Here’s a focused guide on \*\*Player Profile Handling & Economic Updates\*\*, explaining where player data lives, how it’s manipulated, and how the economic flows tie into the game logic. It’s organized so that a new developer can trace and safely modify any of these behaviors without breaking existing functionality.

---

## 1. Project Organization (Relevant to Player Profiles)

```

.

├── \_\_main\_\_.py # Entry-point: wires repos, config, and MatchEngine

├── config.yaml # All economic parameters (fees, rates, cuts)

├── core/

│ ├── models.py # ORM definitions (Player, Character, Match, etc.)

│ ├── repositories.py # Abstract and SQLite implementations of PlayerRepo, etc.

│ ├── engine.py # MatchEngine: drives rounds, applies events, does payouts

│ ├── config\_loader.py # Loads config.yaml

│ ├── scenario\_loader.py # Loads JSON scenarios into memory

│ └── utils.py # Helpers: name generator, seedable RNG

├── data/coin\_clash.db # SQLite database backing Player & inventory persistence

└── scenarios/… # Event text JSON files (affecting stats/inventory)

```

This layout cleanly separates:

\* \*\*Data models & persistence\*\* (`core/models.py` + `core/repositories.py`)

\* \*\*Game flow & economic logic\*\* (`\_\_main\_\_.py` + `core/engine.py`)

\* \*\*Configuration & content\*\* (`config.yaml`, `scenarios/`)

---

## 2. Player Profile: Data Model & Relationships

### 2.1 `Player` Model

Defined in \*\*`core/models.py`\*\*, `Player` holds all per-user state:

```python

class Player(Base):

\_\_tablename\_\_ = "players"

id = Column(Integer, primary\_key=True)

username = Column(String, unique=True, nullable=False)

balance = Column(Float, default=0.0)

wins = Column(Integer, default=0)

kills = Column(Integer, default=0)

total\_sui\_earned = Column(Float, default=0.0)

created\_at = Column(DateTime, server\_default=func.now())

characters = relationship("Character", back\_populates="player\_owner")

inventory = relationship("PlayerItem", back\_populates="player")

```

\* \*\*`balance`\*\*: current SUI balance.

\* \*\*`wins`, `kills`, `total\_sui\_earned`\*\*: aggregate stats used for leaderboards or cosmetics.&#x20;

### 2.2 `PlayerItem` (Inventory)

Represents how many of each `Item` a player holds:

```python

class PlayerItem(Base):

\_\_tablename\_\_ = "player\_items"

player\_id = Column(Integer, ForeignKey("players.id"), primary\_key=True)

item\_id = Column(Integer, ForeignKey("items.id"), primary\_key=True)

quantity = Column(Integer, default=1)

acquired\_at = Column(DateTime, server\_default=func.now())

…

```

Cosmetics or gear found during play are added here .

---

## 3. Repository Layer: `PlayerRepo` API

All database interactions go through `PlayerRepo` (and its concrete `SqlPlayerRepo`), ensuring isolation of persistence logic . Key methods:

| Method | Purpose |

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

| `get\_or\_create\_player(username)` | Look up or insert a new `Player`. |

| `update\_player\_balance(player\_id, amount\_change)` | Add (or subtract) SUI from `balance`. |

| `add\_win(player\_id)` | Increment `wins`. |

| `add\_kill(player\_id)` | Increment `kills`. |

| `add\_sui\_earned(player\_id, amount)` | Increment `total\_sui\_earned`. |

| `get\_player\_inventory(player\_id)` | List all `PlayerItem`s. |

| `add\_item\_to\_inventory(player\_id, item\_id, quantity=1)` | Add or bump an item in inventory. |

\*\*Modifying behaviors\*\* (e.g. logging each balance update to an audit table) is as simple as extending or decorating these methods without touching game-flow code.

---

## 4. Economic Flows in Game Logic

### 4.1 Entry Fees & Protocol Cut

In `\_\_main\_\_.py`’s `simulate\_character\_purchases(...)`, for each player:

1. \*\*Determine\*\* how many characters to buy (1–3 by default).

2. \*\*Compute\*\*

\* `total\_cost = num\_chars × entry\_fee`

\* `cut\_amount = total\_cost × protocol\_cut\_rate` (10%, 8%, 6% tiers)

\* `net\_fee\_to\_pool = total\_cost − cut\_amount`

3. \*\*(Stub)\*\* Deduct from player via

```python

# player\_repo.update\_player\_balance(player.id, -total\_cost)

```

4. \*\*Accumulate\*\* totals for prize-pool logic .

> \*\*To enable real fee‐checks\*\*: uncomment and validate `update\_player\_balance` returns non-negative before character creation.

### 4.2 Kill-Award Payouts

\* \*\*Configured\*\* in `config.yaml` (`kill\_award\_rate\_default: 0.1`) .

\* During \*\*lethal events\*\*, after determining a kill, you should call:

```python

player\_repo.add\_kill(killer\_id)

player\_repo.update\_player\_balance(killer\_id, entry\_fee × kill\_award\_rate)

player\_repo.add\_sui\_earned(killer\_id, entry\_fee × kill\_award\_rate)

```

\*Note: The current stub logs kill events but does not yet award per-kill payouts—this is the spot to plug in those calls.\*

### 4.3 Final Winner Payout

At match-end in `MatchEngine.run\_match()`:

1. \*\*`add\_win`\*\* for the winner.

2. \*\*`\_calculate\_payouts`\*\* computes `(protocol\_cut, total\_kill\_awards, winner\_payout)`.

3. \*\*Award SUI\*\* via:

```python

player\_repo.add\_sui\_earned(winner\_id, winner\_payout)

player\_repo.update\_player\_balance(winner\_id, winner\_payout)

```

4. \*\*Persist\*\* winner character on `Match` record.&#x20;

---

## 5. Stats & Inventory Hooks

\* \*\*Kills & Wins\*\*: tracked via `add\_kill` / `add\_win`.

\* \*\*Total SUI Earned\*\*: updated on each payout via `add\_sui\_earned`.

\* \*\*Item Finds\*\* (non-lethal story events): in `MatchEngine.\_process\_event`, after logging an “item\\_find” or similar event, call

```python

item\_repo.add\_item\_to\_inventory(player\_id, item\_id)

```

and update DB. This ensures cosmetics persist across matches .

---

## 6. Character Ownership & Naming

\* \*\*`Character` Model\*\* links back to `Player` via `owner\_username` and carries `is\_alive` state .

\* \*\*Sequential Naming\*\* is done via `get\_next\_character\_name()` in `core/utils.py`; can be swapped for any naming strategy without touching engine logic.

---

## 7. Validation Checklist

Before handing off or extending:

\* \*\*Model Changes\*\*: if adding fields to `Player`, update both `models.py` \*\*and\*\* any affected repo methods.

\* \*\*Economic Parameters\*\*: adjust `config.yaml`—all currency logic reads from config.

\* \*\*Persistence\*\*: all state changes must go through repo methods; avoid raw SQL in engine.

\* \*\*Tests\*\*: any change to fee or reward logic should be covered by unit tests around:

\* Entry-fee charging

\* Kill-award payouts

\* Winner payout rounding

\* Inventory additions

This guide ensures you can locate, understand, and modify any part of the player-economy pipeline—from DB schema to in-match awarding—without side-effects on unrelated game logic.