Here’s a guide focused on introducing an \*\*API layer\*\*—a thin intermediary between the core game logic and any future UI (e.g. a Telegram bot)—so that newcomers can safely hook in new features or objectives without touching the engine itself.

---

## 1. Project Structure & Core Responsibilities

At a high level, \*\*Coin Clash\*\* is organized like this:

```

.

├── \_\_main\_\_.py # standalone simulation entrypoint

├── config.yaml # all tunable parameters (fees, weights, etc.)

├── core/ # pure-Python “engine” modules

│ ├── config\_loader.py # reads config.yaml into a dict

│ ├── utils.py # seedable RNG, name generators, misc helpers

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

│ ├── repositories.py # abstract + SQLite repos (PlayerRepo, MatchRepo, ...)

│ ├── scenario\_loader.py # loads JSON scenario pools by category

│ └── engine.py # `MatchEngine`: the full match flow :contentReference[oaicite:0]{index=0}:contentReference[oaicite:1]{index=1}

├── scenarios/ # JSON files for each event category :contentReference[oaicite:2]{index=2}:contentReference[oaicite:3]{index=3}

│ ├── direct\_kill.json

│ ├── self.json

│ └── …

└── tests/ # (empty) placeholder for unit tests

```

\* \*\*config\\_loader.py\*\*: single function `load\_config()` returns all parameters (fees, event‐weights, thresholds) .

\* \*\*scenario\\_loader.py\*\*: `load\_scenarios()` returns a dict of lists of scenario texts, keyed by category.

\* \*\*models.py\*\* & \*\*repositories.py\*\*: define DB models and abstract repositories plus SQL implementations for persistence (players, matches, events, items) .

\* \*\*engine.py\*\*: central `MatchEngine` class orchestrates rounds: sampling event types, picking scenarios, placeholder substitution, applying kills/resurrections, logging via `EventRepo`, payout calculation, and match lifecycle (`pending → active → completed`) .

\* \*\*\*\*main\*\*.py\*\*: example script showing how to wire up `load\_config()`, repos, simulate purchases, and call `MatchEngine.run\_match()` .

---

## 2. Why an API Layer?

\* \*\*Separation of Concerns\*\*: Keep UI code (Telegram, web, CLI) completely separate from core logic.

\* \*\*Stable Contract\*\*: Define a small set of service methods; core can evolve internally without breaking adapters.

\* \*\*Dependency Injection\*\*: Pass in config, repos, RNG, etc., so you can swap or mock for tests or different backends.

\* \*\*DTOs & Validation\*\*: Convert raw DB models into plain data objects for UI; centralize input checks.

---

## 3. Designing the API Layer

### 3.1. Location & Structure

Create a new module, e.g.:

```

coin\_clash/

└── api/

└── game\_service.py

```

### 3.2. Core Service Class

```python

# coin\_clash/api/game\_service.py

from typing import List, Dict, Any

from core.repositories import PlayerRepo, CharacterRepo, MatchRepo, EventRepo, ItemRepo

from core.engine import MatchEngine

from core.config\_loader import load\_config

from core.scenario\_loader import load\_scenarios

from core.utils import SeedableRandom

class GameService:

def \_\_init\_\_(

self,

player\_repo: PlayerRepo,

character\_repo: CharacterRepo,

match\_repo: MatchRepo,

event\_repo: EventRepo,

item\_repo: ItemRepo,

seed: int = None

):

self.config = load\_config()

self.scenarios = load\_scenarios()

self.random = SeedableRandom(seed)

self.player\_repo = player\_repo

self.character\_repo = character\_repo

self.match\_repo = match\_repo

self.event\_repo = event\_repo

self.item\_repo = item\_repo

def create\_match(self, entry\_fee: float, kill\_award\_rate: float,

start\_method: str, start\_threshold: int) -> int:

match = self.match\_repo.create\_match(entry\_fee, kill\_award\_rate,

start\_method, start\_threshold)

return match.id

def purchase\_characters(self, match\_id: int,

username: str, count: int) -> List[int]:

"""Charge entry fee, create Character records, assign to match."""

# 1. Fetch or create player

player = self.player\_repo.get\_or\_create\_player(username)

# 2. Calculate fee & protocol cut per config

# 3. Update player balance stub or real

# 4. Loop count times:

# name = get\_next\_character\_name()

# char = self.character\_repo.create\_character(name, username)

# self.character\_repo.assign\_character\_to\_match(char.id, match\_id)

# collect char.id

# 5. Return list of character IDs

…

def start\_match(self, match\_id: int, participants: List[int]) -> Dict[str, Any]:

"""

Instantiates MatchEngine, runs the match, returns dict:

{ "winner": { … }, "events": [ … ], "payouts": { … } }

"""

engine = MatchEngine(

match\_id=match\_id,

config=self.config,

scenarios=self.scenarios,

player\_repo=self.player\_repo,

character\_repo=self.character\_repo,

match\_repo=self.match\_repo,

event\_repo=self.event\_repo,

item\_repo=self.item\_repo,

random\_seed=self.random.seed

)

winner\_char, log = engine.run\_match(participants)

# Map winner\_char → player info, payouts, etc.

return {

"winner": {

"character\_id": winner\_char.id,

"display\_name": winner\_char.display\_name,

"owner": winner\_char.owner\_username

},

"log": log

}

def get\_match\_events(self, match\_id: int) -> List[Dict[str, Any]]:

"""Fetch all logged events for UI display."""

events = self.event\_repo.get\_events\_for\_match(match\_id)

# Convert ORM to dicts

return [ { "round": e.round\_number, "text": e.scenario\_text, … } for e in events ]

```

> Notice how \*\*GameService\*\* never deals with sockets, Telegram updates, or JSON marshalling. It purely wraps core logic .

---

## 4. Integrating with a Telegram Bot

1. \*\*Dependency Setup\*\*

```python

from core.repositories import SessionLocal, SqlPlayerRepo, …

from api.game\_service import GameService

db = SessionLocal()

repos = {

"player": SqlPlayerRepo(db),

"character": SqlCharacterRepo(db),

…

}

service = GameService(\*\*repos, seed=42)

```

2. \*\*Bot Handlers\*\*

\* \*\*/start\\_match\*\* → call `service.create\_match(...)` → reply “Match #42 created.”

\* \*\*/buy 2\*\* → take username from update, call `service.purchase\_characters(42, user, 2)` → reply with character IDs.

\* \*\*/run\*\* → call `service.start\_match(42, [list of char IDs])` → stream back `log` lines one by one or as a batch.

\* \*\*/events\*\* → call `service.get\_match\_events(42)` → format for chat.

Because all domain logic lives in `core/`, handlers remain trivial.

---

## 5. Best Practices & Extensions

\* \*\*Dependency Injection\*\* enables injecting mock repos for unit tests.

\* \*\*DTO Layer\*\* (e.g. Pydantic models) can sit between `GameService` and UI for validation.

\* \*\*Versioning\*\*: Keep the service interface stable; if core logic changes, adapt only `game\_service.py`.

\* \*\*New Objectives\*\*: Adding a new “tournament” flow or alternate economy simply means adding methods to the API layer—core engine remains untouched.

\* \*\*Testing\*\*: Write unit tests against `GameService` methods using in-memory SQLite or mock repos.

---

## 6. Completeness Checklist

\* [x] \*\*Configuration\*\*: all fees, weights, thresholds loaded once in service&#x20;

\* [x] \*\*Scenario Pools\*\*: categorised JSON → `scenarios` dict&#x20;

\* [x] \*\*Player & Character Management\*\*: repos abstract persistence&#x20;

\* [x] \*\*Match Flow\*\*: start, rounds, payout, end handled by `MatchEngine`&#x20;

\* [x] \*\*Logging & Events\*\*: stored via `EventRepo`, retrievable for UI

\* [x] \*\*Randomness\*\*: seedable for reproducibility via `SeedableRandom`&#x20;

\* [x] \*\*Entry-Point Isolation\*\*: `\_\_main\_\_.py` used only for CLI/testing&#x20;

With this API‐layer guide, a new developer can immediately see where to hook UI code, where to grow new features, and how to preserve the integrity of the existing game logic.