Below is a focused guide to the \*\*match data\*\* in Coin Clash—how it’s defined in the ORM, managed by repositories, and driven through the engine—so that a new developer can confidently adapt match logic without breaking existing structure.

---

## 1. Data Models (core/models.py)

1. \*\*Match\*\*

\* \*\*Table\*\*: `matches`

\* \*\*Key fields\*\*:

\* `id` – primary key

\* `entry\_fee`, `kill\_award\_rate` – per-match economic settings

\* `start\_method` (“cap” or “timeout”), `start\_threshold` – when to fire off the match

\* `start\_timestamp`, `end\_timestamp` – persisted lifecycle markers

\* `winner\_character\_id` – FK to winning `Character`

\* `status` (pending → active → completed)

\* \*\*Relationships\*\*:

\* `participants` – all `Character` in this match

\* `events` – all `MatchEvent` rows for logging rounds&#x20;

2. \*\*MatchEvent\*\*

\* \*\*Table\*\*: `match\_events`

\* \*\*Key fields\*\*:

\* `id`, `match\_id`, `round\_number`

\* `event\_type` (e.g. direct\\_kill, comeback…)

\* `scenario\_source`, `scenario\_text` – reference to the sampled scenario

\* `affected\_character\_ids` – comma-separated IDs of victims/resurrectees

\* `timestamp` – when logged

\* \*\*Backref\*\*: `match.events`&#x20;

3. \*\*Character\*\*

\* \*\*Table\*\*: `characters`

\* Fields include `id`, `name`, `owner\_username` (FK → Player), `match\_id` (FK → Match), and `is\_alive` flag

\* Used as the unit of participation; belongs to one `Match` at a time&#x20;

---

## 2. Repository Layer (core/repositories.py)

Repositories abstract all persistence so engine logic never touches SQLAlchemy directly.

1. \*\*`SqlMatchRepo`\*\* implements `MatchRepo` with methods to:

\* \*\*Create\*\* a new match

\* \*\*Fetch\*\* by `id`

\* \*\*Update\*\* status (`pending` → `active` → `completed`)

\* \*\*Set\*\* `start\_timestamp`, `end\_timestamp`, and `winner\_character\_id`&#x20;

2. \*\*`SqlEventRepo`\*\* implements `EventRepo` with:

\* \*\*create\\_match\\_event(...)\*\* – inserts a `MatchEvent` row each time an event fires

\* \*\*get\\_events\\_for\\_match(match\\_id)\*\* – to replay or audit a match’s log&#x20;

3. \*\*`SqlCharacterRepo`\*\* (and its base abstract) handles:

\* \*\*Assigning\*\* characters into a match (`assign\_character\_to\_match`)

\* \*\*Updating\*\* alive/dead status (`update\_character\_status`)&#x20;

---

## 3. Engine Integration (core/engine.py)

`MatchEngine` drives the simulation and uses repos to persist every key step.

1. \*\*Initialization\*\*

```python

self.match = match\_repo.get\_match\_by\_id(match\_id)

self.entry\_fee = self.match.entry\_fee

self.kill\_award\_rate = self.match.kill\_award\_rate

```

­– loads settings from the `Match` row&#x20;

2. \*\*Starting the Match\*\*

```python

match\_repo.update\_match\_status(self.match\_id, "active")

match\_repo.set\_match\_start\_time(self.match\_id)

```

­– flips status and stamps the start time&#x20;

3. \*\*Per‐Round Event Loop\*\*

\* For each event (primary or extra), after sampling and effect application, it calls:

```python

event\_repo.create\_match\_event(

match\_id, round\_number, event\_type,

scenario\_source, scenario\_text, affected\_ids

)

```

to append to `match\_events` and the in-memory `match\_log`&#x20;

4. \*\*Ending the Match\*\*

\* When one `Character` remains:

1. Update player stats (wins, kills, SUI) via `PlayerRepo`

2. Calculate payouts (protocol cut, kill awards, winner payout)

3. Persist winner and status:

````python

match\_repo.set\_match\_winner(self.match\_id, winner.id)

match\_repo.update\_match\_status(self.match\_id, "completed")

match\_repo.set\_match\_end\_time(self.match\_id)

``` :contentReference[oaicite:18]{index=18}:contentReference[oaicite:19]{index=19}

````

---

## 4. Typical Match Workflow

1. \*\*Match Creation\*\* (via CLI or main script)

\* `match = match\_repo.create\_match(...)`

2. \*\*Character Purchases\*\*

\* Simulate entry-fee payments and call `char\_repo.assign\_character\_to\_match(char.id, match.id)`.

3. \*\*Run Engine\*\*

\* `engine = MatchEngine(match.id, …); winner, log = engine.run\_match(participants)`

4. \*\*Audit & Extend\*\*

\* Query `match\_repo.get\_match\_by\_id` for metadata.

\* Query `event\_repo.get\_events\_for\_match(match.id)` to reconstruct the round log.

---

## 5. Tips for Safe Modifications

\* \*\*Adding a field\*\* to `Match` or `MatchEvent`:

1. Update the ORM class in \*\*models.py\*\*.

2. Run `init\_db()` or adjust migrations so the column exists in SQLite.

3. Expose new field in the corresponding repo method (e.g. in `create\_match` or `create\_match\_event`).

4. Modify `MatchEngine` to read/write the new property where appropriate—relying on repos for persistence.

\* \*\*Changing status semantics\*\* (e.g., adding a “paused” state):

\* Extend `status` in the ORM default; update `SqlMatchRepo.update\_match\_status`; then insert corresponding calls in the engine at new transition points.

\* \*\*Extending events\*\* (new event types or extra payload):

\* Add to `MatchEvent.event\_type` enum-style handling.

\* In `MatchEngine.\_log\_event`, include any extra fields in the call to `create\_match\_event`.

\* Update consumers (e.g., replay/Audit UIs) to handle the new data.

By following this mapping—from ORM definitions through repository abstractions to engine usage —a developer can safely trace, extend, or refactor any match-related feature without unintended side effects.