BACKEND — Contrato y guía de implementación (v1)

Proyecto: Random Reversi
Stack backend: Python + FastAPI (API REST), PostgreSQL, Docker/Docker Compose

12. Hu Qiheng Propuesta técnica…

Objetivo del doc: que backend pueda implementar un servidor autoritativo que web (React+TS) y Android (Kotlin) consuman sin inventar formatos.

---

1) Principios de arquitectura (lo que asumimos)

1. **Servidor autoritativo**: el cliente nunca decide si una jugada es válida.
2. **API REST** como canal principal (v1): fácil de integrar, luego se puede añadir WebSockets sin romper nada.
   12. Hu Qiheng Propuesta técnica…
3. **Estado de partida centralizado**: el backend devuelve siempre un `GameState` completo (o casi completo) para que el cliente solo renderice.
4. **Persistencia en PostgreSQL** de usuarios y partidas (al menos estado serializado)
   12. Hu Qiheng Propuesta técnica…
5. **Contenerización** con Docker/Compose para entorno reproducible
   12. Hu Qiheng Propuesta técnica…

---

2) Contrato de datos (clientes ↔ backend)

2.1 Tipos base (JSON)

Piece: `"EMPTY" | "BLACK" | "WHITE" | "RED" | "BLUE"`

Cell

```
{
  "piece": "EMPTY",
```

```
  "special": "MYSTERY",
  "locked": false
}
```

- `special` opcional: `"MYSTERY"` representa casilla “?”
- `locked` opcional (reservado para habilidades)

Player

```
{
  "id": "u_123",
  "name": "Ana",
  "color": "BLACK",
  "seat": 0
}
```

2.2 Ability (extensible)

Formato: objeto con `type` + params.

```
{ "type": "BOMB", "radius": 2 }

{ "type": "SKIP_TURN" }
```

Lista inicial (se puede ampliar sin romper el contrato):

- `BOMB { radius: 1 | 2 }`
- `SKIP_TURN {}`
- `SWAP_COLORS {}`
- `STEAL_ABILITY {}`
- `GRAVITY {}`

  Importante: el cliente no necesita implementar la lógica de habilidades; solo mostrar y enviar.

---

3) GameState (la respuesta canónica del backend)

El backend devuelve esto en:

- `joinGame`
- `getGame`
- `sendAction`

```
{
  "gameId": "g_001",
  "mode": "DUEL_1V1",
  "status": "IN_PROGRESS",
  "board": {
    "size": 8,
    "cells": [[{ "piece": "EMPTY" }]]
  },
  "players": [
    { "id": "u1", "name": "Ana", "color": "BLACK", "seat": 0 },
    { "id": "u2", "name": "Luis", "color": "WHITE", "seat": 1 }
  ],
  "currentTurnPlayerId": "u1",
  "legalMoves": [{ "row": 2, "col": 3 }],
  "abilitiesByPlayerId": {
    "u1": [{ "type": "BOMB", "radius": 1 }],
    "u2": []
  },
  "scoresByPlayerId": {
    "u1": 12,
    "u2": 10
  },
  "lastAction": { "type": "MAKE_MOVE", "row": 2, "col": 3 },
  "updatedAt": "2026-02-10T10:00:00.000Z"
}
```

Notas de implementación:

- `legalMoves` recomendado para UI (ahorra calcular en frontend).
- `scoresByPlayerId` puede ser simple “número de fichas” (en v1).
- `updatedAt` se usa para polling/refresh.

---

4) Acciones del cliente (GameAction)

El cliente solo envía acciones. El backend valida y actualiza.

MAKE_MOVE

```
{ "type": "MAKE_MOVE", "row": 3, "col": 5 }
```

USE_ABILITY

```
{
  "type": "USE_ABILITY",
  "ability": { "type": "BOMB", "radius": 1 },
  "target": { "row": 4, "col": 4 }
}
```

`target` opcional (según habilidad).

PASS (opcional)

```
{ "type": "PASS" }
```

---

5) Errores estándar (ApiError)

El backend responde con estructura uniforme:

```
{
  "error": {
    "code": "INVALID_MOVE",
    "message": "La jugada no es válida."
  }
}
```

Códigos recomendados:

- `UNAUTHORIZED`
- `GAME_NOT_FOUND`
- `GAME_NOT_JOINABLE`

- `NOT_YOUR_TURN`
- `INVALID_MOVE`
- `ABILITY_NOT_OWNED`
- `ABILITY_INVALID_TARGET`

---

6) Endpoints REST (FastAPI) — v1

> Ruta base recomendada: `/api/v1`

Auth

- `POST /api/v1/auth/register`
- `POST /api/v1/auth/login`
- `GET /api/v1/me`

Auth: token tipo Bearer (JWT o similar).

Games

- `POST /api/v1/games`
  - body: `{ "mode": "DUEL_1V1" | "FREE_FOR_ALL_4P" }`
  - resp: `{ "gameId": "g_001" }`
- `POST /api/v1/games/{gameId}/join`
  - resp: `GameState`
- `GET /api/v1/games/{gameId}`
  - resp: `GameState`
- `POST /api/v1/games/{gameId}/actions`
  - body: `{ "action": GameAction }`
  - resp: `GameState`
  - error: `{ error: ApiError }`

Semántica importante:

- `actions` es idempotente *solo si* la acción no se aplica dos veces; recomendación: validar turno y que la acción corresponde al estado actual.
- Si queréis robustez: añadir `clientActionId` o `expectedUpdatedAt` (optimistic concurrency) *en v2*.

---

7) Modelo de persistencia (PostgreSQL) — mínimo viable

El objetivo es: *reconectar y retomar partidas*.

Tabla `users`

- `id` (uuid o texto)
- `email` (unique)
- `name`
- `password_hash`
- `created_at`

Tabla `games`

- `id`
- `mode`
- `status`
- `created_at`
- `updated_at`
- `current_turn_player_id`
- `state_json` (JSONB) ✅ recomendado para velocidad de desarrollo
- (opcional) `version` (int) para control de concurrencia

Tabla `game_players`

- `game_id`
- `user_id`
- `seat`
- `color`
- `joined_at`

> V1 recomendada: guardar **todo el GameState como JSONB** en `games.state_json` (más rápido).
> V2: normalizar si hace falta.

---

8) Lógica del juego (servidor autoritativo)

8.1 Reglas base (MVP)

1. Inicializar tablero con 4 fichas centrales.
2. Calcular `legalMoves` para el jugador del turno.
3. `MAKE_MOVE`:
    - validar: es su turno, celda vacía, movimiento legal
    - aplicar: colocar ficha, voltear líneas capturadas
    - recalcular `scores`, `legalMoves`
    - pasar turno al siguiente jugador con jugadas; si ninguno tiene jugadas → fin o pass automático según regla.
4. `PASS`:
    - solo si `legalMoves` vacío.

8.2 Casillas “?” y habilidades (cuando llegue)

- Al crear partida: generar aleatoriamente algunas celdas `special: "MYSTERY"`.
- Al ocupar una `MYSTERY`:
    - asignar habilidad aleatoria a ese jugador (añadir a `abilitiesByPlayerId[playerId]`)
    - la casilla deja de ser “mystery” (o no, según regla final).

8.3 Turno (mover O habilidad)

En vuestro diseño: “en un turno puedes mover o usar habilidad”.
Backend debe validar:

- si acción es `USE_ABILITY`, **consume turno** (y la habilidad desaparece)
- si `MAKE_MOVE`, consume turno normal

---

9) Contrato de “compatibilidad con clientes”

Para que web/móvil no sufran:

- **Siempre devolver `GameState` completo** tras `join/get/actions`.
- Mantener nombres de campos estables.
- Si se añade un campo nuevo: opcional (los clientes lo ignoran).

---

10) DevOps mínimo (Docker + Compose)

Dado que el despliegue está pensado con contenedores

12. Hu Qiheng Propuesta técnica…

:

- `docker-compose.yml` con:
  - `api` (FastAPI)
  - `db` (PostgreSQL)
- Variables:
  - `DATABASE_URL`
  - `JWT_SECRET`
  - `ENV=dev|prod`

---

11) Testing recomendado (para no romper reglas)

- Tests unitarios del motor de reversi:
  - jugadas válidas en posiciones conocidas
  - volteos correctos
  - fin de partida
- Tests API básicos:
  - create/join
  - una secuencia de 2–3 jugadas

---

12) Qué necesita el frontend web para conectar (prioridad)

Orden recomendado de implementación backend (para desbloquear web):

1. `POST /games` create
2. `POST /games/{id}/join`
3. `GET /games/{id}`
4. `POST /games/{id}/actions` con `MAKE_MOVE`
5. auth básica (puede ser mock al principio)
6. persistencia en `state_json`

---

13) Entregable de backend para coordinación

- Publicar en repo:
  - `docs/openapi.json` o FastAPI swagger accesible
  - este contrato como `docs/BACKEND_CONTRATO_Y_GUIA_V1.md`