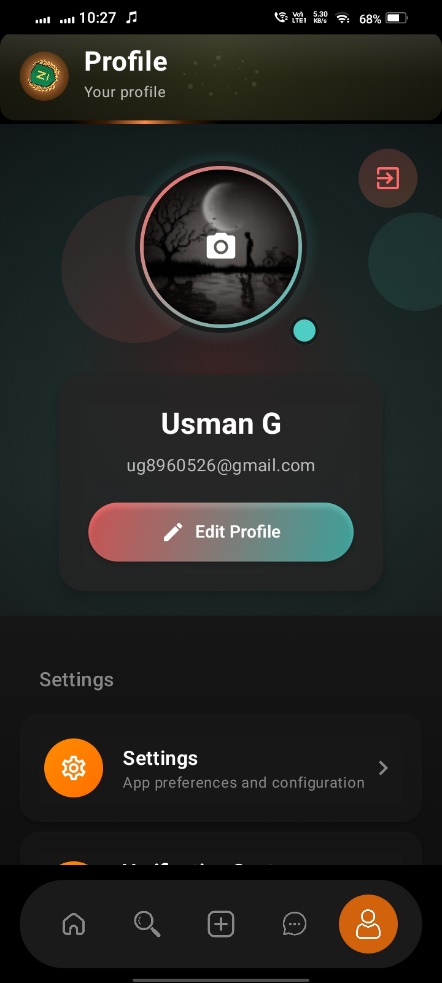


Figure 1: Profile Screen

**DESCRIPTION:**This screen is dedicated to user profile management. Users can update their display name and profile picture. It also displays the user's current administrator status, if applicable, and provides direct access to various other app settings, help and support, and feedback options.

### **ADMIN CONTROL DASHBOARD**

#### **Admin Dashboard**

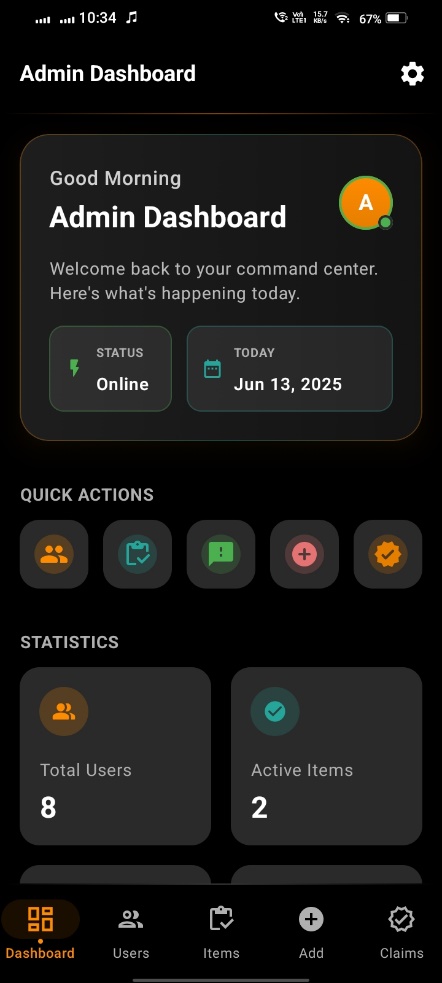


Figure 1: Admin Dashboard

**DESCRIPTION:**This is the central hub for administrators, providing an overview of key statistics such as user count, item count, and claims. It offers quick navigation to other administrative sections like user management, item listings, feedback, and verification, allowing admins to efficiently manage the app's operations.

#### **Verification Screen**

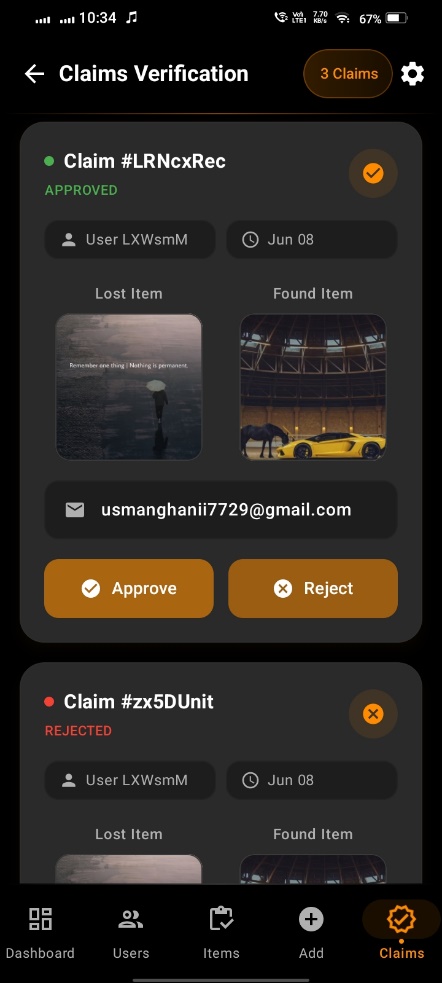


Figure 1: Verification Screen

**DESCRIPTION:**This screen is an administrative tool that empowers administrators to manage and verify claims related to lost and found items. It allows administrators to review claims, approve them if the details match, or reject them if discrepancies are found, ensuring the authenticity of item claims.

#### **User Management Screen**

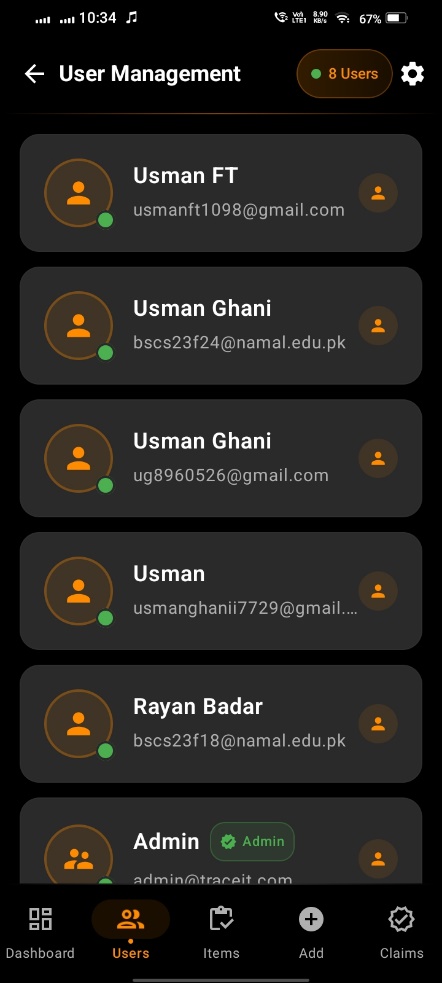


Figure 1: User Management Screen

**DESCRIPTION:**This screen is designed for administrators to view a comprehensive list of all registered users in the application. It displays essential user information, including their count and verification status. The screen also offers convenient navigation to other administrative features through a dedicated bottom navigation bar.

#### **Item Management Screen**

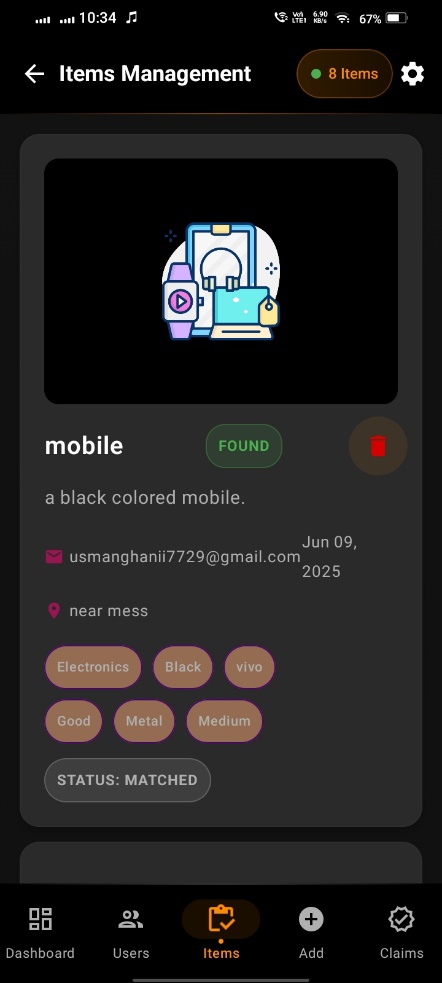


Figure 1: Item Management Screen

**DESCRIPTION:**This screen is primarily for administrators, displaying a list of all lost and found items within the application. Administrators can view details of each item and manage them. The screen also incorporates a bottom navigation bar, providing easy access to other admin-specific sections like Users, Reports, and Verification.

#### **Feedback Screen**

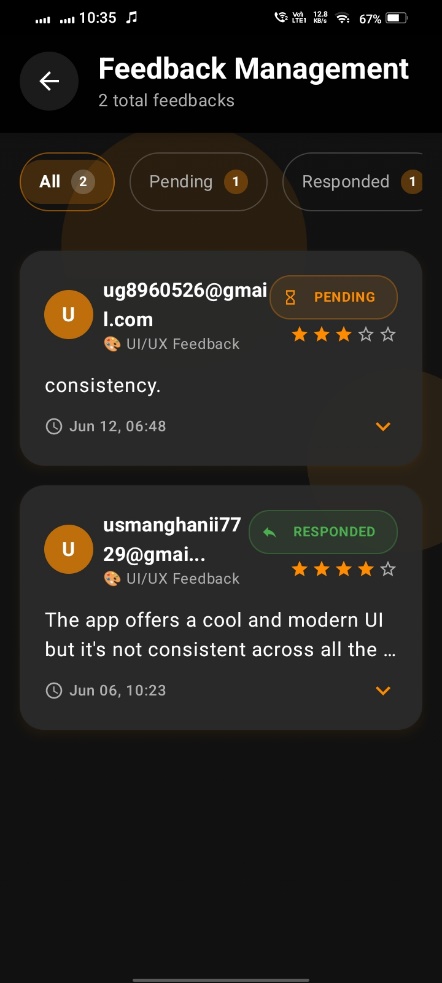


Figure 1: Feedback Screen

**DESCRIPTION:**This screen allows administrators to view all feedback submissions from users. It provides a centralized place for reviewing user comments, suggestions, and bug reports, enabling administrators to track and address user feedback efficiently. It also includes filtering options to categorize feedback (e.g., "All," "Pending," "Responded") and displays key statistics.

# CHAPTER 6: CONCLUSION

## ****LESSONS LEARNED****

Throughout the development of the TraceIt project, our team acquired a range of valuable lessons spanning technical, collaborative, and managerial domains. The process of designing, developing, and deploying a functional lost and found management system provided us with deeper insights into real-world software engineering practices.

Below are the key takeaways:

### **TECHNICAL LESSONS**

1. **Full-Stack Integration**: One of the most significant learning curves was integrating the Android front end with a MySQL database using PHP-based REST APIs. We learned how to manage API endpoints, handle JSON responses, and maintain secure data transfer between the client and server.
2. **Database Design**: Crafting an efficient relational schema helped us understand normalization, referential integrity, and the practical benefits of stored procedures and indexing for performance optimization.
3. **Jetpack Compose Mastery**: Developing the GUI in Jetpack Compose enhanced our understanding of declarative UI paradigms. We learned how to structure Composables, manage state reactively, and implement reusable UI components effectively.
4. **Asynchronous Programming**: Leveraging Kotlin coroutines taught us how to perform background operations (like network or database calls) without blocking the main UI thread, enhancing app responsiveness and user experience.

### **PROJECT MANAGEMENT LESSONS**

1. **Version Control Discipline**: Regular usage of Git enabled smoother collaboration and conflict resolution. We learned the importance of feature branches, meaningful commits, and pull requests for team-based development.
2. **Requirement Tracing**: Maintaining clear documentation of features, user stories, and milestones helped us stay aligned with project objectives and evaluate our progress consistently.
3. **Milestone Planning**: Breaking the project into iterative deliverables ensured that we could test and refine components incrementally, rather than rushing at the end. This approach minimized bugs and improved maintainability.

### **TEAMWORK AND COMMUNICATION**

1. **Role Distribution**: Dividing responsibilities among team members based on strengths (e.g., UI, backend, database) allowed us to work efficiently in parallel and fostered a collaborative mindset.
2. **Conflict Resolution**: Through frequent meetings and open communication, we addressed development conflicts constructively and found optimal solutions collectively.
3. **Peer Learning**: Sharing knowledge and helping each other debug code or understand database queries strengthened both our team bond and individual skill sets.

### **REAL WORLD APPLICATION INSIGHT**

Finally, building TraceIt gave us exposure to solving real-life problems through software. Understanding user needs for a lost and found platform and translating those needs into a functional, reliable application gave us firsthand experience in user-centered design and full-stack development.

* 1. **CHALLENGES AND SOLUTIONS**

During the development of the TraceIt application, our team encountered several challenges across different stages of the project—from initial design and database modeling to implementation and testing. Each hurdle provided a valuable learning experience and compelled us to develop practical solutions to keep the project on track.

### **DATABASE CONNECTIVITY (MySQL Integration)**

1. **Challenge:**  
   Kotlin/Jetpack Compose does not support direct MySQL connectivity, and most tutorials default to Firebase. Establishing secure and efficient communication between the Android frontend and the MySQL backend was a technical challenge.
2. **Solution:**  
   We implemented a PHP-based REST API to act as a middle layer between the Android app and the MySQL database. This approach ensured structured JSON communication and allowed us to abstract complex SQL operations behind HTTP requests, improving maintainability and security.

### **DESIGNING OF RELATIONAL SCHEMA**

1. **Challenge:**  
   Modeling real-world concepts like item verification, claims, matching, and user roles into normalized relational tables without introducing anomalies was complex.
2. **Solution:**  
   We used ER diagrams initially to visualize the data structure and gradually transformed them into a 3NF relational schema. We made use of bridge tables and carefully assigned foreign keys to represent relationships (especially one-to-many and many-to-one) precisely.

### **HANDLING ASYNCHRONOUS OPERATIONS**

1. **Challenge:**  
   Fetching data over the network and updating the UI asynchronously without blocking the main thread was difficult, especially for new Kotlin developers.
2. **Solution:**  
   We utilized **Kotlin coroutines** for all network operations. By using suspend functions and structured concurrency (e.g., viewModelScope.launch), we were able to maintain app responsiveness while managing background data tasks effectively.

### **UI STATE MANAGEMENT**

1. **Challenge:**  
   Managing UI state reactively in Jetpack Compose, especially when data changed dynamically (e.g., matched items, notifications), proved to be tricky.
2. **Solution:**  
   We adopted **Mutable State** backed by remember or view Model to hold and control UI changes. We also used data binding techniques and composable re-composition to ensure the UI updated consistently with database changes.

### **COORDINATION BETWEEN MULTIPLE CONTRIBUTORS**

1. **Challenge:**  
   With multiple team members working on different parts of the system (e.g., database, API, frontend), maintaining consistency and avoiding merge conflicts was sometimes difficult.
2. **Solution:**  
   We maintained a shared Trello board and GitHub repository with assigned tasks and naming conventions. Regular sync-ups and pair programming sessions helped in resolving miscommunications early and maintaining consistency across modules.

### **LIMITED TESTING TIME FOR END TO END FLOW**

1. **Challenge:**  
   Due to tight deadlines, testing the complete flow from item reporting to return was challenging within the available time.
2. **Solution:**  
   We created mock data and scripts for rapid testing. Unit tests for API endpoints, along with manual walkthroughs of each major flow (reporting, matching, claiming, returning), helped us identify bugs quickly and ensure reliability.

Each challenge not only tested our technical capabilities but also strengthened our problem-solving and collaboration skills. Overcoming them contributed significantly to the robustness and success of the TraceIt project.

## FUTURE WORK AND IMPROVEMENTS

While the current version of TraceIt successfully delivers the core functionalities of a Lost and Found Management System, there are several potential areas for future enhancements to expand its usability, scalability, and user experience. The following are key improvements and features we envision for future iterations of the application:

### **ROLE BASED ACCESS CONTROL (RBAC)**

1. **Improvement:** Introduce more granular user roles such as moderators, super-admins, and verified users.
2. **Benefit:** This would enable more refined control over actions like item verification, user banning, or content moderation, improving overall system governance.

### **REAL TIME NOTIFICATIONS**

1. **Improvement:** Replace periodic refreshes with WebSocket-based real-time notification updates.
2. **Benefit:** Users and admins can be instantly notified of claims, matches, or feedback without needing to refresh manually, leading to a more responsive interface.

### **MAP-BASED ITEM LOCATION**

1. **Improvement:** Integrate location services (e.g., Google Maps API) for geotagging items instead of text-based locations.
2. **Benefit:** Users can visually drop pins on maps, making the item location more accurate and intuitive.

### **QR CODE AND RFID SUPPORT**

1. **Improvement:** Generate QR codes for registered items or support RFID tag integration for physical objects.
2. **Benefit:** Increases the reliability of identifying and matching physical items with digital records.

### **IMAGE SIMILARITY MATCHING USING AI**

1. **Improvement:** Implement machine learning algorithms to match items based on images using models like ResNet or MobileNet.
2. **Benefit:** Automates the matching process and increases accuracy, especially when textual descriptions are limited or vague.

### **MULTILINGUAL SUPPORT**

1. **Improvement:** Add localization support for Urdu and regional Pakistani languages.
2. **Benefit:** Makes the application more inclusive and accessible for a broader audience across Pakistan.

### **OFFLINE DATA ENTRY AND SYNCRONIZATION**

1. **Improvement:** Enable users to register lost/found items offline and sync the data once internet connectivity is restored.
2. **Benefit:** Ensures usability in remote areas or during low network coverage, improving reliability.

These proposed enhancements would not only refine the user experience but also increase the scalability and relevance of TraceIt in real-world deployments, particularly within academic institutions, public transportation, or large public venues.

* 1. **FINAL THOUGHTS**

Working on the TraceIt: Lost & Found Management System has been an incredibly rewarding experience that pushed us to apply and integrate multiple domains of knowledge—from database design and backend development to mobile UI/UX and system integration with MySQL. This project not only strengthened our technical expertise in technologies like Kotlin, Jetpack Compose, and PHP API interaction with MySQL, but also deepened our understanding of how real-world problems can be tackled through smart and accessible digital solutions.

Throughout this journey, we observed how each layer of the system—from relational schema and data flow to interface responsiveness—contributes to the robustness and usability of the final product. It was particularly fulfilling to see how a conceptual idea evolved into a working application capable of serving a meaningful purpose, such as helping users reconnect with their lost possessions.

We also gained a stronger appreciation for team coordination, clear documentation, and iterative development—values that are critical in any professional software development environment. This milestone-driven approach helped us break down the project into manageable components, enabling better tracking of progress and delivery of quality output.

Finally, we would like to express our gratitude to our mentor and instructor for their guidance throughout this project. Their feedback and encouragement helped us stay aligned with our learning goals while challenging us to build something functional and practical.

With this project, we close this phase not only with a complete and functional application but also with a sense of accomplishment, knowing that we’ve created something we can confidently demonstrate and potentially scale in the future.

# CHAPTER 7: REFERENCES

|  |  |
| --- | --- |
| [1] | A. K. H. F. &. S. S. Silberschatz, Database System Concepts, McGraw-Hill, 2019. |
| [2] | Firebase, "Firebase Authentication Docs," 2024. [Online]. Available: https://firebase.google.com/docs/auth. [Accessed 2025]. |
| [3] | R. B. M. S. K. Usman Ghani, "Database Design Document (V 1.0)," 2025. |

## APPENDIX: AI PROMPTS

Our group utilized AI tools (specifically, **ChatGPT by OpenAI**) responsibly and ethically throughout the development of **Milestone 4** of our project **TraceIt.** The AI was used primarily for guidance, structural analysis, and refinement of technical documentation. All final decisions, coding, integration, and implementation were executed by the team independently.

The tool supported us in enhancing clarity, accuracy, and professionalism across various components of our GUI-based application and database report. Below is a list of prompts we used to gain technical insights and ensure a polished submission:

1. "Can you analyze the GUI report sections based on our Kotlin + Jetpack Compose client integrated with MySQL via PHP APIs to identify any ambiguities or areas for clarification?"
2. "Suggest improvements for the Database Connectivity section when integrating Kotlin with MySQL using PHP APIs."
3. "Suggest professional and academically appropriate alternatives to simple headings like 'User Panel' and 'Admin Panel' for better documentation."
4. "Provide guidance on how to describe screenshots of our application's interfaces in a concise and explanatory manner."
5. "Help identify key takeaways and project management insights for writing Lessons Learned, Challenges & Solutions, and Future Work sections."