Database Design and Implementation

Role Playing Game Database

Mandatory Assignment Final

Databases for Developers

Sofie Amalie Thorlund

Soth0003

Viktor Bach Mekis

Viba0001

10.12.2025

Github Repo: https://github.com/SofieAmalie44/DB-project

Table of Contents

1. Introduction:

This report tells the story of how we have designed and implemented a relational database system, to support the core mechanisms of a role-playing game. The database will manage the users, their characters, and all related gameplay elements such as inventories, items, quests, skills, battles and non-player characters (NPC's). Throughout, each movement will be explained logically, what alternatives we might have considered and how it connects to the requirements.

The purpose of the project is to demonstrate the ability to design, normalize and implement a relational database system with a capability of supporting a dynamic game world. The implementation was developed in MySQL Workbench, which was used to model the ERD and generate SQL scripts for the creation of the database as well as testing. The project focuses on creating a flexible and normalized structure that can be scaled and integrated into a potential game server or application layer. The database will then be migrated to a document database (MongoDB) and a graph database (Neo4j).

1.2 Brief explanation of what the system must ensure:

Player management is prioritized so the player accounts is supported with secure login information.

The players should be able to manage their characters, so that each player can create and manage multiple characters, each with individual stats, like HP, mana, level, guild and money.

Inventory and items, makes it possible for characters to own multiple items, tracked with quantities and the attributes of the items, such as rarity and value.

Characters should take part in quests, trach their status and receive XP or money when completed.

Different levels, mana cost and damage should allow the character to learn and use multiple skills.

NPC's should provide quests, which will then be accessible for the characters.

Characters can engage in battle, with the results tracked, like enemy faced, outcome and reward gained.

1.3 The database will be implemented in MySQL, after a step-by-step design approach:

1. First the main entities and relationships was identified in a Conceptual ERD.
2. Then the foreign keys, primary keys and the relationship cardinalities was defined in a Logical ERD.
3. The Physical ERD was made by implementing the database schema with appropriate data types, constraints, indexes and referential integrity.
4. Lastly, the database was populated with sample data and we implemented stored procedures, triggers, views and events to emulate real game functionality.

2. Relational Database

2.1 Intro to Relational Databases

A relational database model is particularly suited for this project because the game world consists of multiple interdependent entities, like users, items, quests and NPCs, that require strong referential integrity and good structured relationships. The relational model makes it possible to maintain consistent data across these entities, also while enabling complex joins and queries needed for gameplay functions.

2.2 Database design

The process of designing the database, began by stating the business requirements. The requirements tell something about what the system must ensure, like "The players

should be able to manage their characters", which becomes important to better understand the systems, and thereby a better solution in the end.

The ERD is a visual representation of how items in a database relate to each other. By defining the entities, their attributes and showing the relationship between them, an ERD can illustrate the logical structure of the database. The ERD models can vary based on the level of details (IBM, What is an ERD) visualized, therefore the Conceptual, Logical and Physical ERD models is used to show the process of ERD modelling at various levels[1].

2.2.1 Entity/Relationship Model

The Entity-Relationship Diagram (ERD) is a key tool used to visualize the structure of the database. It allows us to clearly define the entities, their attributes and the relationships between them. In this section, we present the three levels of ERD modeling, which is Conceptual, Logical and Physical, to show how the design evolves from abstract ideas to an implementable schema.

Conceptual ERD

The conceptual ERD is the most abstract and has least details. As seen in Figure 2, the diagram contains entities, like Character, Player, Battle, NPC, Item, etc., and relationships, but it does not offer any details on specific database columns or cardinality. It provides us with a general, but high-level view of the database design.

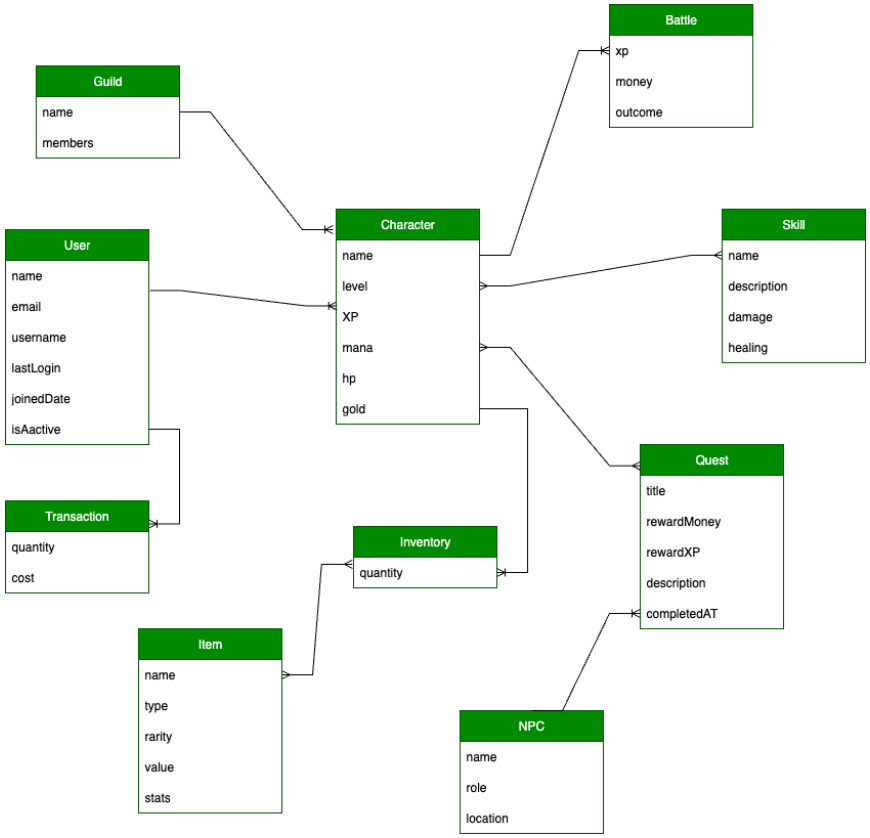


Figure 2: The conceptual ERD showing the initial drawing of the MySQL database.

Logical ERD

When making the Logical ERD, more details is added to the Conceptual model, with the purpose of defining additional entities that are operational and transactional, like Character_Quest, Character_Skill, and Inventory. The additional entities can be seen in Figure 3, as the entities separating the entities from the previous figure.

The inclusion of associative entities such as character _quest and character_skill was necessary to represent many-to-many relationships in the database. For example, a character can have multiple quests and a quest can be assigned to multiple characters. These linking tables ensure that relationships are properly normalized and scalable. This structure introduces a trade-off between efficiency and normalization, as the use of linking tables improves data consistency but can make joins more complex and slightly slower on larger datasets.

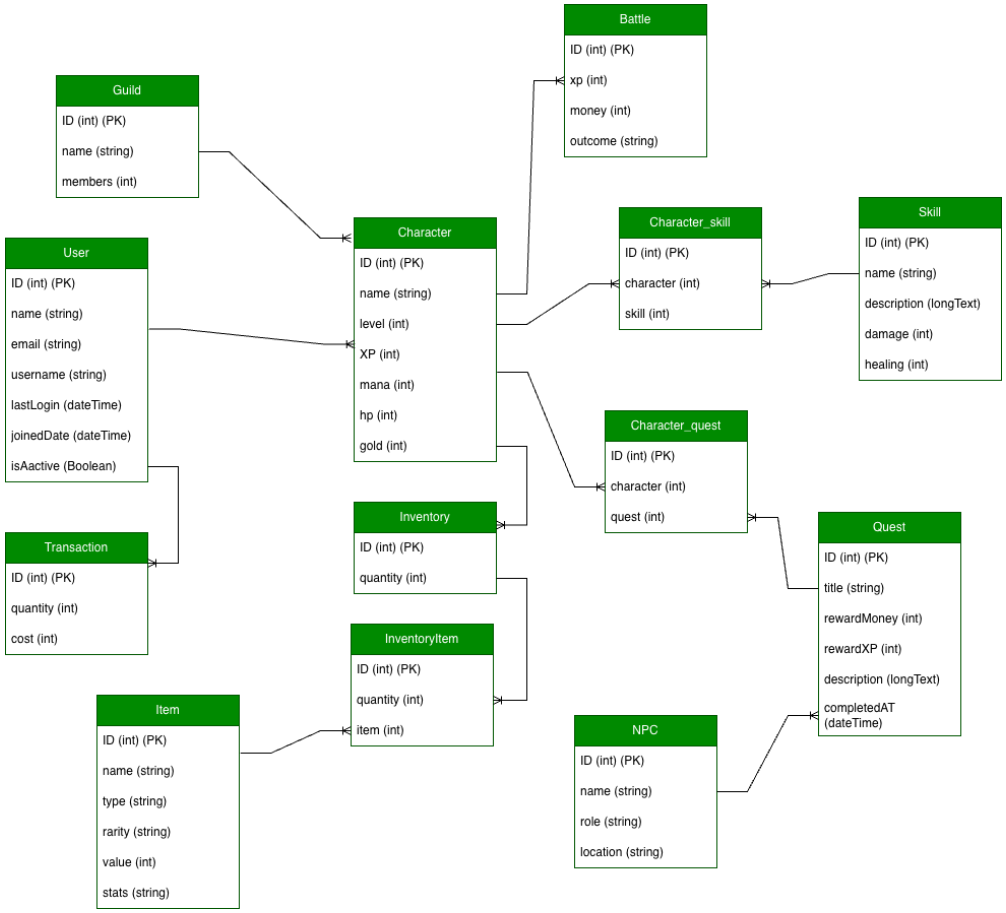


Figure 3: Det logical ERD showing an overview of the MySQL database, with many-to-many representation.

Physical ERD

The Physical ERD serves as the actual design or blueprint of the database. As seen in Figure 4, we added more technical details to the model, like defining cardinality and showing primary and foreign keys of the entities, instead of just their simple semantic names. To represent the columns of the real database, we listed the attributes related to their entities.

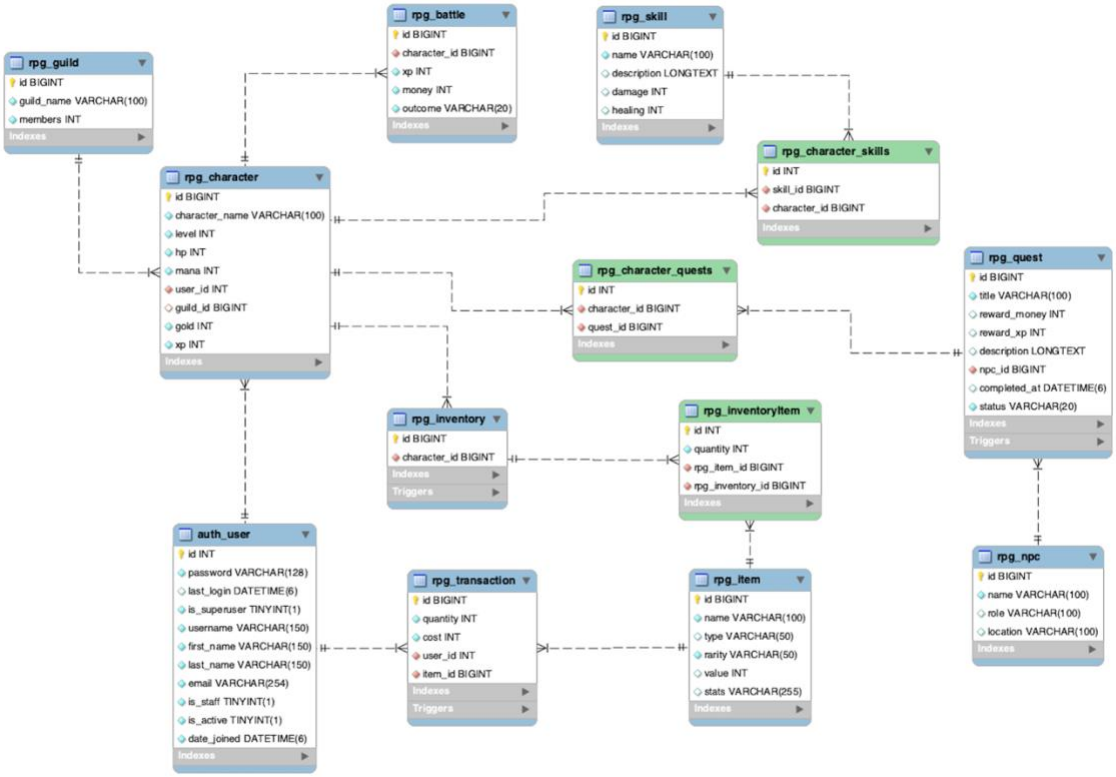


Figure 4: The physical ERD here provides an overview of the structure of the MySQL database, which is implemented in our system.

2.2.3 Normalization Process

Normalization is a process we choose to spend some extra time on, to ensure that the database is balanced and simple to use, manage and expand. During the design process, the data was decomposed into multible related tables so that insertions, updates and deletions could be performed safely without unintended side effects. This follows the principles of First Normal Form (1NF) to Third Normal Form (3NF), which together help maintain data integrity and reduce duplication (Semeraro, 1 Jun. 2023).

First Normal Form, makes sure that all tables in the RPG schema were structured so that each column contains atomic values only. Examples from our project includes: In the rpg_character table, attributes such as level, xp, hp and mana each store a single integer value, and multi-valued attributes such as skills and quests were moved into

join tables, like rpg_character_skill and rpg_character_quests, ensuring that repeating groups exist inside the character table. This all the 1NF requirement that every row-column intersection holds exactly one value.

Second Normal Form applies to tables with composite primary keys, which in this project are the join tables, rpg_character_skill(character_id, skill_id), rpg_character_quests(character_id, quest_id) and rpg_inventoryItem(inventory_id, item_id). In these tables the attribute “quantity” in rpg_inventoryItem depends on the full composite key, not just one part. There is also no partial dependencies, for example, “quantity” does not depend solely on item_id or solely on inventory_id. This means that 2NF is satisfied, because each non-key field depends on the entire primary key.

A table violates 3NF if a non-key attribute depends on another non-key attribute. In our project, it is clear that there is no transitive dependencies. For example, in rpg_user, fields susch as email, first_name and is_staff all depend only on the primary key (id). None of them depend on each other. In rpg_character, attributes such as character_name, level and guild_id depend only on the character’s id. We avoided transitive dependencies by placing related descriptive attributes in their own tables, like item has rarity, skill has damage and quest has rewards.

This all ensures that updates connot accidentally create inconsistent data, when for example changing an NPC’s role updates only the rpg_npc table, not unrelated tables.

Originally, it might seem convenient to store Users, Characters, Skills, Quests and Items in a single large table. However, this would lead to immediate problems, if a user with multiple characters would cause the user’s email and profile data to be duplicated repeatedly. It could cause adding new skills or items for each character, which would create massive structural redundancy. If updating one attribute, like renaming a skill, would require modifying many rows manually.

Therefore, by normalizing to 3NF, we remove redundancy, while preserving referencial integrity, queries become faster and more maintainable and the system scales cleanly as more characters, quests and items are added. This design ensures that the database supports the game mechanics reliabke while maintaining strong data integrity.

2.3 Physical Data Model

The physical data model represents the actual implementation I MySQL. While the Conceptual and Logical models focus on structure and relationships between entities, the Physical models purpose is to define the technical details that makes our system functional. It specifies the data types, keys, indexes, and referential integrity constraints that ensures the data consistency and efficient in a live database.

As we show in Figure 4, each entry is represented as a table with columns (attributes), data types and defined relationships through primary and foreign keys.

The example found in Appendix G, illustrates the technical structure of the physical data model, where datatypes, constraints and keys are explicitly defined to enforce data integrity

2.3.1 Data Types

Each of the attributes in the physical model is assigned a data type appropriate for the kind of information it stores.

For example, the ID's refer to the identifiers and uses INT with AUTO_INCREMENT, to ensure the uniqueness of the various ID's used in the different tables. We use VARCHAR(45) for name and text values, like in character_name, guild_name and item_name, this provides for flexible string storage. The different numeric stats, like level, hp, mana and xp uses INT. For the monetary values we use DECIMAL for maximal precision, this can be seen in cost and reward_money.

Because of our selection of data types we ensure data accuracy, memory efficiency, and optimal query performance.

2.3.2 Primary and foreign keys

The primary keys uniquely identify each record in a table, while foreign keys helps establish relationships between tables and enforce referential integrity. Referential integrity ensures that relationships between data remains valid, for example, a character cannot reference a guild that does not exist.

How it works in in our tables:

ruser.id → Primary key of user

character.user_id → Foreign key referencing user(id)

character_quest.character_id → Foreign key referencing character(id)

inventoryItem.inventory_id → Foreign key referencing inventory(id)

The primary keys prevent duplication, while foreign keys maintain logical consistency between related tables.

In the rpg system, each relationship has been designed with explicit ON DELETE rules to ensure predictable behavior when parent records are removed. These rules prevelt orphaned data, maintain consistency and support the game logic. If a guild is deleted in the character table to guild_id (FK) is using the ON DELETE rule (SET NULL), which means the character simply leaves the guild if the guild is deleted. While in the battle table the character_id (FK) is using the ON DELETE rule (CASCADE), meaning that battle logs for a character are deleted if the character is removed.

These ON DELETE rules were chosen, because CASCADE can be used where the child record cannot logically exist without the parent. SET NULL can be used when deletion should detach rather than destroy. RESTRICT can be used when we want to prevent deleting data that is still in use, for example, an item cannot be delete if it appears in a transaction record. These rules together ensure that the database behaves predictably and prevents accidental data loss.

2.3.3 Indexes

Indexes can be created to improve the speed of queries and data retrieval. In this project, indexes is defined for frequently searched attributes, such as: username in the user table, which speeds up the login queries. The character table, character_name is made to find characters faster. Foreign key columns like user_id, guild_id and inventory_id is very useful for join efficiency.

By making proper indexing the performance of joins and lookups across related entities will be optimized.

2.3.4 Constraints and Referential Integrity

Constraints are used to maintain the accuracy and reliability of data across the system. The most important types include:

NOT NULL refers to mandatory fields, like username or email.

UNIQUE is a constraint that prevents duplicate usernames or guild names.

CHECK makes sure to validate acceptable values, like level must be equal to or greater than 1.

FOREIGN KEY constraints enforce referential integrity between related tables.

2.4. Stored objects – stored procedures / functions, views, triggers, events

Stored objects extend the database's functionality by adding logic that would otherwise be handled in the game's application layer. They provide automation, improve performance and ensure consistency of recurring game actions directly within the databse.

2.4.1 Stored procedures

Stored procedures are queries or SQL code, that can be saved in MySQL, such that it can be used repeatedly, without typing It out every time. In a stored procedure we can also pass parameters, meaning the procedure will act based on that parameters value. In an rpg, there are of places where stored procedures or functions could be useful. We chose to create two of these procedures, one for leveling up a character and one for adding items to a character's inventory. These scenarios are probably the ones most likely to require repeated use, and therefore are suited to be stored as functions, that can be called rather than coded all the way through. The level up procedure looks like this: The item to inventory like this:

While the same logic could technically be implemented in the backend, there are important reasons for placing these operations inside the database itself. Stored procedures centralize business rules directly in MySQL, ensuring consistant behavior no matter where the operation is triggered from. They also reduce network traffic, because multiple SQL operations are executed internally in MySQL, and they guarantee atomicity, either all updates succeed or non do. This is especially relevalt in the RPG system, where many characters may level up or update inventory items at the same time.

Since these procedures are multiple lines of code, but are meant to be ran as one line, there must be a delimiter at the start and end to declare a new way of "ending" a line of SQL code. Normally this delimiter would be a semi colon, however we need semicolons in the procedure itself, so declaring a new delimiter outside the procedure is necessary. This means MySQL can differentiate between when the procedure ends, and when a line of code inside the procedure ends.

In the level up character, we simply update and set new values for the specified character id. In The item adding procedure we define two input parameters: p_character_id and p_item_id. Then we declare a local variable to store the inventory id: v_inventory_id. Now we can select the inventory_id from the give (p_character_id), and store it in v_inventory_id. Then we can insert this item into the join table inventory_has_item.

2.4.2 Views

Views in MySQL make it possible to save a “snapshot” of a query like a select statement, and that can be viewed by third party users. Views are especially useful for security purposes, as they hide the internal details like table structure and attributes, and only show the user what is defined in the view. This makes it easy to hide sensitive data. Instead of writing complex queries with join statements we can save such queries in a view for easy access and display. In our view we show the player overview by selecting the username, character_name, guild_name and level from the user, character and guild tables. Having this overview as a view, makes it convenient to show this data, as we don’t need to write the joins in a query.

2.4.3 Triggers

Triggers are special stored programs, that can be triggered automatically on a specified event in the database. This is very useful for functions that always are expected to occur after certain events. In our database we use a trigger to log new transactions when they are inserted into the transaction table. This makes logging the transactions much simpler than if we had to code it with every new transaction, or in an application.

2.4.4 Events

Events are similar in concept to triggers, but instead of occurring on an event in the database, events can be set to run on a time-based schedule. In our app we simply delete “daily quests” every day, such that players can accept these quests daily, rather than having to remove them from the quest table by deleting/completing them.

2.5 Transactions

In relational SQL, a transaction refers to a unit of work involving multiple write operations that must either:

- All succeed together (COMMIT), or
- All be undone together (ROLLBACK)

This is referred to as ACID consistency:

A = Atomicity	All changes apply or none apply
C = Consistency	Data must remain valid according to schema rules
I = Isolation	Transactions do not interfere mid-execution
D = Durability	Once committed, changes persist even after crashes

2.6 Audit and Trigger-Based Integrity Controls

Auditing in database systems refers to the systematic recording and monitoring of state changes as they occur across critical tables.

It provides traceability, accountability, and security visibility into who changed what, when, and under which circumstances.

In systems that contain player progression data, in-game transactions, and potentially sensitive economic balances (such as gold or item ownership), auditing becomes essential to detect unauthorized behaviour and fraud and preserving the historical evidence for debugging and consistency on an internal level. When auditing is effective, it creates a “chain” of change events that can be reconstructed and validated upon detection of anomalies.

The RPG database implements triggers not only to automate internal actions but to act as an embedded enforcement and audit layer, ensuring that certain changes are either corrected, timestamped, or rejected before violating gameplay or database logic.

Each trigger participates in the broader auditing goal by ensuring correctness at the data boundary—the moment data attempts to enter an invalid state.

The trigger trg_inventory_prevent_negative functions as data integrity protection, where negative item quantities might indicate duplication exploits or rollback inconsistencies. By rejecting invalid states before they take effect, we prevent corruption of character inventories and economy values in the database. The trigger can be seen in Appendix H.

Another trigger, the trg_transaction_adjust_gold will automatically and immediately debit the player's balance, forming a strict transaction ledger after every purchase. This prevents gold duplication and negative purchase cost exploits where the user might be able purchase something without having enough funds. The trigger can be seen in Appendix I.

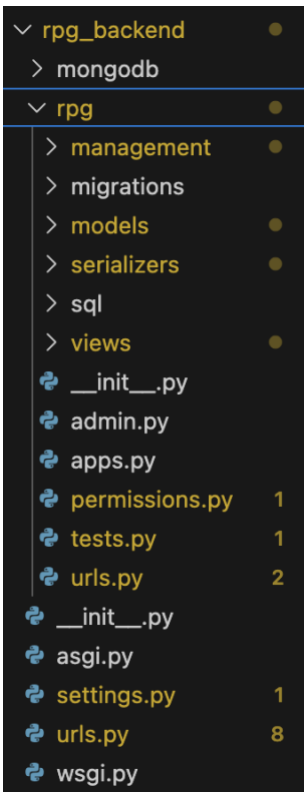
3 Application Layer: Django Implementation, Seeding, and API Testing

With the relational database implemented in MySQL, the next step was to create an application layer capable of interacting with the RPG data in a structured way. For this purpose, we selected the Django framework, which provides a clean architecture, a powerful ORM, and strong support for REST API development through Django REST Framework (DRF).

3.1 Django Project Structure

The Django backend serves as the primary interface between clients (e.g., the game engine or external tools) and the underlying SQL database. The project was structured into modular applications corresponding to major domain concepts such as Users, Characters, Items, Skills, Quests, and Inventory.

- Django's MTV (Model-Template-View) pattern ensures clear separation of concerns:
- Models map directly to the physical MySQL tables described earlier.
- Serializers convert model instances to JSON so they can be transmitted over REST
- Views / ViewSets handle the business logic and CRUD operations on characters, inventories, items, etc.
- URLs / Routers expose REST endpoints for external systems.



Admin configuration allows quick visualization and editing during development.

Django's ORM allowed us to maintain consistency with the relational schema while simplifying query logic. For example, one-to-many relationships such as Character to Quest or Inventory to Item are expressed naturally through Django's ForeignKey and ManyToManyField constructs.

3.2 Model Mapping Example

Appendix L shows the class Character in the models folder, this reflects our MySQL schema exactly and demonstrates that our Django model mirror the physical relational structure.

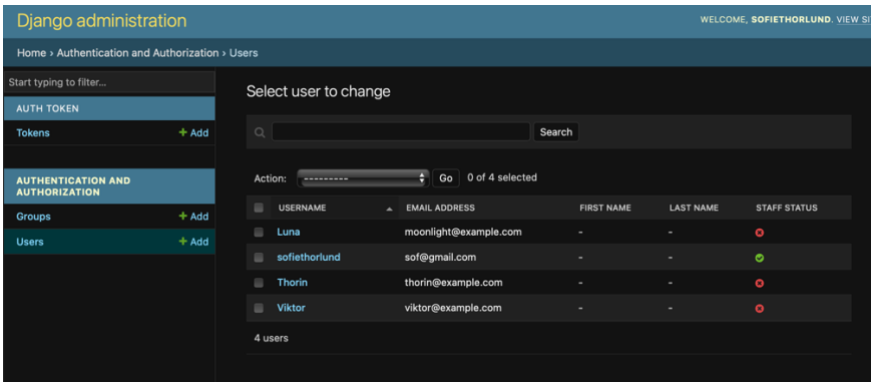
3.2 Database Seeding

To test the API and support gameplay logic, the database was seeded with sample RPG data.

Our seeding approach included:

- SQL insert scripts for items, characters, NPCs, quests and skills.
- Automatic population of ManyToMany relationships (like, character skills).

- Use of Django admin to manually add missing relations during testing.



The Django admin panel was used to verify seeded data and inspect the relational structure during development. The interface provides quick access to all major entities, including Users, Characters, Items, Inventory, Skills and Quests, allowing validation of database integrity after SQL import and during API testing.

3.3 REST API Development with Django REST Framework

Every major RPG entity corresponds to a dedicated REST endpoint. DRF's ModelViewSet allowed us to generate the full CRUD interface with minimal code.

See Appendix K for CharacterViewSet:

This table shows the exposure of all PRG entities as RESTful resources.

Entity	Endpoint Example	Method Support
Characters	/api/characters/	GET, POST
Inventory	/api/inventory/<id>/	GET, PUT
Items	/api/items/	GET

Entity	Endpoint Example	Method Support
Skills	/api/skills/	GET
Quests	/api/quests/	GET

DRF's ModelViewSets offered a concise way to generate CRUD routes while retaining the flexibility to define custom actions (e.g., *add skill to character*, *complete quest*, *buy item*).

Authentication was handled through Django's native user model, and permissions ensured that users could only access and update their own characters.

3.4 API Testing with Postman

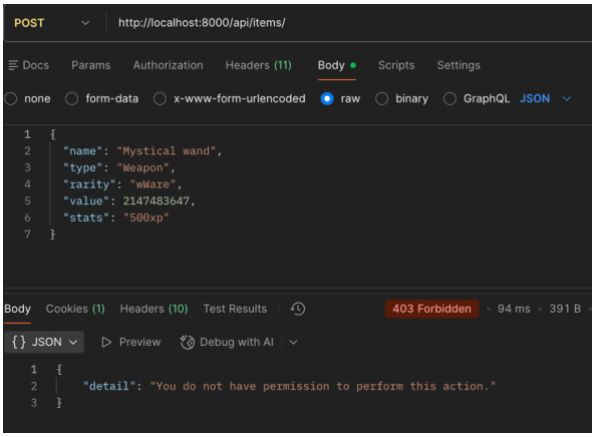
To validate functionality, we used Postman collections covering:

- Character creation and retrieval
- Inventory management
- Item CRUD operations
- Authentication (Token and JWT)
- Permissions (admin vs regular user)

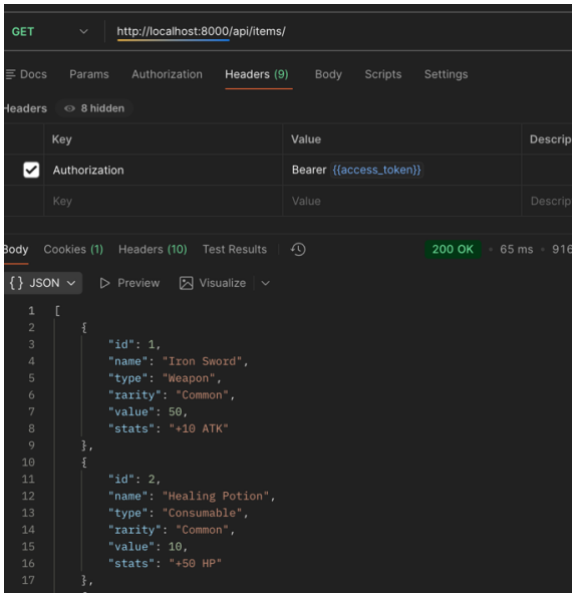
Tests we covered:

- Correct status codes
- Input validation
- Authentication/authorization logic
- Data consistency before and after API calls

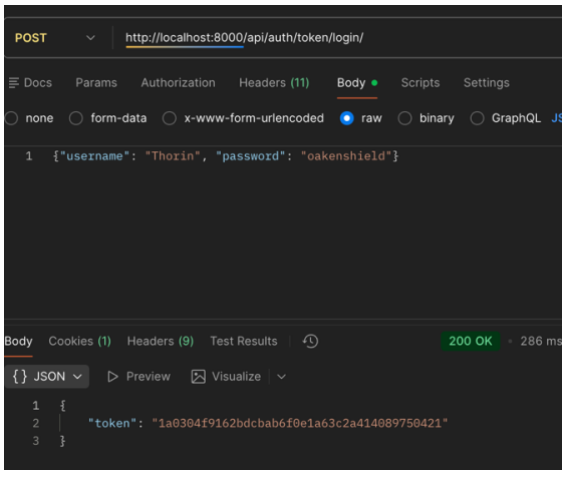
Example of creating a new item without authorization:



Example of getting all items with authorization:



Example of logging in and getting a token:



4 Document Database

This section explains how and why we extend our relational MSQL solution with a denormalized, document-oriented database using MongoDB.

4.1 Introduction to MongoDB

MongoDB is used as a second database technology in this project to demonstrate how a document-oriented storage model differs from a traditional relational system such as MySQL. Whereas MySQL enforces strict schema constraints, predefined relationships and join-based queries, MongoDB allows a flexible schema and stores data in JSON-like documents (MongoDB Documents Model). This makes MongoDB particularly well-suited for game-related domains, where objects often have nested structures, variable attributes and dynamic relationships.

While the SQL database focuses on normalization, referential integrity and transactional correctness, MongoDB enables a more flexible structure optimized for heavy reads and aggregated game-state retrieval (Embedding vs Referencing), like complete character sheets, inventories, skills and battle logs.

This hybrid approach reflects real world game backend architectures, where relational and document stores coexist to support different performance profiles (Schema Best Practices).

4.2 MongoDB Data Model Design

The MongoDB schema is based on the existing SQL model but redesigned to follow document-oriented principles. Each SQL table becomes a MongoDB collection, but the relationships are modeled differently.

4.2.1 MongoDB Data Model Design

MongoDB supports two fundamental approaches to representing relationships. One is Embedding, which is used when related data logically belongs inside the parent

document. The other is Referencing, which is used when related documents should remain independent or reused elsewhere (Embedded vs Referencing).

This project uses a hybrid design, following the recommended MongoDB schema-design patterns.

Embedded is used for Inventory items, each inventory document contains an array of item objects, seen in Appendix A. Embedding makes CRUD operations faster and avoids the need for tables (Schema Design – When to embed).

Referencing is used for Character to Guild, Character to User, Transaction to User and Item, and Quest to NPC. Only the ID is stores, keeping the structure clean and avoiding unnecessary duplication(Referencing Pattern).

4.2.2 MongoDB Diagram

The figure below illustrates the final document schema. It focuses on collections, their fields, and embedded document arrays.

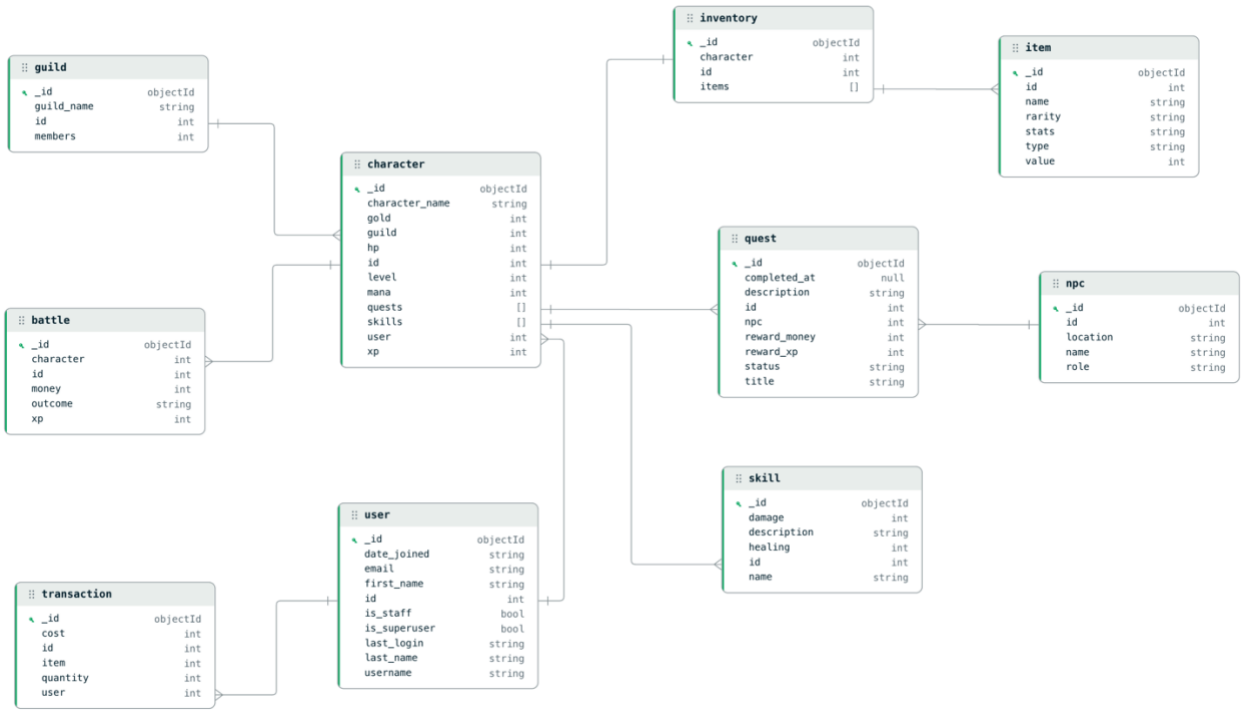


Figure 5: Overview of MongoDB collections and clear structure of each document. It is the document-database equivalent of an ERD (MongoDB Document).

Since document databases do not use foreign keys or normalized tables, a traditional ERD is replaced with a Document Structure Diagram. This diagram communicates collections and the shape of each document.

4.3 Migration Script (From SQL to MongoDB)

To migrate all SQL data into MongoDB, a custom Django management command (Django Management Commands) was implemented

Rpg_back/rpg/management/commands/migrate_to_mongo.py

The Script (Appendix E) automatically iterates through every model in the rpg application and converts relational rows into MongoDB documents (PyMongo CRUD Operations).

The script performs the tasks such as connecting to MongoDB using PyMongo, which is a Python distribution containing tools for working with MongoDB, it reads MySQL data via Django ORM, it transforms relational structures to nested MongoDB documents, it is handling ManyToMany relationships as embedded arrays, it creates the following collections: character, inventory, item, skill, battle, npc, transaction and user, and it handles JOIN tables, like inventory_item, character_quests, character_skills.

These are embedded directly into arrays, for example:

Normalized SQL version:

Table	Fields
character	id, name, hp, guild_id, level, mana, user_id, gold, xp
inventory	id, character_id
inventory_item	inventory_id, item_id, quantity
item	id, name, value, type, rarity, stats

MongoDB – denormalized

The transformation process is illustrated in Appendix A, here the SQL to MongoDB mapping is shown.

4.3.1 Conversion Rules

Rule 1 - ForeignKey to Scalar ID

Istead of storing foreign key objects, the script stores the related object's ID, for example, character.guild_id stores "guild": 3. This keeps documents lightweight and avoids circular references.

Rule 2 - Many-to-Many to Embedded Array of IDs

Many-to-Many becomes embedded arrays (MongoDB Embedding for Related Data). SQL join tables such as character_skills are removed, instead MongoDB stores them directly as lists, like "skills": [1, 2, 7]

Rule 3 - Reverse Relations Are Ignored

Django auto-creates reverse relationships that do not have a direct representation in MongoDB. These are skipped to avoid unnecessary nesting.

Rule 4 - Original SQL ID Preserved

Each MongoDB document contains an _id (MongoDB internal) and id (original SQL identifier). This ensures table references across collections.

4.3.2 Core Migration Logic

The core migration logic is placed inside the migrate_to_mongo.py file, specifically the code snippet in Appendix D, illustrates how each model instance is transformed. The fully implemented migration command is included in Appendix E.

4.4 MongoDB Compass Validation

After running the migration script, the database was inspected visually using MongoDB Compass (MongoDB Compass Docs), as shown in Appendix B.

We checked if all collection were created, if each model contains the expected number of documents, if embedded arrays matched existing SQL many-to-many relationships and if the document structure matched the denormalized schema.

When looking at the character document from Compass in Appendix C. This confirms that the transformation from relational to document format preserved semantic correctness.

4.5 CRUD Operations and API Integration

All MongoDB collections are exposed through dedicated REST API endpoints implemented using Django REST Framework combined with PyMongo (DFR APIView Docs)(PyMongo CRUD). Each collection supports GET all, GET by ID, POST (create), PUT (update) and DELETE. This ensures MongoDB is not only a storage layer but also fully accessible via API.

4.5.1 Example of Inventory CRUD

The inventory API supports advanced operations, including, add and item, update item quantity, remove item and filter by character or item ID.

This design demonstrates MongoDB's strength when handling nested array updates, which would require multiple joins in SQL.

4.6 Query Filtering

Some endpoints provide filtering using MongoDB queries.

For example:

```
GET /api/mongodb/filter/characters?min_level=10
```

which translates to

`query["level"] = {"$gte": 10}` MongoDB's flexible query engine allows efficient filtering (MongoDB Query Operations) without schema constraints.

The full code snippet of the implementation of CRUD operations for filtering can be seen in Appendix F.

4.7 Why Authentication is disables for MongoDB Endpoints

Authentication is intentionally not enabled for MongoDB endpoints. MongoDB is used here strictly as an alternative data model and CRUD demonstration.

Because MongoDB version is a read-optimized mirror, meaning it is not part of the authoritative game logic, it is intended for testing, analytics and demonstration. When doing Postman and Swagger, it also simplifies testing.

This separation mimics real production systems where MySQL handles core gameplay state and MongoDB is the caching layer or optimized API view.

5 Neo4j Graph Implementation

As for our graph database solution, we have chosen Neo4j. This graph database management system stores data as nodes which have relationships connecting with each other in a large graph. There are also properties which are on both nodes and relationships. This would enable the system to natively represent complex gameplay networks—alliances, quest dependencies, faction hierarchies, social interactions, and battle histories—not as rows in join tables, but as first class relational structures that can be traversed in constant time regardless of graph depth or direction.

Example Neo4j fragment

```
CREATE (c:Character {name:'Aelric'})

CREATE (g:Guild {name:'Knights of Dawn'})

CREATE (c)-[:MEMBER_OF]->(g)
```

Where MySQL would require three tables (character, guild, character_has_guild), Neo4j represents them as a direct semantic link.

In our implementation we follow this structure, creating character nodes with properties like this:

```
(:Character {
  sql_id: 1,
  character_name: "Aelric",
  level: 10,
  hp: 200,
  mana: 100,
  xp: 500,
  gold: 1000
})
```

5.1 Relationships in Neo4j

Relationships in graph databases are directed and typed, meaning the relationship type itself is what carries meaning. So a character belonging to a guild would be

(c:Character)-[:GUILD]->(g:Guild).

In MySQL this would require three tables to model so we see an immediate advantage using graph databases because as relationship complexity increases, SQL performance degrades.

5.2 Why Use a Graph Database for an RPG System

As mentioned Neo4j provides constant-time traversal regardless of graph depth, whereas SQL performance degrades as JOIN complexity increases. In Neo4j, each node maintains direct pointers to its relationships, making traversals extremely efficient.

SQL vs. Graph Queries

Objective	SQL Query	Neo4j Query
Find all guild members and their allies	Complex multi-JOIN with self-joins	MATCH (c)-[:MEMBER_OF]->(g)-[:ALLY_OF]->(g2)<-[:MEMBER_OF]-(c2) RETURN c, g2, c2
Build a full quest dependency chain	Recursive CTE (not supported in all SQL dialects)	MATCH p=(q:Quest)-[:LEADS_TO*]->(next) RETURN p
Determine who has fought the same boss	Multiple joins through battles and logs	MATCH (c)-[:FOUGHT]->(b)<-[:FOUGHT]-(c2) RETURN c, c2

5.3 Data Migration Architecture and CRUD REST API for Graph db

We built a Django management command that automatically migrates data from MySQL to Neo4j while preserving all relationships. The migrator reads Django ORM models and creates corresponding nodes and relationships. We can test these nodes and connections using the Neo4j browser that we host through Docker (See appendix J).

We also created a complete CRUD application for Neo4j following the same architectural patterns as our MongoDB implementation. Each entity (Character, Item, Skill, etc.) has a dedicated view class that handles CRUD operations using Cypher queries.

Conclusion

This project demonstrates the design, implementation, and extension of a complete database architecture capable of supporting a dynamic role-playing game environment. Beginning with a fully normalized relational schema in MySQL, we established a foundation for core gameplay mechanics, including user management, character progression, inventories, quests, items, skills, NPC interactions and battle states. Through the structured ERD approach—Conceptual, Logical, and Physical—we ensured that each entity and relationship was introduced with clear purpose, minimal redundancy, and strong referential control. Stored procedures, triggers, views and events further extended this foundation by automating recurring gameplay functions, enforcing consistency, and preventing invalid in-game states.

As the system evolved, focus expanded beyond traditional relational modelling to include modern multi-model design principles. MongoDB was integrated as a document-oriented counterpart, allowing highly aggregated character profiles, nested inventories and logs to be retrieved in a single read operation without joins. The migration process illustrates how normalized SQL relationships transform into embedded arrays and flexible schemas suited for read-heavy gameplay inspection and analytics. This hybrid solution reflects real-world backend patterns where relational engines maintain authoritative state, while document stores provide efficient secondary representations for performance-sensitive queries, dashboards or testing endpoints.

The system was further evolved using Neo4j graph database. By migrating SQL entities into graph nodes and typed relationships, gameplay logic traditionally expressed through multi-layer JOINs becomes semantically direct and traversable at constant relational cost. This unlocks potential for narrative analysis, recommendation mechanics, player clustering and advanced game-world interaction modelling far beyond what relational structures can efficiently express.

In combination, MySQL, MongoDB and Neo4j form a cohesive multi-database architecture evaluated not as competing technologies, but as complementary layers addressing distinct operational needs: transactional correctness (SQL), flexible state retrieval (MongoDB), and deep relationship intelligence (Neo4j). Supported by Django,

REST framework endpoints, Postman and Swagger validation, the implementation demonstrates practical deployment of each model within a common API surface.

The outcome is a stable and scalable solution illustrating how hybrid storage strategies can enhance both gameplay logic and developer efficiency.

References

Relational Database:

1. What is an ERD – Ivan Belcic and Cole Stryker. Available at: https://www.ibm.com/think/topics/entity-relationship-diagram (Accessed on 5 Oct. 2025)
2. Understanding Database Normalization - Anto Semeraro (1 Jun. 2023). Available at: https://blog.dataengineerthings.org/understanding-database-normalization-from-basics-to-er-diagrams-488a53923cf4 (Accessed on 5 Oct. 2025)
3. MySQL Referential Integrity. Available at: https://dev.mysql.com/doc/refman/8.0/en/innodb-foreign-key-constraints.html (Accessed on 6 Oct. 2025)
4. MySQL Stored Procedures, Views, Triggers. Available at: https://dev.mysql.com/doc/refman/8.0/en/stored-programs-views.html (Accessed on 6 Oct. 2025)
5. DRF ModelViewSet Docs. Available at: https://www.django-rest-framework.org/api-guide/viewsets/ (Accessed on 11 Oct. 2025)

MongoDB Official Documentation:

6. *MongoDB Data Modelling Introduction*. Available at: https://www.mongodb.com/docs/manual/data-modeling/ (Accessed on 2 Dec. 2025)
7. *MongoDB Embedded vs Referenced Data*. Available at:

https://www.mongodb.com/docs/manual/data-modeling/#embedding-vs-referencing (Accessed on 2 Dec. 2025)

8. *MongoDB Embedding for Related Data*. Available at: https://www.mongodb.com/docs/manual/data-modeling/#embedding-data (Accessed on 2 Dec. 2025)
9. *MongoDB Schema Design Best Practices*. Available at: https://www.mongodb.com/developer/products/mongodb/schema-design-best-practices/ (Accessed on 2 Dec. 2025)
10. *MongoDB Documents and BSON Structure*. Available at: https://www.mongodb.com/docs/manual/core/document/ (Accessed on 2 Dec. 2025)
11. Referencing Pattern. Available at: https://www.mongodb.com/docs/manual/data-modeling/#references (Accessed on 2 Dec. 2025)
12. *MongoDB CRUD Operations*. Available at: https://www.mongodb.com/docs/manual/crud/ (Accessed on 2 Dec. 2025)
13. *MongoDB Aggregation Framework*. Available at: https://www.mongodb.com/docs/manual/aggregation/ (Accessed on 2 Dec. 2025)
14. *MongoDB Compass Documentiation*. Available at: https://www.mongodb.com/docs/compass/ (Accessed on 2 Dec. 2025)

PyMongo / Django + MongoDB References

15. *PyMongo Usage Guide*. Available at: https://pymongo.readthedocs.io/en/stable/
16. *Django Management Commands* (Official Docs). Available at: https://docs.djangoproject.com/en/stable/howto/custom-management-commands/
17. *Django REST Framework, APIView*. Available at: https://www.django-rest-framework.org/api-guide/views/
18. *Django REST Permissions*. Available at: https://www.django-rest-framework.org/api-guide/permissions/

General NoSQL / Schema Design Academic References

19. *MongoDB University Corse on Data Modeling*. Available at: https://learn.mongodb.com/

List of figures

Figure 2: Conceptual ERD

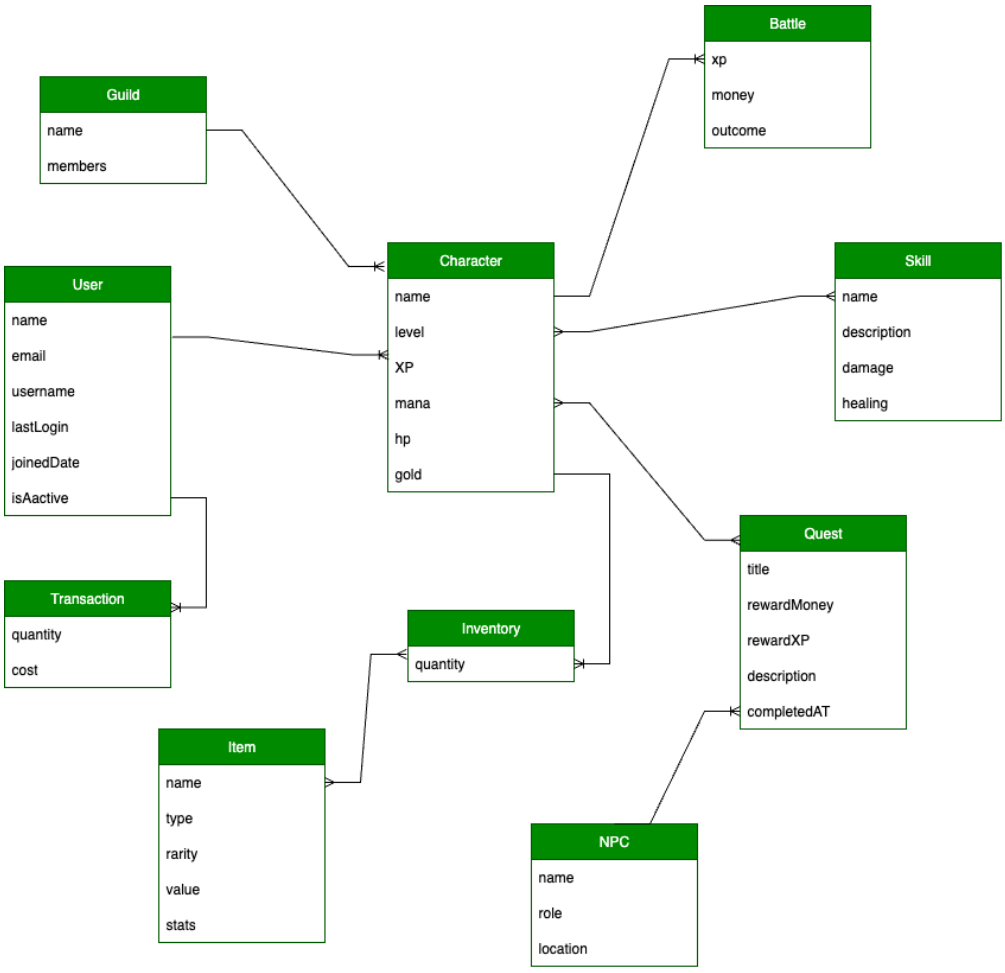


Figure 3: Logical ERD

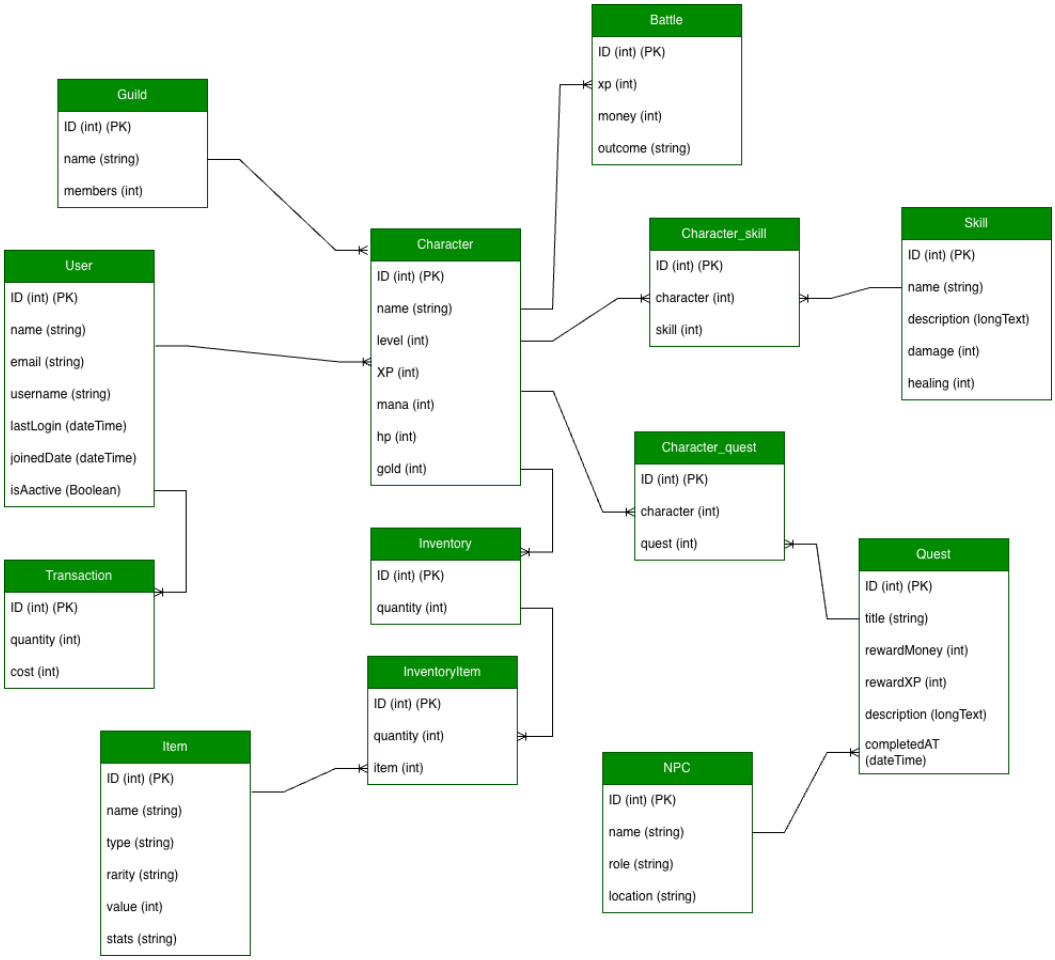
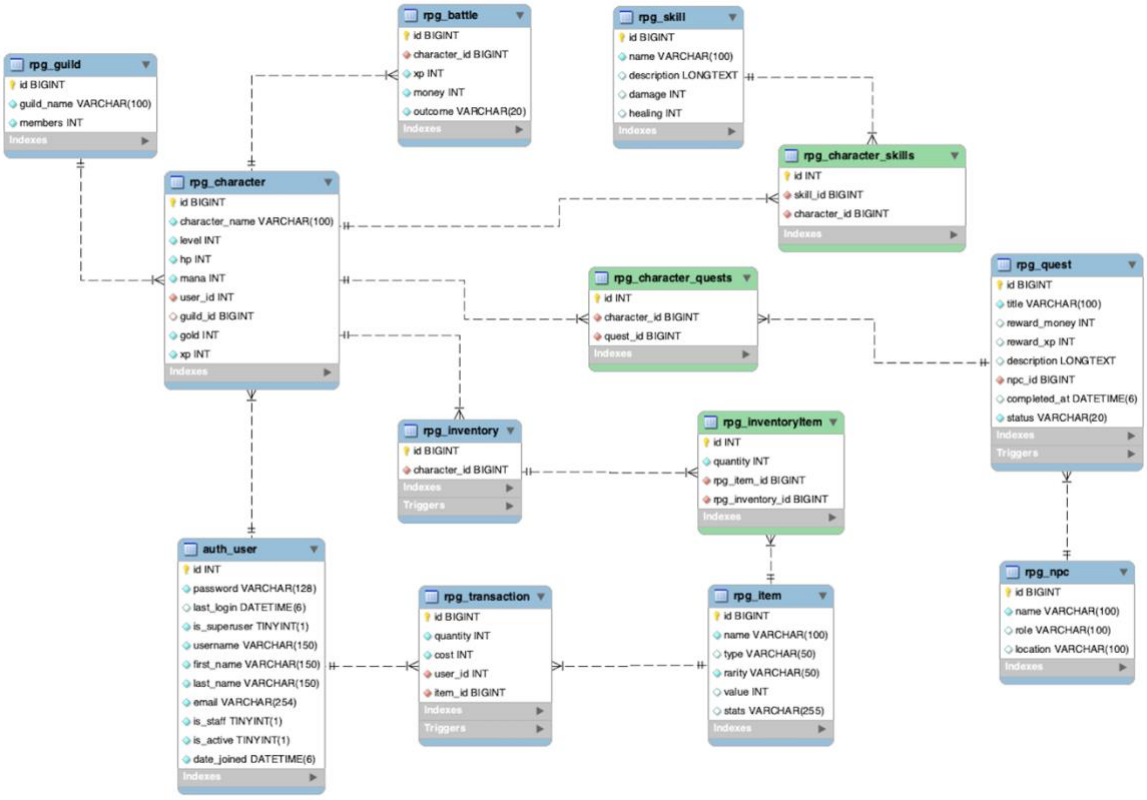


Figure 4: Physical ERD



List of Appendixes:

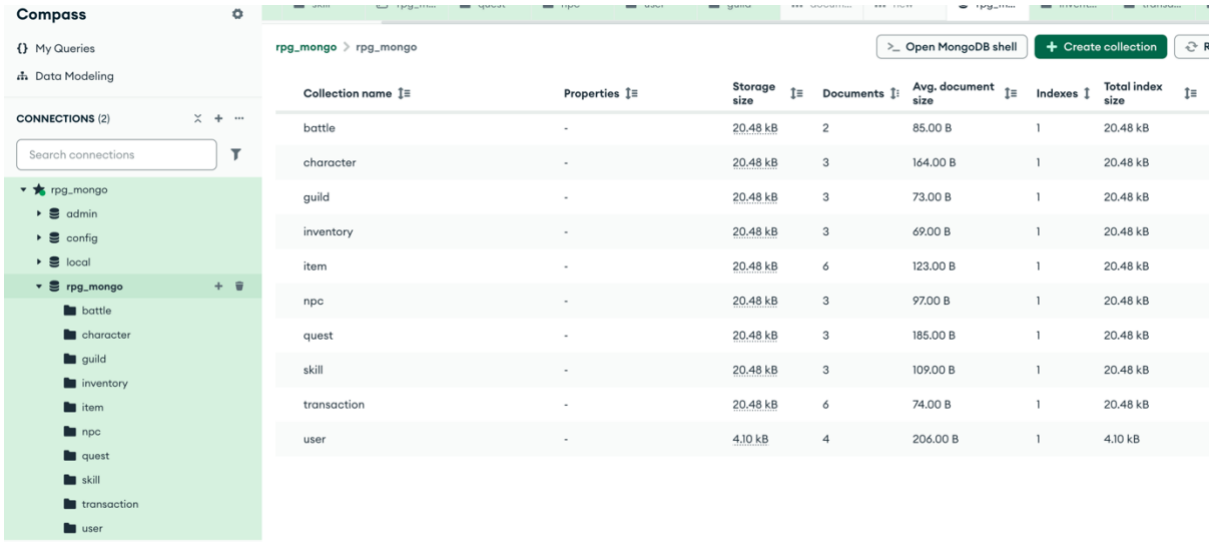
Appendix A:

MongoDB document showing SQL to MongoDB mapping of inventory.


```
{
    "_id": "6936d1be52f96087186edad6",
    "id": 7,
    "character": 3,
    "items": [
        {
            "item": 2,
            "quantity": 5
        },
        {
            "item": 3,
            "quantity": 1
        },
        {
            "item": 4,
            "quantity": 10
        }
    ]
}
```

Appendix B:

Screenshot of Compass connection window looks like.



Appendix C:

Screenshot of a single document (Character) expanded showing nested lists looks like.

```
_id: ObjectId('69342839dc4a044855a274a4')
id : 1
character_name : "Aelric"
user : 1
level : 2
hp : 200
mana : 100
xp : 150
gold : 60
guild : 1
▼ skills : Array (1)
    0: 1
▶ quests : Array (1)
```

Appendix D:

Code snippet of how each model instance is transformed in migrate_to_mongo file.

```
for field in model._meta.get_fields():

    # Skip reverse relationships, because mongo does not need them
    if field.auto_created and not field.concrete:
        continue

    # Many-to-Many: convert to list of IDs
    if isinstance(field, ManyToManyField):
        m2m_ids = list(
            getattr(obj, field.name).values_list("id", flat=True)
        )
        doc[field.name] = m2m_ids
        continue

    value = getattr(obj, field.name)

    # ForeignKey: convert related object to ID
    if field.is_relation and hasattr(value, "id"):
        doc[field.name] = value.id
    else:
        # Regular field, store as-is
        doc[field.name] = value
```

Appendix E:

```python
class Command(BaseCommand):
    def handle(self, *args, **kwargs):
        # ------------------------------------
        rpg_models = apps.get_app_config("rpg").get_models()

        for model in rpg_models:
            model_name = model.__name__.lower()
            collection = db[model_name]

            if model_name == "inventory":
                self.stdout.write(f"Migrating (embedded): {model_name} ...")

                collection.delete_many({})

                rows = model.objects.all()
                documents = []

                for obj in rows:
                    # base inventory fields
                    doc = {
                        "id": obj.id,
                        "character": obj.character.id
                    }

                    # embed related InventoryItem rows
                    embedded_items = []
                    for ii in obj.items.all(): # reverse relationship from InventoryItem
                        embedded_items.append({
                            "item": ii.item.id,
                            "quantity": ii.quantity
                        })
                    doc["items"] = embedded_items
                    documents.append(doc)

                if documents:
                    collection.insert_many(documents)

                self.stdout.write(
                    self.style.SUCCESS(f" {len(documents)} inventory migrated with embedded items")
                )
                continue

            self.stdout.write(f"Migrating: {model_name} ...")

            # Clear existing data in MongoDB collection
            collection.delete_many({})

            # Fetch SQL rows
            rows = model.objects.all()
            documents = []

            # ------------------------------------
            # CONVERT EACH SQL ROW INTO DOCUMENT
            # ------------------------------------
            for obj in rows:
                doc = {}

                for field in model._meta.get_fields():

                    # Skip reverse relationships, because mongo does not need them
                    if field.auto_created and not field.concrete:
                        continue

                    # Many-to-Many: convert to list of IDs
                    if isinstance(field, ManyToManyField):
                        m2m_ids = list(
                            getattr(obj, field.name).values_list("id", flat=True)
                        )
                        doc[field.name] = m2m_ids
                        continue

                    value = getattr(obj, field.name)

                    # ForeignKey: convert related object to ID
                    if field.is_relation and hasattr(value, "id"):
                        doc[field.name] = value.id
                    else:
                        # Regular field, store as-is
                        doc[field.name] = value

                documents.append(doc)

            # ------------------------------------
            # INSERT INTO MONGODB
            # ------------------------------------
            if documents:
                collection.insert_many(documents)

            self.stdout.write(
                self.style.SUCCESS(f"✔ {len(documents)} rows migrated from {model_name}")
            )
```

Appendix F:

```python
@method_decorator(csrf_exempt, name="dispatch")
class MongoFilterInventory(APIView):

    # Endpoints public / no authentication required
    permission_classes = [permissions.AllowAny]

    @swagger_auto_schema(
        operation_description="Filter inventory by character or item",
        manual_parameters=[
            openapi.Parameter("character", openapi.IN_QUERY, description="Character ID", type=openapi.TYPE_INTEGER),
            openapi.Parameter("item", openapi.IN_QUERY, description="Item ID", type=openapi.TYPE_INTEGER),
        ],
        responses={200: InventoryListSchema}
    )
    def get(self, request):
        query ={}

        character = request.GET.get("character")
        item = request.GET.get("item")

        if character:
            query["character"] = int(character)

        if item:
            query["items.item"] = int(item)

        inventories = [fix_id(doc) for doc in inventory_collection.find(query)]
        return Response(inventories, status=status.HTTP_200_OK)
```

Appendix G:

```
CREATE TABLE IF NOT EXISTS `rpgdb`.`user` (
 `id` INT NOT NULL AUTO_INCREMENT,
 `username` VARCHAR(45) NOT NULL,
 `email` VARCHAR(45) ) NOT NULL,
 `password` VARCHAR(45) ) NOT NULL,
 PRIMARY KEY (`id`),
 UNIQUE INDEX `id_UNIQUE` (`id` ASC) VISIBLE,
 UNIQUE INDEX `username_UNIQUE` (`username` ASC) VISIBLE,
 UNIQUE INDEX `email_UNIQUE` (`email` ASC) VISIBLE,
 UNIQUE INDEX `password_UNIQUE` (`password` ASC) VISIBLE)
```

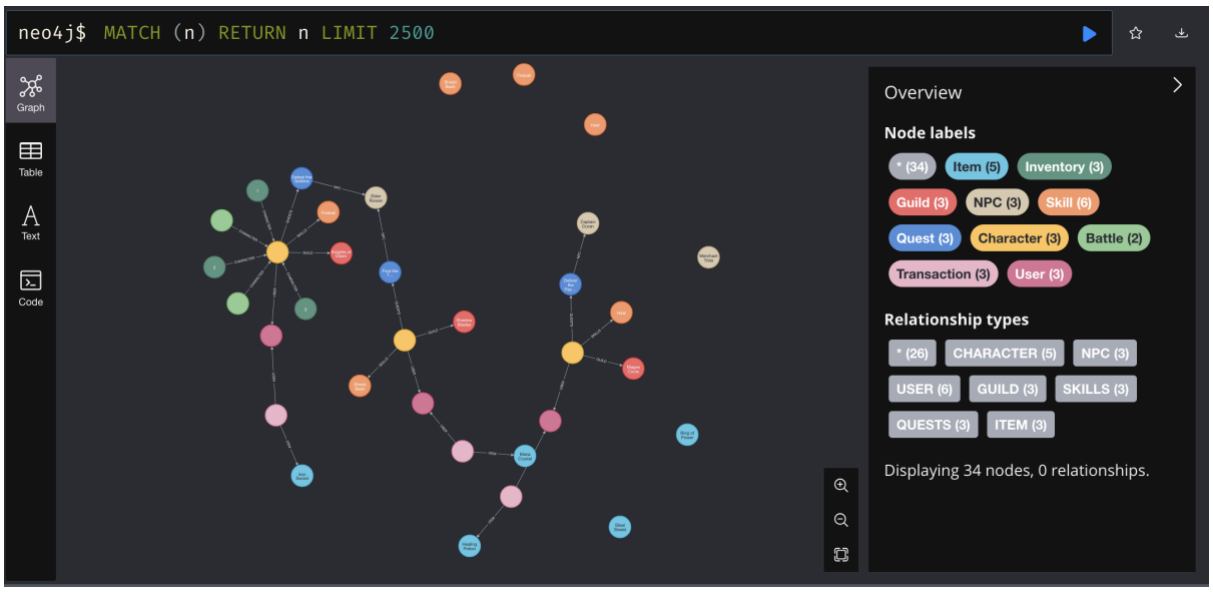
Appendix H:

```
CREATE TRIGGER trg_inventory_prevent_negative
BEFORE UPDATE ON rpg_inventory
FOR EACH ROW
BEGIN
    IF NEW.quantity < 0 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Inventory quantity cannot be negative.';
    END IF;
END$$
```

Appendix I:

```
CREATE TRIGGER trg_transaction_adjust_gold
AFTER INSERT ON rpg_transaction
FOR EACH ROW
BEGIN
    UPDATE rpg_character
    SET gold = gold - NEW.cost
    WHERE user_id = NEW.user_id;
END$$
```

Appendix J:



Appendix K:

```
class CharacterViewSet(viewsets.ModelViewSet):
    queryset = Character.objects.all()
    serializer_class = CharacterSerializer
    permission_classes = [IsAuthenticated, IsOwner]

    def get_queryset(self):
        return Character.objects.filter(user=self.request.user)
```

Appendix L:

```
class Character(models.Model):
    character_name = models.CharField(max_length=100, unique=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    level = models.IntegerField(default=1)
    hp = models.IntegerField(default=100)
    mana = models.IntegerField(default=50)
    xp = models.IntegerField(default=0)
    gold = models.IntegerField(default=0)

    guild = models.ForeignKey(Guild, on_delete=models.SET_NULL, null=True)

    skills = models.ManyToManyField(Skill, blank=True)
    quests = models.ManyToManyField(Quest, blank=True)

    def __str__(self):
        return self.character_name
```