Database Design and Implementation

Role Playing Game Database

Mandatory Assignment 1 + 2

Databases for Developers

Sofie Amalie Thorlund

Soth0003

Viktor Bach Mekis

Viba0001

07.10.2025

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 players, their characters, and all related gameplay elements such as inventories, items, quests, skills, battles and transactions with 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.

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, faction 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 or allow players to buy/sell items. This is where transactions must record item details, NPC involved, type of transaction and the total cost.

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 players, 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 tells 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 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.

[1] https://www.ibm.com/think/topics/entity-relationship-diagram

Conceptual ERD

The conceptual ERD is the most abstract and has least details. As seen in Figure 1, 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.

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, Inventory and Transaction. The additional entities can be seen in Figure 2, as the entities separating the entities from the previous figure.

The inclusion of associative entities such as character_has_guest and inventory_has_item 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.

Physical ERD

The Physical ERD serves as the actual design or blueprint of the database. As seen in Figure 3, 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.

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. In this process we divided the database into an assortment of tables with connecting relationships. The main objective of doing this, was to isolate data in a way, that additions, deletions and any modifications can be made in one table and then spread through the rest of the database via the already defined relationships. Initially we considered only working with one table that includes UserID, name, email, and Characters and so on. However, we realized that this structure would quickly become problematic, because what if the user wanted to create more than one character, then the user's information would be repeated for every entry, which leads to redundancy[2]. The database was normalized to at least Third Normal Form. This will ensure that each attribute depends only on the primary key, which will then eliminate redundancy and maintaining 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 3, each entry is represented as a table with columns (attributes), data types and defined relationships through primary and foreign keys.

[2] https://blog.dataengineerthings.org/understanding-database-normalization-from-basics-to-er-diagrams-488a53923cf4

```
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)
ENGINE = InnoDB;
```

This example 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, enforcing referential integrity.

How it works in in our tables:

user.id → Primary key of user

character.user_id → Foreign key referencing user(id)

character_has_quest.character_id → Foreign key referencing character(id)

inventory_has_item.inventory_id → Foreign key referencing inventory(id)

The primary keys prevent duplication, while foreign keys maintain logical consistency between related tables. This will ensure that character can't exist without an associated user or guild.

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:

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.

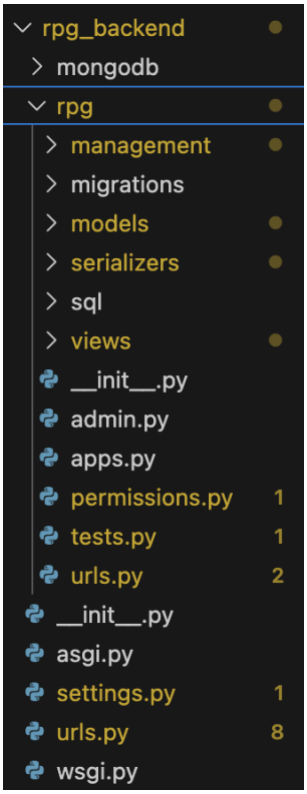
3. Application Layer: Django Implementation, Seeding, and API Testing

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 *User* → *Character* or *Inventory* → *Item* are expressed naturally through Django’s ForeignKey and ManyToManyField constructs.

3.2 Model Mapping Example

The following code snippet shows the class Character in the models folder:

```
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)
```

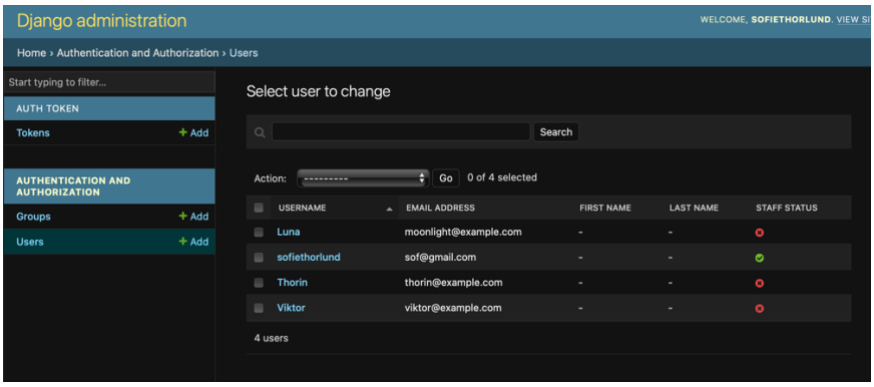
This reflects our MySQL schema exactly and demonstrates that our Django model mirror the physical relational structure.

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.

Code snippet of the CharacterViewSet:

```
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)
```

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
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:

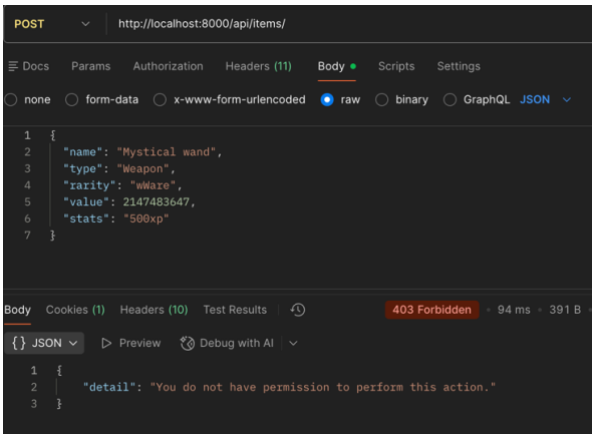
- Character creation and retrieval
- Inventory management
- Item CRUD operations
- Authentication (Token and JWT)
- Permissions (admin vs regular user)

Tests we covered:

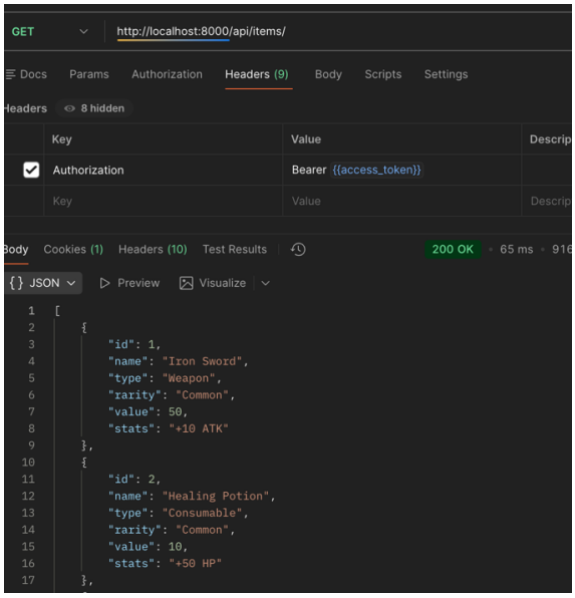
- Correct status codes

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

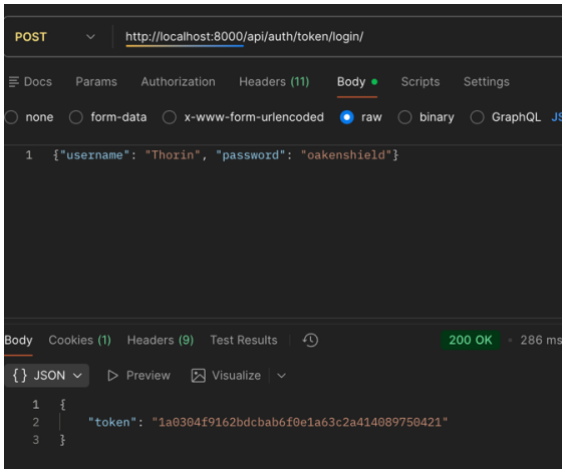
Example of creating a new item without authorization:



Example of getting all items with authorization:



Example of logging in and getting a token:



4 MongoDB

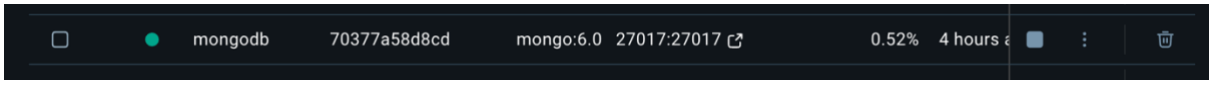
In Assignment 2, we are required to introduce a document-based representation of the RPG database and migrate data from MySQL into MongoDB. The motivation for this is that certain read-heavy queries, especially character profiles including inventory, skills, quests and NPC interactions, require multiple JOINs in SQL but can be retrived as a single nested document in MongoDB.

4.1 Running MongoDB with Docker

MongoDB was deployed using a persistent Docker container:

```
docker run -d \
  --name mongodb \
  -p 27017:27017 \
  -v mongo-data:/data/db \
  mongo:6.0
```

This ensures reproducibility and clean isolation of the NoSQL environment.



4.2 Migration Script (migrate_tomongo.py)

We implemented a standalone Django management command located at:

Rpg_back/rpg/management/commands/migrate_to_mongo.py

The script makes sure to:

- Connect to MongoDB using PyMongo
- Fetch SQL data using Django ORM
- Converting relational structures into nested document structures
- Handling ManyToMany relationships as embedded arrays
- Creating collections dynamically
- Ensuring migrations that is safe to run multiple times

Which can be seen in the short code snippet below:

```
# ------------------------------------
# 2. FETCH ALL MODELS FROM rpg APP
# ------------------------------------
rpg_models = apps.get_app_config("rpg").get_models()

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

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

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

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

4.3 SQL into MongoDB transformation

Here is an example of how we brought the normalized SQL version to the denormalized MongoDB version.

SQL version:

Table	Fields
Character	Id, name, level, user_id
Inventory	Id, character_id
InventoryItems	Inventory_id, item_id
Tiem	Id, name, value

MongoDB version: (screenshot from MongoDB compass)

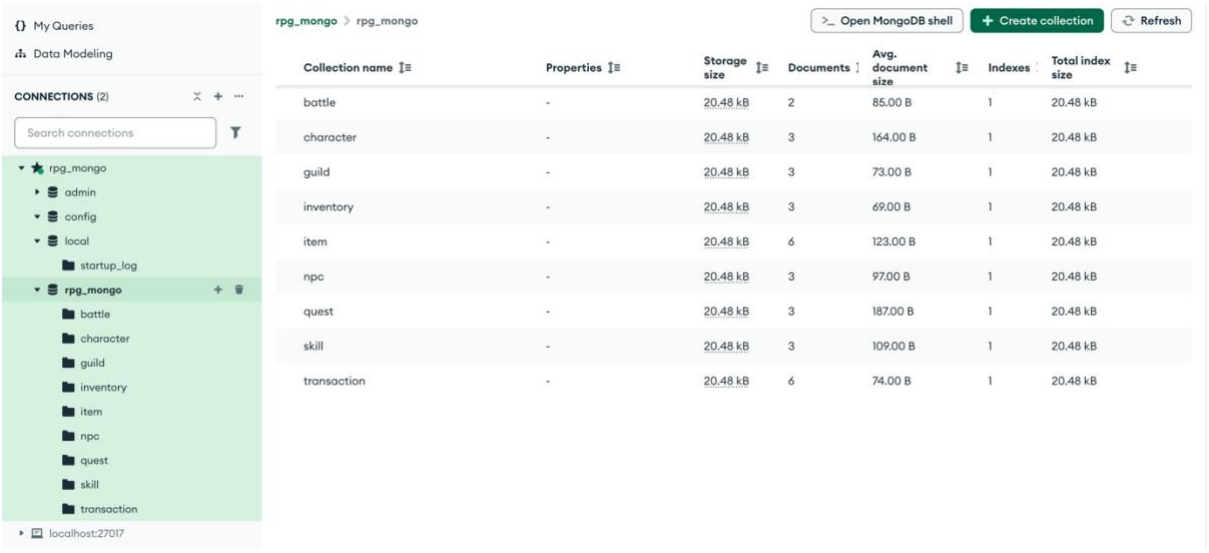
```
_id: ObjectId('6925f6251f723155a25b43bb')
id : 1
character_name : "Aelric"
level : 2
hp : 200
mana : 100
xp : 120
gold : 0
```

4.4 Validation Using MongoDB Compass

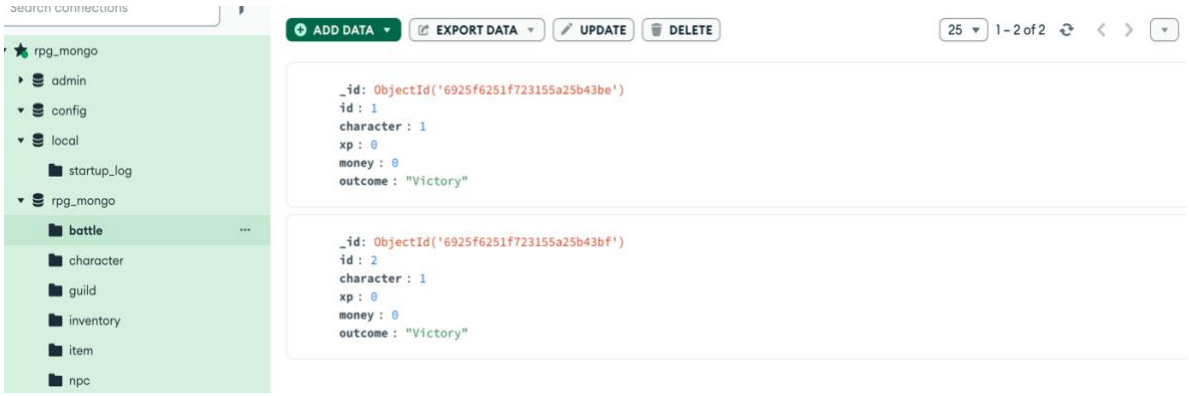
After running the migration, we validated data integrity using MongoDB Compass.

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

This is how our Compass connection window looks like:



And this is how a single document (Battle) expanded showing nested lists looks like:



Conclusion

Across both mandatory assignments, we successfully implemented two complementary data models for the RPG system. The relational MySQL database provides a normalized, transaction-safe foundation with strict integrity through foreign keys, triggers, views and stored procedures. This ensures reliable state management for characters, items, skills, quests and gameplay integrations.

In assignment 2, we extended the system with a MongoDB-based NoSQL representation optimized for read-heavy access patterns. Using a custom Django migration script and Dockerized MongoDB instance, we transformed relational SQL data into denormalized document structures that support fast retrieval of complex character profiles.

Together, these components demonstrate a full-stack, multi-model data architecture that is suitable for future game development or as a foundation for a hybrid backend system combining transactional and analytical workloads.

List of figures

Figure 1: Conceptual ERD

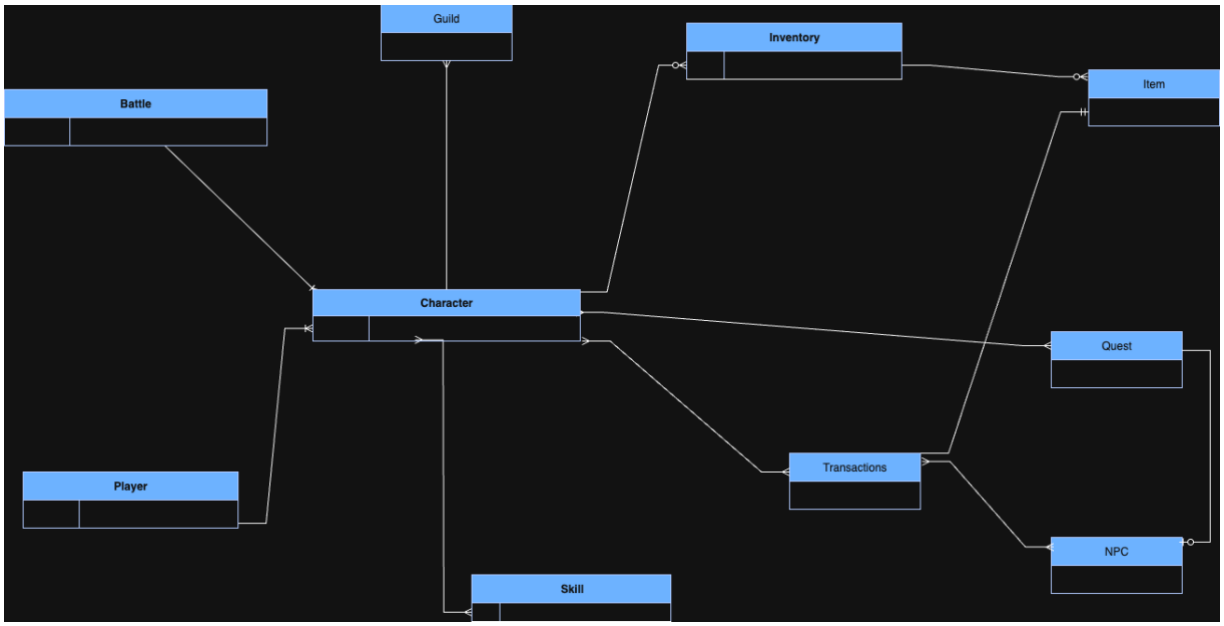


Figure 2: Logical ERD

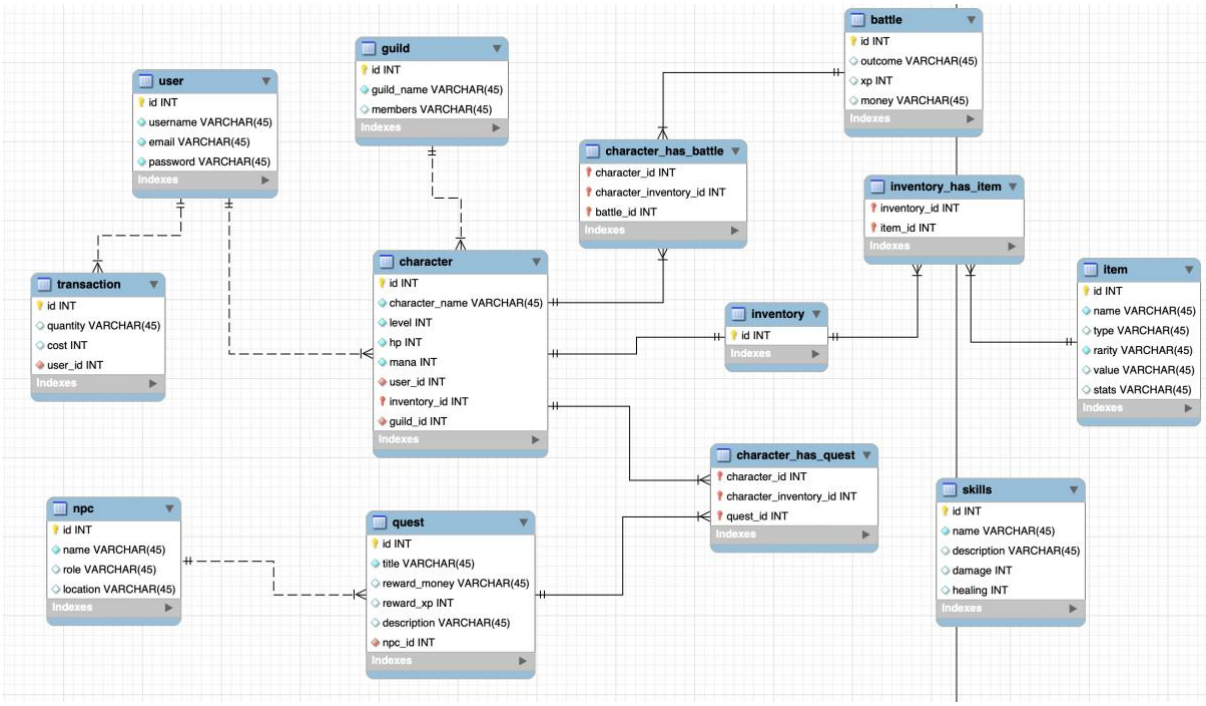


Figure 3: Physical ERD