Here’s a focused guide to the project’s \*\*Database Interaction\*\* layer—how game data (players, characters, matches, events, items) is modeled, stored, and managed so you can confidently make changes without breaking existing functionality.

---

## Database Setup

\* \*\*Initialization\*\*

\* The SQLite database is created (if missing) by calling `init\_db()` from \*\*models.py\*\*, which invokes SQLAlchemy’s `Base.metadata.create\_all(...)`.

\* This is already wired to run on import of the repositories module, so your DB file will exist once you start the app or run any repo code .

\* \*\*Location / Configuration\*\*

\* The connection URL is hard-coded in \*\*models.py\*\* as

```python

DATABASE\_URL = "sqlite:////home/ubuntu/coin\_clash/data/coin\_clash.db"

```

\* If you need to point at a different path or switch to another database engine, update this constant (or refactor to load from `config.yaml`).

\* \*\*Session Management\*\*

\* A `SessionLocal = sessionmaker(...)` is bound to the engine in \*\*models.py\*\*.

\* The helper `get\_db()` in \*\*repositories.py\*\* is a generator that yields a session and ensures it’s closed after use .

---

## ORM Models

All tables inherit from `Base = declarative\_base()`. Relationships between them map to SQL foreign keys, so ORM queries can span objects naturally.

### 1. `players`

| Column | Type | Notes |

| ------------------ | -------- | ----------------------------------------------------------- |

| `id` | Integer | Primary key |

| `username` | String | Unique, non-null |

| `balance` | Float | Player’s currency balance |

| `wins` | Integer | Total matches won |

| `kills` | Integer | Total kills |

| `total\_sui\_earned` | Float | Cumulative awards earned |

| `created\_at` | DateTime | Auto-set to now |

| \*\*Relationships\*\* | | `characters` (their owned Characters), `inventory` (items) |

### 2. `characters`

| Column | Type | Notes |

| ----------------- | -------- | -------------------------------------- |

| `id` | Integer | Primary key |

| `name` | String | E.g. “Warrior”, “Mage” |

| `owner\_username` | String | FK → `players.username` |

| `match\_id` | Integer | FK → `matches.id`, null until assigned |

| `is\_alive` | Integer | 1 = alive, 0 = dead |

| `created\_at` | DateTime | Auto-set |

| \*\*Relationships\*\* | | `player\_owner`, `match` |

### 3. `matches`

| Column | Type | Notes |

| --------------------- | -------- | --------------------------------------------------- |

| `id` | Integer | Primary key |

| `entry\_fee` | Float | Fee per character |

| `kill\_award\_rate` | Float | % of entry\\_fee paid per kill |

| `start\_method` | String | “cap” or “timeout” |

| `start\_threshold` | Integer | Cap size or timeout seconds |

| `start\_timestamp` | DateTime | When match went “active” |

| `end\_timestamp` | DateTime | When match completed |

| `winner\_character\_id` | Integer | FK → `characters.id`, null until declared |

| `status` | String | “pending” / “active” / “completed” |

| `created\_at` | DateTime | Auto-set |

| \*\*Relationships\*\* | | `participants` (Characters), `events` (MatchEvent) |

### 4. `match\_events`

| Column | Type | Notes |

| ------------------------ | -------- | ---------------------------------------- |

| `id` | Integer | Primary key |

| `match\_id` | Integer | FK → `matches.id` |

| `round\_number` | Integer | Round in which this event occurred |

| `event\_type` | String | Category (“direct\\_kill”, “story”, etc.) |

| `scenario\_source` | String | File ID or “generated” |

| `scenario\_text` | Text | Fully substituted text |

| `affected\_character\_ids` | String | CSV of Character IDs |

| `timestamp` | DateTime | Auto-set |

| \*\*Relationships\*\* | | `match` |

### 5. `items`

| Column | Type | Notes |

| -------------------- | ------- | --------------------------------------- |

| `id` | Integer | Primary key |

| `name` | String | Unique |

| `type` | String | Cosmetic, gear, consumable |

| `rarity` | String | Common, rare, legendary |

| `description` | Text | Free text |

| `on\_find\_hook\_info` | String? | Metadata for find-time hook (optional) |

| `on\_award\_hook\_info` | String? | Metadata for award-time hook (optional) |

| \*\*Relationships\*\* | | `player\_items` |

### 6. `player\_items` (join table)

| Column | Type | Notes |

| ----------------- | -------- | --------------------------------------- |

| `player\_id` | Integer | FK → `players.id`, part of composite PK |

| `item\_id` | Integer | FK → `items.id`, part of composite PK |

| `quantity` | Integer | Count |

| `acquired\_at` | DateTime | Auto-set |

| \*\*Relationships\*\* | | `player`, `item` |

---

## Repositories & Session Usage

All DB interactions go through \*\*Repository\*\* classes—this decouples SQLAlchemy from business logic and makes unit-testing easier.

\* \*\*`BaseRepo`\*\*

\* Holds a `self.db: Session`

\* Concrete repos inherit common patterns (add, get, commit, refresh) .

\* \*\*`SqlPlayerRepo`\*\*

\* CRUD for `Player` (create, fetch by username, update balance, wins, kills, earned SUI).

\* Inventory methods: `get\_player\_inventory()`, `add\_item\_to\_inventory()` .

\* \*\*`SqlCharacterRepo`\*\*

\* `create\_character(name, owner\_username)`

\* Assign to match, mark dead/alive, and query by player or match.

\* \*\*`SqlMatchRepo`\*\*

\* `create\_match(...)` – new match record

\* `update\_match\_status(id, status)`, `set\_match\_start\_time()`, `set\_match\_end\_time()`, `set\_match\_winner()` .

\* \*\*`SqlEventRepo`\*\*

\* `create\_match\_event(...)` – logs each event during simulation

\* `get\_events\_for\_match(match\_id)` to replay or analyze.

\* \*\*`SqlItemRepo`\*\*

\* `get\_item\_by\_name()`, `create\_item()`, `get\_all\_items()`.

\* \*\*Session injection\*\*

\* In \*\*`\_\_main\_\_.py`\*\*, `db\_session = next(get\_db())`

\* Repos built with `SqlXRepo(db\_session)` in `setup\_repositories()` and passed into the engine .

\* At the end of simulation the session is closed.

---

## How the Engine Uses the DB

Within \*\*`MatchEngine`\*\* (in `engine.py`):

1. \*\*Match lifecycle\*\*

\* `match\_repo.update\_match\_status(..., "active")` & `match\_repo.set\_match\_start\_time()`

\* After rounds: `set\_match\_winner()`, `update\_match\_status(..., "completed")`, `set\_match\_end\_time()` .

2. \*\*Event logging\*\*

\* Each round’s outcome is saved via `event\_repo.create\_match\_event(...)`, capturing text, round, effected characters .

3. \*\*Player economics & stats\*\*

\* On a kill: `player\_repo.add\_kill()`, calculate payout = `entry\_fee \* kill\_award\_rate`, then (stub) award SUI and—once implemented—call `player\_repo.update\_player\_balance()` and `player\_repo.add\_sui\_earned()`.

\* On match win: similarly compute and call `add\_sui\_earned()` + `update\_player\_balance()`, and `add\_win()` .

4. \*\*Character status\*\*

\* Elimination calls `character\_repo.kill\_character()` (or equivalent) to set `is\_alive=0` and `match\_id` association.

---

## Making Safe Modifications

\* \*\*Schema changes\*\*

\* Update the model in \*\*`models.py`\*\* (e.g. add a column).

\* Add corresponding methods in the repo interface and its `Sql…` implementation.

\* Run `init\_db()` or use migrations (not yet integrated).

\* \*\*Adding new stats\*\*

\* Extend the `Player` model with new fields (e.g. `assists`).

\* Provide repo methods to update/query them.

\* Hook into `MatchEngine` where appropriate.

\* \*\*Leaderboards / Queries\*\*

\* Build new repo methods for ranking, e.g.:

```python

def get\_top\_players\_by\_wins(self, limit=10):

return self.db.query(Player).order\_by(Player.wins.desc()).limit(limit).all()

```

\* Expose via your bot or API layer as needed.

\* \*\*Switching DB engines\*\*

\* Refactor `DATABASE\_URL` into `config.yaml` and reload in `models.py`.

\* Ensure sessions and repos still use the updated URL.

---

This structure keeps all SQL concerns isolated in \*\*`models.py`\*\* and \*\*`repositories.py`\*\*, with \*\*`MatchEngine`\*\* driving logic via those clean interfaces. You can now confidently extend fields, add new tables (e.g. achievements), or tweak economic rules knowing where and how data flows through the system.