**解釋 server 端**

**1. 最上層：main.py – 啟動與配線中心**

**1-1. 做的事情**

main.py 主要負責三件事：

main

1. **決定要綁定的 IP 和 public IP**
    - 從 config.json 讀出 public_host / public_hosts，用 pick_public_ip_from_list() 先找出「這台機器真的擁有」的 IP。
    - 如果找不到，就用外網服務 / 本地 UDP trick 偵測 IP，最後不行就用 127.0.0.1。
2. **動態選出兩個 port**
    - developer_port（給 DevServer）
    - lobby_port（給 LobbyServer）
    - 用 _pick_free_port() 在 10000–65535 亂數試 bind，確保沒有衝突。
3. **同時跑兩個 server + 寫出 runtime_ports.json**
    - 把 developer_port、lobby_port、以及對應的 dev_host / lobby_host 寫進 server/runtime_ports.json，讓 client 端（開發者 / 玩家）可以知道要連去哪裡。
    - 用 asyncio.gather(run_dev_server, run_lobby_server)，實際上是把同步的 serve_dev_sync() / serve_lobby_sync() 丟到 thread pool 裡跑。
    - 用 stop_event = threading.Event() 讓 Ctrl+C 時可以叫兩邊 server 乖乖收攤。

你可以把 main.py 想成「總控」：分配 port、決定對外 IP、啟動 Dev / Lobby 兩顆核心。

---

**2. 共用基礎設施：db.py & auth.py**

**2-1. common/db.py – 簡單 JSON 資料庫**

- 所有 persistent 資料都放在 data/ 目錄底下，以一個檔名 = 一份 JSON 檔的方式管理。

db

- load(name)：讀 data/name，失敗就回傳預設值。
- save(name, obj)：寫回 JSON，indent=2，有加 lock 防止多 thread 同時寫壞檔案。

常用的檔案有：

- dev_users.json：開發者帳號。

dev_server

- player_users.json：玩家帳號 + played 紀錄。

lobby_server

- games.json：遊戲的 metadata（作者、各版本 manifest、zip…）。
- rooms.json：房間 / 對局資訊。

lobby_server

---

**2-2. common/auth.py – in-memory token session**

- 用 issue_token(user, role) 發 token，只允許 **同一個 (user, role)** 同時只登入一個地方。

auth

- SESSIONS: token -> {user, role, ts}
- USER_ACTIVE: (role, user) -> token，所以 player A 不能在兩台機器同時登入「player 角色」。
- verify_token(token, role)：檢查 token 是否存在 & role 是否符合。
- revoke_token(token)：登出時清掉 session。

token 全部都在記憶體，不會存到檔案，所以重開 server 所有登入狀態都會消失。

---

**3. Dev Server：dev_server.py – 給開發者用的後台**

這顆 server 專門給「遊戲開發者」處理帳號與上架遊戲。

dev_server

**3-1. 路徑與資料**

- UPLOADED_DIR = server/uploaded_games：實際存遊戲 zip 解壓後的地方（每個遊戲一個資料夾）。
- dev_users.json：開發者帳號密碼。
- games.json：遊戲的 metadata：
- "my_game": {
- "name": "my_game",
- "author": "devA",
- "status": "active",
- "versions": {
- "1.0.0": {
- "manifest": {...},
- "zip_b64": "..." // 原始上傳 zip（base64）
- }
- },
- "latest": "1.0.0",
- "reviews": { "player1": {...} },

- "avg_rating": 4.5,
- "review_count": 3
- }

**3-2. 主要 API（kind）**

所有請求都是 **單行 JSON + \n**，依 kind 分派：

dev_server

- register：開發者註冊（帳號存在就拒絕）。
- login：開發者登入 → 發 developer role token。
- logout：revoke token。
- upload_game：
    1. 驗證 token 是開發者。
    2. 檢查 version 格式：major.minor.patch。
    3. 若遊戲已存在：檢查作者是不是自己，status 若不是 active 會被改回 active；新版本必須 **嚴格大於** current latest。
    4. 解出 zip 到 uploaded_games/<name>/<version>/。
    5. 更新 games.json 的版本資訊、latest 指向這個新版本。
- remove_game：把 status 改成 "removed"，不會真的刪檔案。
- my_games：回開發者自己所有遊戲的精簡資訊（版本列表、avg_rating、review_count、玩家的文字評論）。
- version_hint：查詢這個遊戲目前的 latest 以及推薦下一個版本號（patch+1）。

**3-3. DevServer 的定位**

- **只跟「開發者 client」互動**（例如 developer_client.py）。
- **負責檔案管理 + 版本管理 + 遊戲 metadata**。
- 玩家端完全不會直接連到 DevServer。

---

**4. Lobby Server：lobby_server.py – 玩家大廳 / 房間管理**

這是**玩家入口**，處理註冊登入、遊戲列表、下載、房間建立、配對、評分等。

lobby_server

**4-1. 路徑與全域設定**

- PUBLIC_HOST：經過 pick_public_host() 選出這台機器對外的 IP（會比對 public_hosts 哪個真的是本機）。
- UPLOADED = server/uploaded_games：和 DevServer 共用的遊戲檔案位置。
- 使用同一份 games.json，但由 Lobby 端用 _scan_uploaded_games() 讀檔案系統，確保真的有檔案。

**4-2. 玩家帳號與登入**

- PLAYER_USERS_FILE = "player_users.json"：玩家帳號 + played 次數。

lobby_server

- handle_register / handle_login / handle_logout：邏輯跟 DevServer 類似，但 role="player"。

**4-3. 遊戲清單與細節**

- handle_list_games：
    1. 先確認玩家有登入。
    2. _scan_uploaded_games() 掃 uploaded_games/ 底下有哪些遊戲資料夾 + 有 manifest.json 的版本。
    3. 只顯示 games.json 中 status = active 的遊戲，版本取「檔案系統 + DB 交集」。
    4. 回傳 display_name, avg_rating, review_count 等。

lobby_server

- handle_game_details：回傳某遊戲完整資訊（各版本 manifest、評分等），但不給 zip。
- handle_download_game：根據 games.json 的 latest 找對應版本的 zip_b64 + manifest 給玩家下載。

---

**4-4. 房間 / 對局管理**

房間資訊統一存在 rooms.json：

lobby_server

```
"game-1731231234-1234": {
  "game": "tetris",
  "version": "1.0.0",
  "host": "140.113.17.xx",     // 玩家要連線的 IP（PUBLIC_HOST）
  "port": 12345,
  "status": "waiting" | "in_game" | "closed",
  "owner": "playerA",
  "start": {...},                  // 開局提議狀態
  "players": ["playerA", "playerB"],
  "ready_players": [],
  "max_players": 2,
  "pid": 123456                        // 遊戲子程序 PID
}
```

主要 API：

- list_rooms：列出所有房間。
- create_room：這是邏輯最重的一段：

lobby_server

1. 驗證 player token。

2. 檢查遊戲名稱存在 + status == "active"。
3. 從 games.json 的 latest 取得標準化版本號（normalize_version），確保「只允許最新版本開房」。
4. 檢查 uploaded_games/game/該版本 真的存在且有 manifest.json。
5. 用 _find_free_port() 找一個 10000+ 的 port。
6. 準備環境變數，包含：
    - GAME_HOST="0.0.0.0"（遊戲 server 綁定）
    - GAME_PORT
    - ROOM_ID, GAME_NAME, GAME_VERSION
    - LOBBY_HOST, LOBBY_CONNECT_HOST, LOBBY_PORT（讓遊戲 server 可以回報遊戲結束、或做其他 RPC）
7. subprocess.Popen([python, entry_server.py], cwd=game_root, env=env) 啟動實際遊戲 server。
8. 反覆嘗試連線 127.0.0.1:port，最多 10 秒，確認遊戲 server 真正開好。
9. 成功後才把房間資訊寫進 rooms.json，status="waiting"，players=[owner]。

- join_room / leave_room：
    - 檢查房間存在、人數上限、有無在 players 裡。
    - 離開時若房主走了，會把 owner 移交給第一個剩下的人，start 狀態重設。
    - 若 status == "in_game" 又有人離開，按設計會把狀態改回 waiting + start={"state":"idle"}。
- player_ready / player_unready：標記 ready 狀態，全部玩家 ready 時，房間狀態改為 "ready"。
- propose_start / respond_start：開始遊戲的「同意機制」：

lobby_server

- 只有房主可以 propose_start，人數必須達到 max_players。
- 房客一個個 respond_start(accept=True)，全部房客都同意後：
    - status="in_game"
    - ready_players=[]
    - 呼叫 _mark_played(game_name, players)，在玩家資料裡記錄「玩過這個遊戲」(之後才能評分)。
- game_finished：由遊戲 server 反向呼叫 Lobby 告知某局結束。
    - 如果 kick_all=True：標記房間 closed，broadcast，稍等一下再把房間從 rooms.json 刪掉。
    - 否則只是把房間狀態 reset 成 waiting，start={"state": "idle"}，

ready_players=[]。

lobby_server

---

**4-5. 房間存活監控（處理你 Ctrl+C 遊戲 server 的情境）**

- room_liveness_monitor_loop：Lobby 啟動時會開一條背景 thread，每 2 秒掃一次所有 rooms。

lobby_server

- 對每個房間，拿 host & port 做 is_room_alive()：
  - 先試房間記錄的 host（通常是 PUBLIC_HOST），再試 127.0.0.1。
  - 如果都連不到 → 判定房間 server 死了（像你在遊戲端 Ctrl+C）。
  - 直接把這個房間從 rooms.json 刪掉。

這個機制就是專門處理「遊戲子程序被 Ctrl+C 或 crash 掉」時，大廳能自動把房間清掉。

---

**4-6. SSE 風格的房間更新（長連線）**

- subscribe_room / unsubscribe_room + room_subscribers：維護「有哪些 TCP 連線正在訂閱某個房間」。

lobby_server

- handle_subscribe_room：
  - 驗證玩家登入 & 房間存在。
  - 把這條 conn 丟進 room_subscribers[room_id]。
  - 先送一次目前房間狀態給 client。
  - 然後直接進一個 while True: time.sleep(10) 的保活 loop，不關這條連線。
- broadcast_room_update(room_id)：
  - 每當房間狀態改變（ready、join/leave、start 等），就把最新房間 JSON 丟給所有訂閱者。

前端只要開一條「訂閱房間」的長連線，就可以即時更新 UI。

---

**4-7. 評分系統**

- handle_rate_game：只有玩過的玩家（player_users.json 的 played[game_name] > 0）才能評分。

lobby_server

- games[name].reviews 統一用 dict：{ user: {rating, text, ts} }。
- 每次評分會重算 avg_rating & review_count，並寫回 games.json。

DevServer 在 my_games 也會把這些評論讀出來給開發者看。

---

**5. utils.py – 公網 IP 偵測工具（目前比較像輔助）**

server/utils.py 裡有一個 get_public_ip() + get_server_host()，可以自動偵測公網 IP 或從 config 決定。

util

現在 main.py 有自己的一套 IP 偵測，所以 utils.py 比較像是「獨立可重用的工具」，未必每個地方都有直接用到。

---

**6. 總結成一句話的架構圖**

- **main**：選 IP & port → 開 DevServer + LobbyServer → 寫 runtime_ports。
- **DevServer**：
  - 服務開發者（註冊 / 登入 / 上傳遊戲 / 版本管理 / 下架 / 看自己遊戲 + 評分）。
  - 管 games.json（遊戲 metadata）與 uploaded_games/。
- **LobbyServer**：
  - 服務玩家（註冊 / 登入 / 列表 / 下載 / 房間 / 評分）。
  - 念 games.json + uploaded_games/，確保只顯示有檔案、active 的遊戲。
  - 為每個房間開一個「遊戲 server 子程序」，寫 rooms.json。
  - 用 room liveness monitor 清理「掛掉的房間」。
  - 用 SSE 風格的訂閱機制即時 broadcast 房間更新。
- **db + auth**：
  - db = 檔案型 JSON DB。
  - auth = in-memory token，限制同帳號只能單點登入（每種角色各一條）。

---

---

幫我解釋「從 developer_client / player_client 一直到 game server 的完整流程」

**0. 整個系統有哪些「Server」？**

**(1) main.py：總控台 / 啟動器**

- 啟動時會：
  1. 讀 config.json
  2. 自動偵測一個「對外用 IP」（public_ip）
  3. 幫 **DeveloperServer** 和 **LobbyServer** 各找一個 **空的 port**
  4. 把這些資訊寫進 server/runtime_ports.json：
  5. {
  6. "developer_port": 5xxxx,

7. "lobby_port": 6xxxx,
8. "dev_host": "你的對外 IP",
9. "lobby_host": "你的對外 IP"
10. }
11. 分別用 thread 跑：
    - dev_server.serve(host_dev, dev_port, stop_event)
    - lobby_server.serve(host_lobby, lobby_port, stop_event)

=> **developer_client / player_client 啟動時，就是來連這兩個 port。**

---

**(2) dev_server.py：開發者專用後端**

- 專門給 **developer_client** 用：
  - 註冊 / 登入 開發者帳號
  - 上傳／更新遊戲 (upload_game)
  - 刪除遊戲 (remove_game)
  - 查自己的遊戲 (my_games)
- 資料儲存在：
  - data/dev_users.json：開發者帳號
  - data/games.json：**全系統遊戲清單 / 版本 / 下架狀態等**
- 遊戲實際檔案放在：
  - server/uploaded_games/<game_name>/<version>/
    （某一版遊戲的完整資料夾）

---

**(3) lobby_server.py：玩家商城 + 大廳 + 房間管理**

- 給 **player_client** 用：
  - 註冊 / 登入 玩家帳號
  - 列出所有遊戲 (list_games)
  - 查看某遊戲詳細資訊 (game_details)
  - 下載遊戲 (download_game)
  - 建立 / 加入 / 離開房間 (create_room / join_room / leave_room)
  - 準備／取消準備、投票開始遊戲 (player_ready / propose_start / respond_start)
  - 評分 / 留言 (rate_game)
- 共享的資料：
  - 同一個 data/games.json（跟 dev_server 共用）
  - data/player_users.json：玩家帳號 & played 記錄
  - data/rooms.json：目前所有房間狀態
- 並且會：
  - 去掃 server/uploaded_games 確認遊戲實際檔案存在

- 當玩家「建立房間」時，**spawn 一個 game server 子行程**

---

**(4) common.auth：登入 / Token 管理**

- 記在記憶體裡（不是檔案）：
  - SESSIONS: token → { user, role, ts }
  - USER_ACTIVE: (role, user) → token
- 角色分兩種：
  - "developer"
  - "player"
- verify_token(token, role=...) 會檢查：
  - token 是否存在
  - role 是否對
  - token 是否還沒被 revoke

---

**(5) common.db：超小型 JSON DB**

- 所有「檔案型 DB」（data/*.json）都透過這裡讀寫：
  - load(name, default)
  - save(name, obj)
- 幫你上鎖，避免 multi-thread 同時寫爆。

---

**1. Developer 端：從 developer_client 到 server 的流程**

**1-1. developer_client 怎麼找到 DevServer？**

1. main.py 啟動後，把 dev_port / dev_host 寫進 server/runtime_ports.json。
2. developer_client.py 啟動時：
   - 先讀 config.json or runtime_ports.json
   - 再看有沒有環境變數覆蓋，例如：
     - DEV_CONNECT_HOST
     - DEV_CONNECT_PORT
3. 決定最後要連的 (host, port)。

---

**1-2. 註冊 / 登入 開發者帳號**

1. 註冊：
   - developer_client 傳一包 TCP + 一行 JSON（結尾 \n）：
   - {"kind": "register", "user": "...", "password": "..."}
   - dev_server._handle_conn 收到後：
     - 用 handle_register() 檢查有沒有重複帳號
     - 存到 data/dev_users.json

        - 回傳 { "ok": true } 或錯誤訊息
2. 登入：
    - developer_client：
    - {"kind": "login", "user": "...", "password": "..."}
    - handle_login():
        - 檢查密碼
        - 呼叫 auth.issue_token(user, role="developer")
        - 回傳：
        - { "ok": true, "token": "abc123...", ... }
    - developer_client 把 token 存到 tokens.json，之後所有 request 都會帶上 token。

---

**1-3. 上傳 / 更新遊戲 (upload_game)**

假設你在本機有一個遊戲資料夾，例如 games/tetris/：

1. **developer_client 做的事：**
    - 讀 manifest.json（包含 name, display_name, type, max_players, entry_server, entry_client 等）
    - 把整個遊戲資料夾壓縮成 zip → base64 字串 zip_b64
    - 準備請求：
    - {
    - "kind": "upload_game",
    - "token": "...",
    - "name": "tetris",
    - "version": "1.0.0",
    - "manifest": {...},
    - "zip_b64": "...."
    - }
    - 傳給 DevServer。
2. **DevServer 做的事 (handle_upload_game)：**
    - 用 auth.verify_token(token, role="developer") 確認這是登入的開發者
    - 檢查版本字串 major.minor.patch 格式
    - 從 data/games.json 讀出目前這個遊戲的版本資訊，確認：
        - 新版本一定要「比最新的大」 (version_greater)
    - 把 base64 解碼 → unzip 到：
    - server/uploaded_games/<name>/<version>/
    - 更新 data/games.json：
    - {

- "tetris": {
- "author": "dev_name",
- "status": "active",
- "latest": "1.0.0",
- "versions": {
- "1.0.0": {
- "manifest": {...},
- "uploaded_at": ...
- }
- }
- },
- ...
- }
- 回傳 { "ok": true, "next_version_hint": "1.0.1" } 之類。

**到這一步為止：**

- 伺服器已經有 games.json 裡的 metadata
- 也有 uploaded_games 裡的實體遊戲檔案
  → **Lobby / Player 就可以看見 & 下載這個遊戲了。**

---

**2. Player 端：從 player_client 到 LobbyServer 的流程**

**2-1. player_client 怎麼找到 LobbyServer？**

跟 dev 類似：

1. main.py 寫 runtime_ports.json：
2. {
3. "lobby_host": "你的對外 IP",
4. "lobby_port": 61234
5. }
6. player_client.py 啟動時：
   - 讀 config.json / runtime_ports.json
   - 再看環境變數：
     - LOBBY_CONNECT_HOST, LOBBY_CONNECT_PORT
   - 形成 (host, port) 去連 LobbyServer。

---

**2-2. 註冊 / 登入 玩家帳號**

流程跟 developer 幾乎一樣，只是 role 不同：

- 註冊：
- {"kind": "register", "user": "...", "password": "..."}

→ lobby_server.handle_register()

→ 寫入 data/player_users.json

- 登入：
- {"kind": "login", "user": "...", "password": "..."}

→ handle_login() → auth.issue_token(role="player")

→ 回傳 token，player_client 存在自己的 tokens.json

---

**2-3. 遊戲商城：看遊戲 / 下載遊戲**

1. 列出所有遊戲 **(list_games)**
    - player_client：
    - {"kind": "list_games", "token": "..."}
    - handle_list_games()：
        - 讀 data/games.json
        - 每個遊戲會附帶 latest version、評分平均、是否下架等資訊
    - 回傳一個列表給前端 UI 顯示。
2. 看某遊戲詳細資訊 **(game_details)**
    - 會把該遊戲所有版本、每版 manifest、留言／評分整理好丟回去。
3. 下載遊戲 **(download_game)**
    - player_client：
    - {
    - "kind": "download_game",
    - "token": "...",
    - "game": "tetris",
    - "version": "1.0.0" // 通常是 latest
    - }
    - handle_download_game()：
        - 去 server/uploaded_games/tetris/1.0.0/ 把資料夾壓縮成 zip
        - 回傳 zip_b64 + manifest 等資訊
    - player_client 在本機解壓到自己的 developer/games/tetris/... 或 player/games/...。

到這邊為止：**玩家本機已經有遊戲 client 端程式可以跑。**

---

**3. 建房 → 開一個 game server → 玩家連上去**

重點來了，從 Lobby → game server 的完整路徑。

**3-1. Player 在 Lobby 建房 (create_room)**

1. player_client 送：

2. {
3. "kind": "create_room",
4. "token": "...",
5. "game": "tetris",
6. "version": "（可選）"
7. }
8. handle_create_room() 流程：
    1. 用 auth.verify_token(token, role="player") 找出 session_user
    2. 確認這個遊戲存在於：
        - 檔案系統：掃 server/uploaded_games → _scan_uploaded_games()
        - DB：data/games.json 裡 status == "active"
    3. 找出 DB 裡的 latest，并且：
        - 強制要求「開房一定用最新版本」
        - 若 payload 指定其它版本 → 直接拒絕
    4. 在檔案系統對應到實際資料夾（可能有版本字串正規化）
    5. game_root = UPLOADED / req_game / actual_folder
    6. manifest_path = game_root / "manifest.json"
    7. entry = manifest["entry_server"] (預設 "start_server.py")
    8. max_players = manifest["max_players"]
    9. **選一個 game server 用來監聽的 port**：
    10. port = _find_free_port()
    11. room_id = f"{req_game}-{int(time.time())}-{random.randint(1000, 9999)}"
    12. 準備兩個重要的 host：
    13. server_bind_host = "0.0.0.0" # game server 綁定用
    14. client_connect_host = PUBLIC_HOST # player client 要用這個連
    15. 建 env，傳給遊戲的 start_server.py：
        - LOBBY_HOST, LOBBY_PORT：讓 game server 知道要打回哪個 Lobby
        - LOBBY_CONNECT_HOST, LOBBY_CONNECT_PORT：同上／或實際用來建立 TCP 回連的 IP/port
        - ROOM_ID, GAME_NAME, GAME_VERSION
        - 還有前面挑好的 SERVER_PORT（你在 code 裡有傳）
    16. subprocess.Popen([...], cwd=game_root, env=env, ...)
        → **真正的「game server」程式啟動**（例如 tetris 的房間伺服器）
    17. Lobby 這邊會用一個小 loop 反覆嘗試連 127.0.0.1:port：

- 成功表示 game server 已經開始 listen
- 若 10 秒內都連不上 → 判定啟動失敗 → kill 子行程，回傳錯誤

18. **確認 game server OK 之後，才把房間寫入 DB：**
    - data/rooms.json：
    - {
    - "tetris-1733840000-1234": {
    - "game": "tetris",
    - "version": "1.0.0",
    - "host": "PUBLIC_HOST（給玩家用）",
    - "port": 54321,
    - "status": "waiting",
    - "owner": "某玩家",
    - "start": {"state": "idle"},
    - "players": ["owner"],
    - "ready_players": [],
    - "max_players": 2,
    - "pid": <game server 的 pid>
    - }
    - }
19. 呼叫 broadcast_room_update(room_id)：透過 SSE 把最新房間資訊推給訂閱者。

9. 回傳給 player_client：
10. {
11. "ok": true,
12. "room_id": "tetris-1733840000-1234",
13. "host": "PUBLIC_HOST",
14. "port": 54321,
15. ...
16. }

---

**3-2. 其他玩家加入房間 (join_room)**

1. player_client 送：
2. {
3. "kind": "join_room",
4. "token": "...",
5. "room_id": "tetris-1733840000-1234"
6. }

7. handle_join_room()：
    - 驗 token / role
    - 找 rooms[room_id]
    - 檢查 status == "waiting"、人數沒有滿
    - 把這個玩家加進 players，存回 rooms.json
    - broadcast_room_update(room_id)
    - 回傳房間完整資訊（含 host, port, players 等）

**此時 player_client 端就知道：要連線遊戲伺服器 = room.host : room.port。**

---

**3-3. 玩家真正連到 game server**

接下來就跟 Lobby 無關，是「遊戲自己的協定」：

- CLI 遊戲（RPS） 或 GUI 遊戲（Tetris）都會在 client 端有一段 code：
    - 拿到 host, port
    - 開 socket 連上去，做遊戲自己定義的 handshake / 傳輸

例如（概念上）：

```
s = socket.socket()
s.connect((room_host, room_port))
# 接下來就是 game 協定：login_room / sync_state / send_input / receive_update...
```

game server 那邊會：

- 用 LOBBY_HOST / LOBBY_PORT 之類的環境變數，**在遊戲開始／結束時打回 Lobby**：
    - propose_start / respond_start：協調開始遊戲
    - game_finished：回報誰贏誰輸，讓 Lobby 更新 played 次數、評分解鎖等等

lobby_server.py 裡面可以看到：

- handle_propose_start
- handle_respond_start
- handle_game_finished

都是專門給 game server 或 client 透過 Lobby 來改動房間狀態的。

---

**3-4. 房間存活監控 & 自動清理**

在 lobby_server.serve() 裡有啟動：

- start_room_liveness_monitor(stop_event)

這個背景 thread 大概會：

1. 定期讀 rooms.json
2. 針對每個 room：
    - 用 pid 看 game server 進程還在不在，或

- 用 is_room_alive(host, port) 試著建立 TCP 連線

3. 如果 game server 掛掉：
    - 把房間標成結束 / 移除
    - broadcast_room_update 通知前端

這樣就算 game server 當掉或你 Ctrl+C 關掉，Lobby 也能慢慢把殘骸清掉。

---

**4. 用一條線串起來（超濃縮流程）**

**開發者路線**

1. developer_client 啟動 → 讀 runtime_ports.json → 連 dev_server
2. register / login → auth.issue_token(role="developer")
3. upload_game (name, version, manifest, zip_b64, token)
4. DevServer：
    - 解壓到 server/uploaded_games/<name>/<version>
    - 更新 data/games.json 的遊戲清單

**玩家路線（玩一場遊戲）**

1. player_client 啟動 → 連 lobby_server
2. register / login → auth.issue_token(role="player")
3. list_games / download_game → 本機就有遊戲程式
4. create_room：
    - Lobby 驗證遊戲存在 & 最新版
    - subprocess.Popen(start_server.py) → 開 game server
    - 寫 rooms.json，廣播房間資訊
5. 其他玩家 join_room：
    - Lobby 更新 players，廣播
6. player_client 根據房間資訊，連線到 **game server 的 host:port**
7. 遊戲完成 → game server 用 game_finished 打回 Lobby
8. Lobby 更新 player_users.json 的 played 記錄，解鎖評分／留言

---

**Player 端（lobby client）：從 player/lobby_client.py 到 game server 的流程**

**1. lobby_client 一開始在做什麼？**

你貼的 lobby_client.py 最前面這些東西是「**決定要連哪個 Lobby / Dev server**」：

```
ROOT = Path(__file__).resolve().parents[1]
CONFIG = json.load(open(ROOT / "config.json", "r", encoding="utf-8"))
runtime_path = ROOT / "server" / "runtime_ports.json"
SERVER_RUNTIME = json.load(open(runtime_path)) if runtime_path.exists() else {}
```

接著 _pick_target() 會依序考慮：

1. **環境變數**

- LOBBY_CONNECT_HOST / LOBBY_CONNECT_PORT
- DEV_CONNECT_HOST / DEV_CONNECT_PORT

2. server/runtime_ports.json 裡 dev/lobby 實際 host/port
3. config.json 裡的 lobby_endpoint, developer_endpoint 預設值
4. 把 "0.0.0.0" 替換成 127.0.0.1 或 public_hosts 裡的外網 IP

最後得到兩個全域常數：

LOBBY_HOST, LOBBY_PORT

DEV_HOST, DEV_PORT

之後所有 send_req() 都是丟到 LOBBY_HOST:LOBBY_PORT。

**2. Ctrl+C 時自動 logout token**

```
def install_sigint_handler(get_lobby_host, get_lobby_port, get_token):
    def handler(sig, frame):
        remote_logout(get_lobby_host(), get_lobby_port(), get_token())
        print("\n[LobbyClient] bye")
        sys.exit(0)
    signal.signal(signal.SIGINT, handler)
```

- 你在 async_main() 裡呼叫：
- install_sigint_handler(lambda: LOBBY_HOST, lambda: LOBBY_PORT, lambda: token)
- 所以玩家在任何階段按 Ctrl+C：
  - client 會送一個 {"kind":"logout", "token":...} 給 lobby_server，
  - server 釋放 token，
  - 然後才結束程式 → 不會留下 zombie session。

**3. 商城流程（從 lobby_client → lobby_server）**

主要用這幾個 API：

- list_games：拿到所有可玩的遊戲 + latest 版本 + 平均評分。
- game_details：某一款遊戲的詳細資訊（作者、status、所有評論…）。
- download_game：請 lobby_server 拿出這款遊戲的 **最新版本壓縮包**。

在 client 端：

1. fetch_playable_games(token) →
   send_req_auth({"kind":"list_games","token":token})
   → 拿到 games dict → print_game_menu() 展示。
2. download_game：
   - 送 request 到 lobby_server。
   - 回傳 { ok: True, version, zip_b64 }。
   - client：
     1. base64.b64decode(zip_b64)。
     2. 解壓縮到 player/downloads/<player>/<game>/<version>/。

3. 清掉舊版子資料夾，確保只留一個最新版。

**重點：**

這裡用的是 lobby_server 的 handle_download_game()，而 download 的來源是 dev_server 在上傳時寫進 games.json / uploaded_games 的資料。

**4. 建立房間 → 啟動遊戲 server**

**(1) 玩家在「大廳 → 建立房間」**

lobby_client 做的事：

1. 先 list_games，讓玩家選遊戲。
2. 確認自己有 **本地最新版**：
3. if not has_local_game_version(player, name, latest_ver):
4. print("請先到商城下載最新版")
5. 送 request：
6. {
7. "kind": "create_room",
8. "token": token,
9. "game": name,
10. "version": latest_ver
11. }

**(2) lobby_server.handle_create_room() 做的事（很關鍵）**

1. 驗證 player token。
2. 從 games.json 確認這款遊戲存在，且 db_latest 是目前最新版本。
3. 檢查 server/uploaded_games/<game>/<version>/manifest.json 是否存在。
4. 讀 manifest.json：
    - entry_server: 遊戲 server 的入口（例如 start_server.py）
    - max_players
5. 挑一個可用 port + 決定要 bind 的 host（server_bind_host）。
6. 準備 env：
7. env.update({
8. "GAME_HOST": server_bind_host,
9. "GAME_PORT": str(port),
10. "ROOM_ID": room_id,
11. "GAME_NAME": req_game,
12. "GAME_VERSION": version,
13. "LOBBY_HOST": LOBBY_HOST,
14. "LOBBY_CONNECT_HOST": lobby_connect_host,
15. "LOBBY_CONNECT_PORT": str(LOBBY_PORT),
16. })

17. 用 subprocess.Popen([sys.executable, entry], cwd=game_root, env=env, ...) 啟動 **該房間專屬的遊戲 server**。
18. 每 0.2 秒試著 TCP connect 到 GAME_HOST:GAME_PORT，最多等 10 秒：
    - 成功 → 認為遊戲 server ready。
    - 失敗 → kill subprocess，回報錯誤。
19. Ready 之後才把房間寫入 rooms.json：
20. rooms[room_id] = {
21. "host": server_bind_host,
22. "port": port,
23. "game": req_game,
24. "version": version,
25. "owner": session_user,
26. "players": [session_user],
27. "status": "waiting",
28. "ready_players": [],
29. "max_players": max_players,
30. "pid": proc.pid,
31. }
32. db.save(ROOMS_FILE, rooms)
33. 回傳給 client：
34. {
35. "ok": true,
36. "room_id": "...",
37. "host": "...",
38. "port": ...,
39. ...
40. }

lobby_client 收到後會呼叫 room_interface(token, player, room_id, resp) → 進入 AsyncRoomUI。

**5. Room UI & SSE：AsyncRoomUI 怎麼跑？**

**(1) 訂閱房間更新**

self.reader, self.writer = await asyncio.open_connection(LOBBY_HOST, LOBBY_PORT)
line = json.dumps({"kind": "subscribe_room", "token": token, "room_id": room_id}) + "\n"

- lobby_server 的 _handle_conn() 看到 kind == "subscribe_room"：
    - 呼叫 handle_subscribe_room(req, conn)。
    - subscribe_room(room_id, conn) 把這條連線加入該房間的訂閱列

表。
- 回傳一次「目前房間狀態」給你。
- **然後不關連線**，保持這條 socket 用來推 room_update。

AsyncRoomUI.update_loop() 會一直：

```
while self.running:
    data = await self.reader.readline()
    msg = json.loads(data)
    if msg["event"] == "room_update":
        self.room_info = msg["room"]
        self.display()
        # 檢查 start.state 是否變成 "agreed" → 自動 start_game()
```

**(2) 玩家按鍵指令**

AsyncRoomUI.handle_input() 讀使用者輸入：

- r → player_ready / player_unready
- s → 房主提議開始 → propose_start
- y / n → 房客回覆 → respond_start
- q → leave_room

這些都走 send_req_auth() → 一次性 request/response（不是走 SSE）。

**(3) 所有人都同意開始 → 自動啟動 game client**

- lobby_server 在 handle_respond_start() 內：
  - 當所有 guests 都 accept=True 時：
  - r["start"] = {"state": "agreed", "by": owner, ...}
  - r["status"] = "in_game"
  - r["ready_players"] = []
  - db.save(...)
  - broadcast_room_update(room_id)
- AsyncRoomUI.update_loop() 收到新的 room_update，start.state == "agreed"：
  - should_auto_start(prev_state, curr_state) 會回 True。
  - 呼叫 start_game()。

---

**六、start_game()：從房間 UI 跳到實際 game client**

```
client_dir = get_local_client_dir(self.player, self.join_info["game"],
self.join_info["version"])
manifest = json.load(open(client_dir / "manifest.json"))
entry = manifest.get("entry_client", "start_client.py")
```

1. 確認玩家本地有下載這款遊戲的正確版本。（如果沒有，要求去商城下載）

2. 讀 manifest 拿到 entry_client。
3. 準備環境變數：
4. env.update({
5. "GAME_HOST": self.join_info["host"],
6. "GAME_PORT": str(self.join_info["port"]),
7. "ROOM_ID": self.room_id,
8. "GAME_NAME": self.join_info["game"],
9. "GAME_VERSION": self.join_info["version"],
10. "PLAYER_USERNAME": self.player,
11. "PLAYER_NAME": self.player
12. })
13. 起一個新 process：
14. subprocess.Popen(
15. [sys.executable, entry],
16. cwd=str(client_dir),
17. env=env,
18. creationflags=... # Windows 開新 console
19. )

**也就是：**

房間 UI 只負責「大家 Ready → 同意開始 → 用 env 告訴遊戲 client 要連哪裡」。

真正的遊戲邏輯在 start_client.py / start_server.py 裡面互相通訊。

在遊戲 server 裡，它會用 GAME_HOST / GAME_PORT 去 bind socket，

用 LOBBY_HOST / LOBBY_CONNECT_HOST / LOBBY_CONNECT_PORT 回呼 Lobby（例如 game_finished）。

---

## 七、房間結束 / Game Server 中止 時發生什麼？

**1. 遊戲正常結束**

- Game server（start_server.py）會在一局結束時，送給 lobby_server：
- {"kind": "game_finished", "room_id": "...", "kick_all": true/false, ...}
- lobby_server.handle_game_finished()：
    - 如果 kick_all=True → 把房間刪掉，或把 players 清空。
    - 或把 status 改回 waiting（可以再開下一局）。
    - broadcast_room_update → 房間 UI 看到房間關閉 / reset。

**2. Game server 被 Ctrl+C 殺掉**

- lobby_server 有一個 room_liveness_monitor_loop()：
- while True:
- rooms = db.load(ROOMS_FILE, {})

- for rid, r in rooms:
- if not is_room_alive(r["host"], r["port"]):
- rooms.pop(rid)
- db.save(rooms)
- time.sleep(2)
- 也就是每 2 秒 ping 一次各個房間的 game server，如果連不上就認定房間死掉 → 刪除 room。
- AsyncRoomUI.display() 每次收到 room_update / 或之後再 list_rooms 會發現房間不見，
  會顯示「房間已關閉，按 Enter 返回大廳」。

---

小結：一句話串起整個流程

- **開發者路線**：
  developer_client → dev_server → 把遊戲 zip ＋版本資訊寫進 server/uploaded_games + games.json。
- **玩家路線（商城）**：
  lobby_client → lobby_server 的 list_games / download_game → 把 zip 抓回自己 player/downloads。
- **玩家路線（大廳 / 房間 / 遊戲）**：
  lobby_client（登入）→
  create_room / join_room →
  lobby_server 開啟 game server subprocess + 建立 room →
  AsyncRoomUI.subscribe_room 建立 SSE 連線 →
  Ready / start 換來換去 →
  start.state="agreed" → AsyncRoomUI.start_game()
  → 起 start_client.py，用 env 連到該房間的 **game server**
  → 遊戲結束時 game server 用 game_finished 通知 lobby，更新 room 狀態或關閉房間。

---

---

token 存放不是放 list，是放在「兩個全域的 dict 裡」。
現在這套架構是這樣：

auth.py 裡有這幾個全域變數：

```
# token -> { "user": "abc", "role": "player", "ts": 12345 }
SESSIONS = {}

# (role, user) -> token
USER_ACTIVE = {}
```

所以：

- **SESSIONS 是 dict**：
  key = token 字串
  value = { "user": 使用者帳號, "role": "player"/"developer", "ts": 登入時間 }
- **USER_ACTIVE 也是 dict**：
  key = (role, user) 這個 tuple
  value = 該使用者目前那組 token
  → 用來保證「同一個帳號同一個角色只會有一個 active token」。

登入時（在 dev_server.py / lobby_server.py）會呼叫像：

token = auth.issue_token(u, role="player")

issue_token() 裡（雖然程式被省略，但可以確定邏輯）大概就是：

1. 產生 UUID 當 token。
2. 在 SESSIONS[token] = {...} 記錄 user / role / 時間。
3. 在 USER_ACTIVE[(role, user)] = token 做反向索引（方便之後踢掉舊登入）。

登出或你手動 revoke 時，會走到你檔案裡的這段：

```
def revoke_token(token: str | None):
    if not token:
        return

    with _LOCK:
        info = SESSIONS.pop(token, None)
        if not info:
            return

        key = (info["role"], info["user"])
        if USER_ACTIVE.get(key) == token:
            USER_ACTIVE.pop(key, None)
```

也就是說：

- 從 SESSIONS 把這個 token 刪掉。
- 如果 USER_ACTIVE[(role, user)] 指向的正好是這個 token，就也一起刪掉。

把上線的人變成 server 的全域變數，不要存到 database

- token 完全沒有透過 db.save() 寫進任何 JSON 檔。
- 都只在 auth.py 這兩個 dict 裡記憶體存著（算是「全域變數」）。
- 只要 server 重啟，SESSIONS / USER_ACTIVE 清掉 → 所有 token 自動失效。

所以總結：**Server**：token 是存在 auth.SESSIONS / auth.USER_ACTIVE 這兩個 **dict**，不是 list，而且不寫入資料庫。

---

user case: D1

**1. 目標 & 前置條件在程式裡的對應**

**目標：**

讓開發者把自己在本機寫好的遊戲，透過 developer_client.py → dev_server.py 上架成一個「有版本管理」的遊戲，最後會被寫進 data/games.json，並且把實際的遊戲檔案解壓到 server/uploaded_games/<name>/<version>/，之後 Lobby / Player 就可以看到。

**前置條件在程式裡是：**

1. 開發者已完成遊戲
   → 在檔案系統上要有這個資料夾：
2. developer/games/<遊戲名稱>/
3. ├– manifest.json
4. ├– start_server.py
5. ├– start_client.py
6. └ 其他遊戲檔案...
7. Developer Client 能連線到 Developer Server
   → developer_client.py 會透過 _pick_dev_target() 決定 DEV_HOST, DEV_PORT，再用 asyncio.open_connection(DEV_HOST, DEV_PORT) 去連 dev_server.py。
8. 開發者已登入
   → 在 async_main() 裡第一層是「登入選單」，選 2) login 成功後，server 會發一個 token 回來，client 把它存在 token / CURRENT_TOKEN 裡，之後所有 request 都會帶 {"token": token} 給 server。

---

**2. Developer Client 端的實際流程（對應 D1 步驟）**

**Step 1：啟動 Developer Client 並登入**

- 使用者執行：
- python developer/developer_client.py
- async_main() 先跑「登入選單」：
  - 選 1) register → 發 {"kind":"register", "username":..., "password":...} 給 dev_server
  - 選 2) login → 發 {"kind":"login", "username":..., "password":...} dev_server 驗證成功後，用 auth.issue_token(role="developer", user=...) 產生一個 token，回傳給 client。
- client 收到後：

- if resp.get("ok"):
- token = resp["token"]
- CURRENT_TOKEN = token
- developer = u

此時「前置條件：已登入」就成立。

---

**Step 2：開發者選擇「上架新遊戲」**

登入後進入「開發者主選單」：

=== 開發者主選單 ===

1) 上傳/更新遊戲
2) 查看我的遊戲
3) 下架遊戲
4) 登出
5) 離開

- 選 **1) 上傳/更新遊戲**，進入你的 Use Case D1 的主流程。

---

**Step 3：輸入遊戲名稱 → 拿 version_hint（詢問 server 狀態）**

在 choice == "1" 這段：

1. 先問開發者「遊戲名稱」：
2. game_name = input("遊戲名稱: ").strip()
3. 發一個 version_hint 給 dev_server：
4. hint = await send_req_auth({
5. "kind": "version_hint",
6. "token": token,
7. "name": game_name
8. })
9. dev_server.handle_version_hint() 會：
    - 讀 data/games.json
    - 看這個 name 有沒有存在
    - 回傳：
        - 如果是新遊戲：{"exists": False, "suggested": "1.0.0"}
        - 如果已存在：{"exists": True, "latest": <目前最新>, "suggested": <建議下一個版本>, "versions": [...]}
10. client 根據回傳決定要印什麼提示，例如：
    - 新遊戲：顯示「建議初始版本號：1.0.0」
    - 舊遊戲：顯示目前最新版本、建議下一版、以及已有版本列表。

這一步就對應 spec 裡的「系統引導開發者輸入必要資訊（版本、是否新遊戲等）」。

---

**Step 4：讀取本地遊戲資料夾 & manifest（相當於 config）**

接著 developer_client.py 會：

1. 找遊戲資料夾：
2. game_dir = GAMES_ROOT / game_name  # developer/games/<game_name>
3. manifest_path = game_dir / "manifest.json"
4. 檢查資料夾有沒有存在、manifest.json 有沒有存在，沒有就直接：
5. print("❌ 找不到遊戲資料夾")
6. 或
7. print("❌ 遊戲資料夾缺少 manifest.json")

→ 對應 spec 的：「若選取的遊戲路徑不存在或檔案無法讀取，要顯示錯誤訊息」。

8. 讀 manifest.json，並檢查必要欄位 REQUIRED_MANIFEST_KEYS：
9. REQUIRED_MANIFEST_KEYS = [
10. "name",
11. "display_name",
12. "type",
13. "max_players",
14. "entry_server",
15. "entry_client",
16. "description",
17. ]

缺任何一個，就：

print("❌ manifest.json 缺少以下重要欄位：", ", ".join(missing_keys))

→ 這就是 spec 裡提到的「config 檔缺少必要欄位時，要提示哪一項缺少，並拒絕上架」。

18. 如果 manifest["name"] != game_name，會給一個「⚠ 警告」，提醒盡量保持名稱一致。

這部分就是你實作的「設定檔檢查」，用 manifest.json 扮演 spec 裡的 config 檔角色：

- 遊戲名稱 / 顯示名稱
- 遊戲類型（CLI / GUI）
- 人數上限
- 啟動 server/client 的 entry point

---

**Step 5：把遊戲整個壓縮成 ZIP，轉成 base64**

接著 client 會把整個 game_dir 壓縮起來：

buf = io.BytesIO()

```
with zipfile.ZipFile(buf, "w", zipfile.ZIP_DEFLATED) as z:
    for path in game_dir.rglob("*"):
        if path.is_file():
            rel = path.relative_to(game_dir)
            z.write(path, rel.as_posix())
zip_bytes = buf.getvalue()
zip_b64 = base64.b64encode(zip_bytes).decode("utf-8")
```

也就是：

- 走訪整個遊戲資料夾
- 每個檔案都塞進 zip
- zip 的 bytes 用 base64 轉成字串，準備送給 server

這一步對應 spec 的「提交要上傳的遊戲檔案」。

---

**Step 6：決定版本號 → 驗證 → 送給 Server**

client 先決定版本號：

```
while True:
    ver_input = input("版本號（例如 1.0.0；直接 Enter 使用建議值 ...）：
").strip()
    version = suggested if not ver_input else ver_input

    ok_ver, msg_ver = validate_version(version)
```

- validate_version(version) 用 regex VERSION_RE 檢查是否符合 major.minor.patch。
- 不合法 → 印出錯誤訊息，問你要不要重打。

當版本合法之後，正式送 request 給 dev_server：

```
resp = await send_req_auth({
    "kind": "upload_game",
    "token": token,
    "name": game_name,
    "version": version,
    "manifest": manifest,
    "zip_b64": zip_b64
})
```

如果成功：

```
print(f"✓ 上傳成功：{resp.get('name')} 最新版 {resp.get('latest')}
(status={resp.get('status')})")
```

如果失敗（例如版本不夠大），server 會回 latest + suggested，client 會提示你，可以重新輸入版本號再試一次。

---

**3. Dev Server 端的實際處理流程**

**Step A：驗證 token 身分**

在 dev_server.py 的 handle_upload_game(payload) 一開始：

```
token = payload.get("token")
tokinfo = auth.verify_token(token, role="developer")
if not tokinfo:
    return auth_fail()
developer = tokinfo["user"]
```

- 用 auth.verify_token 確認這個 token 是「developer」角色，且有效。
- 驗證失敗就用統一的 auth_fail() 回 { "ok": False, "code": "NOT_LOGGED_IN", ... }。

這對應 Use Case 的「開發者必須是已登入且具開發者身分」。

---

**Step B：檢查基本欄位 & 版本格式**

接著：

```
name = payload.get("name","").strip()
version = payload.get("version","").strip()
manifest = payload.get("manifest", {})
zip_b64 = payload.get("zip_b64","")

if not name or not version or not manifest or not zip_b64:
    return {"ok": False, "error": "缺少必要欄位"}
```

然後再用 parse_version(version) 再驗證一次版本格式，錯就回錯誤和建議值：

```
if not parse_version(version):
    return {
        "ok": False,
        "error": "版本格式錯誤，需為：major.minor.patch（例如 1.0.3）。",
        "suggested": "1.0.0"
    }
```

---

**Step C：讀取 / 更新 games.json（遊戲清單 DB）**

1. 讀 DB：
2. games = db.load(GAMES_FILE, {})
3. 如果是新遊戲，就建立骨架：
4. if name not in games:
5. games[name] = {
6. "author": developer,
7. "created": now_ts,

8. "updated": now_ts,
9. "status": "active",
10. "versions": {}
11. }
12. else:
13. # 已存在 → 檢查作者是否是同一個人、版本是否有變大
14. 已存在的遊戲會檢查：
    - author 是否等於現在登入的 developer，不是就拒絕（避免 A 把 B 的遊戲搶走）。
    - 新的 version 必須比 game["latest"] **嚴格大**：
    - if not version_greater(version, current_latest):
    - suggested = suggest_next_version(current_latest)
    - return {
    - "ok": False,
    - "error": f"目前最新版本為 {current_latest}，新的版本號必須大於目前版本。",
    - "latest": current_latest,
    - "suggested": suggested
    - }

這就是「版本管理政策」。

---

**Step D：解壓 ZIP 到 server/uploaded_games（實際檔案部署）**

handle_upload_game 會呼叫 _extract_upload(name, version, zip_b64)：

```
dst = UPLOADED_DIR / name / version    #
server/uploaded_games/<name>/<version>
if dst.exists():
    shutil.rmtree(dst)
dst.mkdir(parents=True, exist_ok=True)

with zipfile.ZipFile(io.BytesIO(raw), "r") as z:
    z.extractall(dst)
```

- 先 base64.b64decode
- 再用 ZipFile 解到指定資料夾
- 如果解壓失敗，回傳錯誤訊息

這對應 Use Case 的「與 Server 端互動，完成檔案上傳」。

---

**Step E：更新 DB → 回應成功**

如果解壓成功，才會更新 games：

```
game["versions"][version] = {
    "manifest": manifest,
    "zip_b64": zip_b64
}
game["latest"] = version
game["updated"] = now_ts

db.save(GAMES_FILE, games)
```

然後回給 client：

```
return {
    "ok": True,
    "name": name,
    "latest": game["latest"],
    "status": game.get("status", "active")
}
```

Developer Client 收到這個，就印「✓ 上傳成功」。

---

**4. 錯誤處理：你的實作對應到 spec 的哪些點？**

對照題目給的錯誤情境：

1. **必填欄位未填**
    - 在 client：
        - 如果 game_name 空 → 直接提示「遊戲名稱不可空白」。
    - 在 server：
        - name/version/manifest/zip_b64 有任何一個缺 → 回 {ok: False, error: "缺少必要欄位"}。
2. **路徑不存在 / 檔案無法讀取**
    - client 檢查 game_dir.exists()、manifest_path.exists()，錯就印：
    - ✕ 找不到遊戲資料夾
    - ✕ 遊戲資料夾缺少 manifest.json
    - 讀 manifest 失敗也會 catch 例外並顯示錯誤。
3. **config（manifest）缺欄位**
    - 檢查 REQUIRED_MANIFEST_KEYS，有缺就列出缺哪個欄位，強迫你修正後才能上架。
      → 完全符合 spec 那條。
4. **版本號問題**
    - 格式錯誤：validate_version / parse_version 會擋掉。
    - 新版本不比舊版本大：server 回傳目前 latest + 建議版本，client 顯示並問你要不要重試。

5. **Server 端暫時無法連線或上傳失敗**
    - send_req 連線失敗時回：
    - {"ok": False, "error": "無法連線到開發者伺服器（DevServer）。請稍後再試。"}
    - zip 解析 / 解壓失敗時，server 會回「zip base64 解析失敗」或「zip 解壓失敗」的明確錯誤字串。

---

**5. 預期結果 & 後續效果**

上架成功後：

1. data/games.json 會多出（或更新）這個遊戲：
    - 包含：
        - author（開發者帳號）
        - latest 最新版本
        - versions 裡每個 version 的 manifest + zip_b64
        - 之後還會有評分、review 之類欄位（給玩家端用）
2. server/uploaded_games/<name>/<version>/ 會有完整遊戲檔案。
   未來 Lobby / Room 在開房、啟動遊戲 server 時就會從這裡載入。
3. 在 Developer Client 選 **2) 查看我的遊戲** 時，會呼叫 my_games：
   dev_server.handle_my_games() 會把你是 author 的所有遊戲整理成列表，顯示：
    - 每個遊戲的 狀態、最新版本
    - 各版本的 display_name, type, max_players
    - 玩家評分 / 評論（之後由 player 端寫入）
4. 再往後，在 Lobby / Player 的流程裡，lobby_server 會讀同一份 games.json，把這個遊戲變成商城上的一個項目，玩家可以：
    - 在「商城」看到它
    - download_game 把 zip_b64 解壓到 player/downloads/<player>/<game>/<version>/
    - 建房 / 加房 / 啟動對局

user case: D2

以下是 **Use Case D2（開發者更新已上架遊戲版本）** 在你程式碼中的「完整對應流程」，全部根據你現有的 developer_client.py、dev_server.py、games.json 的真實邏輯來說明！

---

**✅ Use Case D2：開發者更新已上架遊戲版本（程式實作流程）**

下面我只講「程式裡實際發生什麼」，讓你可以寫在報告裡或當文件用。

---

**★ 1. 目標（程式對應）**

在你的程式裡：

- **開發者不需要新遊戲條目，只要上傳新版本 → Server 自動替遊戲新增一個版本號，並更新 latest。**
- 新版本的檔案會被放到：

server/uploaded_games/<game_name>/<new_version>/

- 版本清單、作者、最新版本資訊全寫入：

data/games.json

---

**★ 2. 前置條件（程式對應）**

程式在以下情況下才允許更新：

**✓（1）遊戲必須已存在於 games.json**

因為 developer_client.py 會先使用：

{"kind": "version_hint", "name": game_name}

如果是既有遊戲 → Server 回傳：

{"exists": true, "latest": "...", "versions": [...]}

**✓（2）開發者必須登入 Developer Client**

登入後會取得 token 作為身分識別。

**✓（3）此遊戲必須由該開發者所建立**

在 dev_server.py：

```
if game["author"] != developer:
    return {"ok": False, "error": "你不是這款遊戲的作者，無法更新。"}
```

---

**★ 3. 實際流程 Step-by-Step（程式對應）**

---

**● Step 1：開發者啟動 Developer Client 並登入**

成功登入後進入：

=== 開發者主選單 ===

1) 上傳/更新遊戲

2) 查看我的遊戲

3) 下架遊戲
4) 登出
5) 離開

---

● **Step 2：系統列出開發者所有已上架的遊戲（選單 2）**
choice == "2" 時會呼叫：
send_req_auth({"kind": "my_games", "token": token})
Server 回傳：

```
{
   "games": {
      "tetris": {
          "author": "devA",
          "latest": "1.2.0",
          "versions": {...},
          "avg_rating": ...,
          "reviews": ...
      },
      ...
   }
}
```

---

● **Step 3：開發者選擇要更新的遊戲（在選單 1 進行）**
選單 1：「上傳 / 更新遊戲」：
game_name = input("遊戲名稱: ")
接著送出：
{"kind": "version_hint", "name": game_name}
如果這是既有遊戲，Server 回傳：

```
{
   "exists": true,
   "latest": "1.2.0",
   "suggested": "1.2.1",
   "versions": ["1.0.0", "1.1.0", "1.2.0"]
}
```

→ **這一步就是確認「可更新的遊戲」與作者是否一致。**

---

● **Step 4：輸入新版本號 & 檔案來源（本地遊戲資料夾）**
Developer Client 要求：

- 版本號

- 使用者本機遊戲資料夾的位置（固定為 developer/games/<game_name>/）
- 讀取 manifest.json
- 壓縮整個遊戲資料夾（ZIP + base64）

版本驗證：

validate_version(version)

如果格式錯誤 → 重新輸入。

若版本 <= 最新版本 → Server 會回：

```
{
  "ok": false,
  "error": "新的版本號必須大於目前版本",
  "latest": "現有版本",
  "suggested": "建議新版本"
}
```

---

**● Step 5：開發者確認更新 → 送資料給 Server**

確認後 Developer Client 送出：

```
{
  "kind": "upload_game",
  "token": "...",
  "name": "tetris",
  "version": "1.2.1",
  "manifest": {...},
  "zip_b64": "...."
}
```

---

**● Step 6：Server 處理更新**

**Server 端會做：**

1. **驗證 token**
2. **檢查遊戲存在**
3. **確認作者一致**
4. **檢查版本合法**（必須大於 latest）
5. **解壓 ZIP → 存到：**

server/uploaded_games/tetris/1.2.1/

6. **更新 games.json：**

```
"tetris": {
  "latest": "1.2.1",
  "versions": {
```

```
        "1.2.1": { manifest... },
        "1.2.0": {...},
        ...
    }
}
```

最後回應：

```
{
    "ok": true,
    "name": "tetris",
    "latest": "1.2.1",
    "status": "active"
}
```

---

**★ 4. 錯誤處理（程式對應）**

以下完全是你程式碼裡已實作的錯誤情境：

---

**❌（1）不是作者卻想更新**

dev_server.py：

```
if game["author"] != developer:
        return {"ok": False, "error": "你不是這款遊戲的作者，無法更新。"}
```

---

**❌（2）本地遊戲資料夾 / manifest.json 不存在**

Developer Client：

```
if not game_dir.exists():
        print("❌ 找不到遊戲資料夾")
if not manifest_path.exists():
        print("❌ 遊戲資料夾缺少 manifest.json")
```

---

**❌（3）版本號格式錯誤或沒比舊版本大**

Developer Client：

```
validate_version(version)
```

Server：

```
if not version_greater(version, latest):
        return {"ok": false, "error": "...", "suggested": "..."}
```

---

**❌（4）ZIP 壓縮或解壓失敗**

Server：

```
except zipfile.BadZipFile:
```

return {"ok": False, "error": "zip 解壓失敗"}

---

❌（5）**Server 連線中斷**

Developer Client：

return {"ok": false, "error": "開發者伺服器回應異常"}

---

⭐ **5. 預期結果（程式對應）**

✓（1）**games.json 中該遊戲的 latest 更新為新版本**

Lobby / Player 端會自動以最新版本為主。

✓（2）**舊版本仍保留在 versions 列表**

玩家若下載，會被強制使用最新版本。

✓（3）**Lobby 建房時永遠使用最新版本**

因為 Lobby 的 create_room 會讀 latest。

---

🪡 **最後整理一句話**

**在系統裡，開發者更新遊戲版本的核心流程就是：Developer Client 壓縮遊戲 → 上傳 zip_b64 + manifest + 新版本號 → Dev Server 驗證並寫入 games.json → 更新 uploaded_games → 完成。**

## ⭑ Use Case D3：開發者下架一款遊戲（程式面的實作流程）

---

### 🎯 1. 目標（在程式中的對應）

在你的系統裡：

- 下架遊戲並不會刪除版本檔案
- 只是把遊戲的 **status** 設為 **"removed"**
- 玩家端與 Lobby 都會看這個欄位來決定：
  - 不再讓玩家看到這款遊戲
  - 無法建立新房間
  - 無法下載

Server 上的 games.json 會變成：

```
"tetris": {
    "status": "removed",
    "latest": "...",
    "versions": {...},
    "author": "devA"
}
```

---

### ⭑ 2. 前置條件（程式中的判斷）

Developer Server 在處理 remove_game 時會檢查：

**✓（1）遊戲存在**

若找不到 → 回傳：

```
{"ok": false, "error": "此遊戲不存在"}
```

**✓（2）開發者是作者**

否則回：

```
{"ok": false, "error": "你不是這款遊戲的作者，無法下架"}
```

---

### ⭑ 3. 程式的實際流程 Step-by-Step

---

**● Step 1：Developer Client 登入 → 進入主選單**

Developer 進入：

```
=== 開發者主選單 ===
3) 下架遊戲
```

---

**● Step 2：開發者選擇「下架遊戲」**

Developer Client 執行：

```
game_name = input("要下架的遊戲名稱: ")
```

並送出：

```
{
   "kind": "remove_game",
   "token": token,
   "name": game_name
}
```

---

**● Step 3：Server 收到 remove_game 請求**

dev_server.py 會：

1. 驗證 token
2. 在 games.json 找到該遊戲
3. 確認 author == developer
4. 更新欄位：

```
game["status"] = "removed"
```

5. 寫回 JSON
6. 回傳成功訊息：

```
{"ok": true, "name": "tetris", "status": "removed"}
```

---

**● Step 4：Developer Client 顯示成功訊息**

Client 列印：

```
{'ok': True, 'name': 'tetris', 'status': 'removed'}
```

---

**⋆ 4. 錯誤處理（在程式中的真實邏輯）**

---

**✗（1）不是作者 → Server 拒絕**

Server：

```
{"ok": false, "error": "你不是這款遊戲的作者，無法下架"}
```

---

**✗（2）遊戲不存在**

```
{"ok": false, "error": "此遊戲不存在"}
```

---

**✗（3）Server 連線失敗**

Client：

```
{"ok": False, "error": "開發者伺服器回應異常或已中斷連線。"}
```

---

**✗（4）遊戲已經下架**

雖然你程式目前 **允許重複下架**（只是再寫一次 "removed"），

但是回傳還是：

{"ok": true}，但基本上重複下架好像也不會造成甚麼影響

---

**⋆ 5. 預期結果（程式中實際效果）**

**✓（1）玩家端看不到這款遊戲**

因為 Player Server 與 Lobby 都會判斷：

```
if game["status"] != "active":
    不讓玩家下載或建房
```

**✓（2）無法建立新房間**

Lobby Server 會拒絕建立房間。

**✓（3）已下載的玩家仍能遊玩（依你目前程式設計）**

系統**不會強制禁止舊玩家啟動該遊戲**，

但無法建立新房間，因此基本上已經不能正常使用。

---

**🗡 最後總結一句話**

**在系統中，下架遊戲就是 Developer Client 發出 remove_game → Developer Server 確認作者 → 將遊戲 status 改成 "removed" → 玩家端與 Lobby 之後都看不到該遊戲，也無法建立新房間。**

**Use Case P1：玩家瀏覽商城（程式流程）**

---

**🎯 1. 目標（程式對應）**

在你的系統裡：

- 玩家透過 **Lobby Client** 向 **Lobby Server** 取得遊戲列表。
- 遊戲資訊來源是 server/data/games.json。
- **只有 status == "active" 的遊戲才會被顯示在商城中。**

---

**⭐ 2. 前置條件（程式中的判斷）**

玩家端（Lobby Client）要進入商城前必須：

1. 成功登入 → 拿到 token。
2. Lobby Server 驗證 token → 使用 auth.SESSIONS。
3. 至少有一款遊戲 status="active"。

---

**⭐ 3. 程式中的完整操作流程 Step-by-Step**

---

**● Step 1：玩家啟動 Lobby Client → 登入**

登入成功後，玩家進入：

=== 玩家主選單 ===

1) 商城

2) 我的遊戲

3) 建立房間

4) 加入房間

...

---

**● Step 2：玩家選擇「商城」→ Client 傳送 request**

Lobby Client 送出：

{"kind": "list_games", "token": "<player_token>"}

---

**● Step 3：Lobby Server 回傳可用遊戲列表**

Lobby Server 讀取 games.json，並過濾只有 **active** 的遊戲：

```
for name, game in games.items():
    if game["status"] != "active":
        continue
```

回傳格式類似：

```
{
  "ok": true,
  "games": {
```

```
      "tetris": {
         "latest": "1.2.1",
         "display_name": "Tetris (GUI)",
         "type": "GUI",
         "max_players": 2,
         "description": "Two-player tetris ...",
         "avg_rating": 4.5,
         "review_count": 12
      },
      "rps": {
         "latest": "1.0.0",
         "type": "CLI",
         "description": "Rock-paper-scissors game",
         ...
      }
   }
}
```


如果沒有遊戲：


```
{"ok": true, "games": {}}
```


---

**● Step 4：玩家選擇其中一款遊戲 → Client 發送詳細資訊請求**

Lobby Client 送出：


```
{"kind": "game_info", "token": "<player_token>", "name": "tetris"}
```


---

**● Step 5：Lobby Server 回傳詳細資訊**

Server 會取出：

- 最新版本的 manifest
- 作者
- 遊戲介紹
- 評分 & 評論

資料格式：


```
{
   "ok": true,
   "name": "tetris",
   "display_name": "Tetris (GUI)",
   "type": "GUI",
   "max_players": 2,
   "description": "Two-player tetris ...",
```

```
    "author": "devA",
    "latest": "1.2.1",
    "avg_rating": 4.5,
    "reviews": {
        "player1": { "rating": 5, "text": "好玩！" },
        "player2": { "rating": 4, "text": "" }
    }
}
```


---

● **Step 6：玩家可以選擇：**

- 返回列表
- 下載遊戲
- 建立房間
- 或離開商城

---

**⋆ 4. 錯誤處理（程式中的真實邏輯）**

---

**✗（1）沒有任何 active 遊戲**

Server 回傳空列表：


{"ok": true, "games": {}}


Client 顯示：

目前沒有可遊玩的遊戲

---

**✗（2）Server 連線失敗**

Lobby Client：

列表載入失敗，請稍後再試。

---

**✗（3）玩家查詢不存在的遊戲或已下架的遊戲**

Server 回：


{"ok": false, "error": "找不到此遊戲"}


或：


{"ok": false, "error": "此遊戲已被下架"}


---

**✗（4）遊戲缺少 fields（如 description）**

Client 會顯示預設內容：

- 沒簡介 → 顯示「尚未提供簡介」
- 沒評論 → 顯示「尚無評論」

不會 crash。

---

**⋆ 5. 預期結果（程式實際效果）**

✓ 玩家可以看到所有 **active** 的遊戲（名稱、類型、版本、描述）

✓ 玩家能查看每款遊戲的詳細資訊

✓ 若想下載、更新、建立房間，會進入下一個 Use Case

✓ 玩家不需要知道後端細節，就能清楚理解遊戲內容

## ⭐ Use Case P2：玩家下載並更新遊戲版本（程式中的流程）

---

### 🎯 1. 程式中的目標

你的系統在 P2 的目的非常清楚：

- 玩家按一個選項，就能下載某款遊戲的 **最新版 ZIP**。
- 若玩家已經有舊版 → 自動更新到最新版。
- 所有檔案都從 Lobby Server 拿 → 由 Server 保證版本一致性。

---

### ⭐ 2. 前置條件（程式對應）

玩家端下載前會檢查：

**✓（1）該遊戲需為 active**

Lobby Server 僅提供：

```
if game["status"] != "active": skip
```

**✓（2）玩家端本地有下載資料夾**

例如：

```
player/games/<game_name>/
```

若該資料夾不存在，Client 會自動建立。

---

### ⭐ 3. 真實的程式流程 Step-by-Step

---

**● Step 1：玩家在 Lobby Client 選擇「下載/更新遊戲」**

玩家進入選單：

3) 下載 / 更新遊戲

Client 送出：

```
{"kind": "list_games", "token": "<player_token>"}
```

→ Lobby Server 回傳所有 active 的遊戲清單。

---

**● Step 2：玩家選擇要下載的遊戲**

Client：

```
{"kind": "game_info", "name": game_name}
```

Server 回傳：

```
{
   "name": "tetris",
   "latest": "1.2.1",
   "display_name": "...",
   "type": "...",
   "max_players": 2,
   ...
```

}

---

**● Step 3：判斷「首次下載」或「更新」**

Player Client 檢查：

player/games/tetris/

- 若不存在 → 視為首次下載
- 若存在 → 檢查本地版本，例如：

player/games/tetris/version.txt

若：

- 本地版本 < 最新版本 → 需要更新
- 本地版本 == 最新版本 → 顯示「已是最新版本」

---

**● Step 4：玩家確認下載 / 更新**

Player Client 送出下載請求：

```
{
  "kind": "download_game",
  "token": "<player_token>",
  "name": "tetris",
  "version": "1.2.1"
}
```

---

**● Step 5：Lobby Server 進行檔案傳輸**

Lobby Server 會：

1. 讀取 Server 端遊戲檔案：

server/uploaded_games/tetris/1.2.1/

2. 壓縮成 zip
3. 轉成 base64
4. 回傳：

```
{
  "ok": true,
  "zip_b64": "<base64-encoded-zip>",
  "version": "1.2.1"
}
```

---

**● Step 6：Player Client 下載並解壓縮 ZIP**

Client 會：

1. 解析 base64
2. 將 zip 寫入：

player/games/tetris/

3. 解壓縮出完整遊戲檔案：
    - start_client.py
    - start_server.py
    - logic_*.py
    - assets（若有）
4. 寫入版本資訊：

player/games/tetris/version.txt

內容：

1.2.1

最後顯示：

遊戲 tetris 已成功下載（或更新到 1.2.1）

---

**★ 4. 錯誤處理（程式上的邏輯）**

---

**❌（1）下載過程中連線失敗**

Client 顯示：

下載失敗：伺服器中斷

並 **不會覆蓋舊版本**（避免玩家得到半套檔案）。

---

**❌（2）zip 解壓失敗**

Client 顯示：

解壓縮失敗

並刪除未完成的檔案，確保狀態乾淨。

---

**❌（3）伺服器回報遊戲不存在或已下架**

Server 回：

{"ok": false, "error": "此遊戲已被下架"}

Client 顯示：

此遊戲目前不可下載

---

**❌（4）版本不存在**

玩家指定的版本不存在時 Server 回：

{"ok": false, "error": "版本不存在"}

Client 不會下載。

---

**★ 5. 預期結果（程式的真正效果）**

✓ 玩家能一鍵下載最新版本

✓ 若已有舊版本 → 自動更新
✓ 本地存放結構清楚且安全：
player/games/<name>/<files>
version.txt
✓ 不會出現半下載、下載失敗後檔案錯亂的情況
✓ 玩家永遠清楚自己的版本是否是最新的

**Use Case P3：玩家建立房間並啟動遊戲**
**——在你程式碼中的真實流程解釋**

---

**🎯 1. 目標（在程式裡的實現）**
在你的架構裡，玩家建立房間的流程其實是：

1. 玩家在 Lobby Client 選好遊戲
2. Lobby Server 幫他開一個房間（並幫房間分配 port）
3. Lobby Server 產生該房間對應的 *game server*（獨立子行程）
4. 玩家自動啟動自己的 *game client*
5. 遊戲 client 連線到 game server → 正式進入遊戲對局

你的程式已完整實作上述全部功能。

---

**⭐ 2. 真實流程（程式逐步對應）**

---

**● Step 1：選擇「建立房間」**
玩家在 Lobby Client 主選單選：
2) 大廳 → 1) 建立房間
→ Player Client 會先取得最新「可玩的遊戲列表」
await fetch_playable_games(token)

---

**● Step 2：列出可用遊戲後，玩家選擇遊戲**
選擇遊戲後，Lobby Client 檢查玩家是否已下載最新版本：

```
if not has_local_game_version(player, name, latest_ver):
    print("❌ 請先下載最新版遊戲")
```

---

**● Step 3：玩家具有最新版本 → 發出建立房間請求**
Client 送出：

```
{
  "kind": "create_room",
  "token": "<player_token>",
  "game": "<game_name>",
  "version": "<latest>"
}
```

---

**● Step 4：Lobby Server 創建房間（程式中的重要機制）**
Lobby Server 做了以下事情：
**✓ 1) 建立 room_id（如 tetris-ab12cd）**
**✓ 2) 找一個可用 port（例如 60123）**

✓ **3) 啟動遊戲 server（子行程）**

以 Tetris 為例，Server 會執行：

python developer/games/tetris/start_server.py --room=xxxx

並且將 room_id 對應的資訊存進 rooms.json。

🚀 **這代表：**

每個房間都有「自己的 game server」。

---

● **Step 5：Lobby Server 回傳房間資訊給玩家**

內容包括：

```
{
   "ok": true,
   "room_id": "tetris-1234",
   "host": "<game_server_host>",
   "port": <game_server_port>,
   "game": "tetris",
   "version": "1.2.1"
}
```

Player Client 使用這些資料進入房間 UI：

await room_interface(token, player, room_id, resp)

---

● **Step 6：進入 SSE 房間 UI**

在 AsyncRoomUI.run()：

- 開啟 SSE（長連線）訂閱房間狀態
- 玩家可 ready / unready
- 房主可 propose_start
- 若所有人都同意 → 自動啟動遊戲

---

● **Step 7：遊戲啟動（最關鍵）**

當房主與所有玩家同意「開始對局」後：

await self.start_game()

裡面做了以下動作：

---

✓ **（1）取得本地遊戲目錄**

client_dir = get_local_client_dir(player, game, version)

---

✓ **（2）找出 entry_client（來自 manifest.json）**

例如：

"entry_client": "start_client.py"

---

✓（3）用 **subprocess** 啟動遊戲 **client**

Windows：

```
subprocess.Popen(
    [sys.executable, entry],
    cwd=str(client_dir),
    creationflags=subprocess.CREATE_NEW_CONSOLE
)
```

Mac/Linux：

```
subprocess.Popen([sys.executable, entry], cwd=str(client_dir))
```

---

✓（4）透過環境變數傳遞 **game server** 的位址

非常重要！

Lobby Client 向遊戲 client 傳入：

```
env = {
    "GAME_HOST": self.join_info["host"],
    "GAME_PORT": self.join_info["port"],
    "ROOM_ID": room_id,
    ...
}
```

→ 遊戲 **client** 讀取這些環境變數

→ 連線到 **game server**

→ 完成最後一步「進入遊戲」

---

⋆ **3.** 錯誤處理（程式裡已經做到）

---

✗ 遊戲已下架（無法建立房間）

Lobby Server 檢查：

```
if game["status"] != "active":
    return {"ok": False, "error": "遊戲已下架"}
```

---

✗ 遊戲版本不符

Player Client：

```
if room_ver != latest_ver:
    print("⚠ 此房間使用舊版本，請先更新")
```

---

✗ **game server** 未成功啟動

Player Client 啟動 game client 時若發生：

- port 不通

- server 未啟動
- game client 啟動失敗

玩家仍留在房間 UI，不會卡死，並顯示：

❌ 無法啟動遊戲

---

**⭐ 4. 預期結果（對應程式運作）**

✓ **玩家一鍵完成「建立房間 → 等待 → 進入遊戲」**

✓ **大廳可以即時看到新房間（SSE room_update）**

✓ **玩家不需要理解：**

- game server 如何啟動
- port 綁定
- process 管理
- 多玩家同步
- 文件版本管理

這些全部在後端自動完成。

**Use Case P4：玩家對遊戲進行評分與留言**

---

**🎯 1. 玩家進到「我的紀錄 → 評分與評論」**

在 **player/lobby_client.py** 主選單：

3) 我的紀錄 → 評分與評論

程式會：

1. 呼叫 fetch_playable_games(token)
2. 列出所有目前「平台仍可見」的遊戲
3. 玩家選擇一個遊戲編號
4. 開始詢問評分與評論內容

---

**⋆ 2. 程式對應 Use Case 的步驟**

---

**● Step 1：玩家選擇「評分與評論」**

主選單：

```
elif choice == "3":
    print("=== 我的紀錄 → 評分與評論 ===")
```

---

**● Step 2：列出可評分的遊戲**

```
games = await fetch_playable_games(token)
items = print_game_menu(games)
```

玩家看到：

- 遊戲名稱
- 作者
- 最新版本
- 平均評分

---

**● Step 3：玩家輸入評分與評論**

```
rating = input("評分 (1-5): ").strip()
text = input("短評 (可留空): ").strip()
```

這完全符合 Use Case 裡的「玩家輸入分數與文字」。

---

**● Step 4：送出評價至 Lobby Server**

Lobby Client 送出 JSON：

```
resp = await send_req_auth({
    "kind": "rate_game",
    "token": token,
    "name": name,
```

```
    "rating": rating_int,
    "text": text
})
```

Lobby Server 接收到後會：

- 檢查玩家是否登入
- 驗證遊戲是否存在
- 儲存評分與留言在 DB（通常是 JSON）
- 更新 avg_rating 與評論數 review_count

---

**● Step 5：Server 回覆成功後，Client 顯示結果**

玩家會看到：

```
if resp.get("ok"):
    print("✓ 評論已送出")
    avg = resp.get("avg_rating")
    cnt = resp.get("count")
    print(f"目前平均分數：{avg}（{cnt} 則評論）")
```

完全符合 Use Case 裡的：

玩家送出後，系統顯示成功回饋

---

**⋆ 3. 錯誤處理（程式裡已經做到）**

程式完全對應 Use Case 中的錯誤處理！

---

**✗ 1. 評分非 1–5（超出合法範圍）**

```
try:
    rating_int = int(rating)
except:
    print("評分需為數字 1-5")
```

若玩家輸入非法字元或數字會馬上提示。

---

**✗ 2. 玩家未登入 → 自動跳回登入介面**

任何 API 若收到：

```
{ "code": "NOT_LOGGED_IN" }
```

會觸發：

```
raise AuthExpired()
```

然後主流程會：

⚠ 你的登入已失效或被登出，請重新登入。

---

**✗ 3. Server 回報錯誤（如遊戲不存在 / 未玩過）**

else:

print("✗ 無法送出評論：", resp.get("error"))

例如可能出現：

✗ 無法送出評論：玩家未曾遊玩此遊戲

（若你實作了"玩過才能評"）

---

### ✗ 4. 網路錯誤（送出失敗）

send_req 會回傳：

{ "ok": false, "error": "連線錯誤：xxx" }

玩家仍保留輸入內容，不會消失。

---

### ⋆ 4. 預期結果（在你程式中的呈現）

玩家成功留言後：

- Lobby Server 更新 DB：
  - reviews
  - avg_rating
  - count
- Player Client 會顯示新的平均分數
- 開發者在 Developer Client 的「查看我的遊戲」中也會看到玩家評論：

```
for user, rv in reviews.items():
    print(f"{user}: {rv.get('rating')} 分 | {rv.get('text')}")
```

→ 玩家評價會出現在開發者介面中！

**0. 整體連線關係（先有大地圖）**

- developer_client.py ↔ **Developer Server**（dev_server.py）
- lobby_client.py（玩家 Lobby） ↔ **Lobby Server**（lobby_server.py）
- **Lobby Server ↔ Game Server 子行程**（各遊戲的 start_server.py）
- 玩家本機 **Game Client**（各遊戲的 start_client.py） ↔ **Game Server 子行程**
- Developer Server / Lobby Server 彼此不直接 socket 聯絡，但**共用同一套 db.py 資料庫檔案**，所以看到的是同一份遊戲資訊、評分、版本等。

Host / Port 的來源：

- 兩個 client 一開始會用 _pick_target() / _pick_dev_target() 去決定要連到哪個 IP / port
  → 來源包含 config.json、runtime_ports.json、環境變數 *_CONNECT_HOST/PORT。

---

**P1：玩家瀏覽商城 / 詳細資訊（Lobby Client ↔ Lobby Server）**

**1. Client 怎麼打到 Lobby Server**

在 player/lobby_client.py：

- 開頭會算出：

```
LOBBY_HOST, LOBBY_PORT = _pick_target(...)
```

- 所有跟 Lobby 溝通都走：

```
async def send_req(payload):
    reader, writer = await asyncio.open_connection(LOBBY_HOST, LOBBY_PORT)
    line = json.dumps(payload) + "\n"
    writer.write(line.encode("utf-8"))
    ...
    data = await reader.readline()
    return json.loads(data.decode("utf-8"))
```

也就是：

☞ 每次請求（list_games, game_details, …）都開一條 TCP 連線，丟一行 JSON，讀一行 JSON 回來，然後關掉。

send_req_auth() 只是包一層，如果回來是「未登入」就丟 AuthExpired。

**2. 列出遊戲列表（對應「看商城」）**

- fetch_playable_games(token)：

```
resp = await send_req_auth({"kind":"list_games","token":token})
```

- Lobby Server 端 lobby_server.py 有一個大 handler（省略細節）：

```
if kind == "list_games":
    resp = handle_list_games(req)
```

handle_list_games() 用 db.load(...) 把遊戲資料讀出來，整理成：

```
{
   "ok": True,
   "games": {
        "tetris": { "latest": "...", "versions": [...], "avg_rating": ..., ... },
        ...
   }
}
```

- Client 收到後丟給 print_game_menu() 印出列表。

**3. 查看遊戲詳細資訊（對應「點某一款看細節」）**

- 在「商城 → 查看遊戲詳細資訊」那段：

```
resp = await send_req_auth({"kind":"game_details","token":token,"name":name})
```

- Server 端 handle_game_details() 從 DB 把那款遊戲的：
  - 作者、狀態、版本
  - 平均分數 avg_rating
  - 各玩家 reviews

  讀出來包成 JSON 回傳。
- Client 就照著這個 JSON 印出「作者 / 評分 / 評論列表」。

**重點：P1 完全是「LobbyClient ⇄ LobbyServer 單次 TCP + JSON」在跑，不牽涉 Game Server。**

---

**P2：玩家下載 / 更新遊戲版本（Lobby Client ↔ Lobby Server + 本地檔案）**

**1. 連線部分**

流程前段跟 P1 一樣，先用 fetch_playable_games() 列表，選一個遊戲後：

```
resp = await send_req_auth({
      "kind": "download_game",
      "token": token,
      "name": name
})
```

這一包一樣是：

- send_req_auth() → send_req() → asyncio.open_connection(LOBBY_HOST, LOBBY_PORT)
- Lobby Server 收到 kind=download_game，進到 handle_download_game()。

**2. Server 端怎麼把遊戲檔給玩家**

在 lobby_server.py 裡（邏輯）：

- 到 server/uploaded_games/<game>/<version>/ 把該版本整個資料夾 zip 起來。
- base64.b64encode(zip_bytes) 變成字串塞進 JSON 回應：

return {
  "ok": True,
  "version": latest,
  "zip_b64": "..."  # zip 的 base64
}

**3. Client 端怎麼存成本機**

Lobby Client 收到後這段：

version = resp["version"]

zip_b64 = resp["zip_b64"]

data = base64.b64decode(zip_b64.encode("utf-8"))

base_dir = DOWNLOADS_ROOT / player / name

# 先刪掉舊版本

...

dest = base_dir / version

safe_extract_zip(data, dest)

- DOWNLOADS_ROOT = player/downloads
- 最終結構：player/downloads/<player>/<game>/<version>/...
- has_local_game_version() 就是看 .../start_client.py 在不在。

**重點：P2 的「連線」只有一條：LobbyClient ⇄ LobbyServer，拿到 base64 zip；真正的檔案落地、覆蓋舊版全部在 Client 端的檔案系統操作。**

---

**P3：玩家建立房間並啟動遊戲（Lobby Client ↔ Lobby Server ↔ Game Server ↔ Game Client）**

這個是「四方連線」：Lobby Client、Lobby Server、Game Server、Game Client。

**1. 建立房間：Player → Lobby Server**

在 Lobby 主選單 → 大廳 → 建立房間：

resp = await send_req_auth({
    "kind": "create_room",
    "token": token,
    "game": name,
    "version": latest_ver,
})

→ send_req_auth() 用 TCP 和 Lobby Server 單次連線。

Server 端：handle_create_room(payload)：

1. 驗證玩家 token。
2. 確認此遊戲存在、版本是最新。
3. 掃 server/uploaded_games/... 找對應的 entry_server（例如

start_server.py）。

4. 找一個空的 port port = _find_free_port()。
5. 組環境變數 env = {...}：
6. env.update({
7. "GAME_HOST": server_bind_host, # game server 綁定用
8. "GAME_PORT": str(port),
9. "ROOM_ID": room_id,
10. "GAME_NAME": req_game,
11. "GAME_VERSION": version,
12. "LOBBY_HOST": LOBBY_HOST,
13. "LOBBY_CONNECT_HOST": lobby_connect_host,
14. "LOBBY_PORT": str(LOBBY_PORT or 0),
15. })
16. 用 subprocess.Popen([sys.executable, entry], cwd=game_root, env=env, ...) 開一個**遊戲伺服器子行程**。

這一步等於在後台「啟動 Game Server 進程」，它會自己在 GAME_HOST:GAME_PORT 上 socket.listen()。

**2. Lobby Server 確認 Game Server 真的活著**

handle_create_room() 裡有一段 loop：

```
for attempt in range(50):
    try:
        test_sock = socket.socket()
        test_sock.settimeout(0.5)
        test_sock.connect(("127.0.0.1", port))   # 測試連線
        test_sock.close()
        server_ready = True
        break
    except ...:
        time.sleep(0.2)
        if proc.poll() is not None:
            # 子行程掛了
            break
```

- 如果測試成功 → server_ready = True，才會：
  - 把房間資訊寫進 rooms.json（經過 db.save）。
  - 回傳 {"ok": True, "room_id": ..., "host": ..., "port": ...} 給 Lobby Client。

**到這裡為止的連線：**

- Player LobbyClient ⇄ LobbyServer（建立房間請求）

- LobbyServer 啟動 GameServer 進程，GameServer 在自己 port 上聽。

**3. 房間畫面與 SSE 更新：Player <-> Lobby Server（長連線）**

玩家進房間後（await room_interface(...)）：

- AsyncRoomUI.connect_stream()：

```
self.reader, self.writer = await asyncio.open_connection(LOBBY_HOST, LOBBY_PORT)
line = json.dumps({"kind": "subscribe_room", "token":..., "room_id":...}) + "\n"
self.writer.write(...)
```

- 這條連線**不會立刻關掉**，會一直 reader.readline() 等待：

```
async def update_loop(self):
    while self.running:
        data = await self.reader.readline()
        msg = json.loads(data.decode("utf-8"))
        if msg.get("event") == "room_update":
            self.room_info = msg["room"]
            self.display()
            ...
```

Server 端 handle_subscribe_room(payload, conn) 會把這個連線註冊在一個 list 裡面（訂閱者），之後房間有變化就 broadcast_room_update() 把最新 room JSON 丟給所有訂閱此房間的人。

這就是你們自己實作的「簡易 SSE / 推播」機制。

玩家在房間裡按 [r]/[q]/[s]/[y]/[n] 的時候：

- handle_input() 都是叫 send_req_auth({...})
- 也就是再開**另一條短暫連線**，對 Lobby Server 發出：
    - "kind": "player_ready" / "player_unready"
    - "propose_start" / "respond_start"
    - "leave_room"

房間資料被更新後，Lobby Server 再透過 **剛才 subscribe_room 的那條長連線** 推送新狀態給 UI。

**4. 啟動 Game Client：Player 本機 ↔ Game Server**

當所有人都同意開始，AsyncRoomUI.should_auto_start() 觸發 → start_game()：

```
client_dir = get_local_client_dir(player, game, version)
manifest = json.load(client_dir / "manifest.json")
entry = manifest.get("entry_client", "start_client.py")

env = os.environ.copy()
env.update({
    "GAME_HOST": self.join_info["host"],
    "GAME_PORT": str(self.join_info["port"]),
```

```
    "ROOM_ID": self.room_id,
    "GAME_NAME": ...,
    ...
})

subprocess.Popen(
    [sys.executable, entry],
    cwd=str(client_dir),
    env=env,
    ...
)
```

這裡的 host / port 就是一開始 create_room() 回來的 Game Server 位址。
所以真正在玩遊戲時：

- Game Client 用 socket.connect((GAME_HOST, GAME_PORT)) 連到 Game Server。
- Lobby 只負責配對 + 房間狀態，不直接收遊戲封包。

---

**P4：玩家評分與留言（Lobby Client ↔ Lobby Server ↔ 共用 DB ↔ Developer Client）**

**1. 玩家送出評價：Lobby Client → Lobby Server**

在 Lobby Client 主選單 → 選「我的紀錄 → 評分與評論」那段：

```
resp = await send_req_auth({
    "kind": "rate_game",
    "token": token,
    "name": name,
    "rating": rating_int,
    "text": text
})
```

一樣是開 socket → LOBBY_HOST:LOBBY_PORT → 丟一行 JSON。

**2. Server 端儲存評價**

lobby_server.py：

```
def handle_rate_game(payload):
    token = payload.get("token")
    t = auth.verify_token(token, role="player")
    ...
    name = ...
    rating = ...
    text = ...
```

```
games = db.load(GAMES_FILE, {})
g = games.get(name)
...
reviews = g.setdefault("reviews", {})
reviews[user] = {
    "rating": rating,
    "text": text,
    "ts": int(time.time())
}

# 重算 avg_rating / review_count
...

db.save(GAMES_FILE, games)

return {
    "ok": True,
    "msg": "已送出評論/評分",
    "avg_rating": g.get("avg_rating"),
    "count": g.get("review_count")
}
```

也就是：

- 讀 JSON 檔 → 修改該遊戲底下的 reviews[user] → 更新平均分數 → 存回去。
- DB 檔是共享的（common/db.py + 同一個檔名路徑）。

**3. 之後怎麼被看見（玩家 / 開發者）**

**玩家端：**

- P1 的 game_details 有顯示 reviews：
  - lobby_client.py 裡：
  - resp = await send_req_auth({"kind":"game_details",...})
  - reviews = d.get("reviews", {})

→ 這裡就是讀剛剛寫好的 reviews。

**開發者端：**

- developer_client.py 的「查看我的遊戲」：

```
resp = await send_req_auth({"kind": "my_games", "token": token})
...
reviews = info.get("reviews", {}) or {}
```

for user, rv in sorted(reviews.items(), key=lambda kv: _ts(kv[1]), reverse=True):

...

這邊 send_req_auth() 連的是 **Developer Server**（DEV_HOST, DEV_PORT），而 Developer Server 內部一樣用 db.load(GAMES_FILE, ...) 讀同一份遊戲 JSON，因此看得到同樣的 reviews / avg_rating。

所以 P4 的「連線串起來」是：

1. Player LobbyClient → LobbyServer（rate_game）
2. LobbyServer 寫入 DB
3. 之後：
    - Player LobbyClient → LobbyServer（game_details）
    - DeveloperClient → DevServer（my_games）
      兩邊都透過共用 DB 讀到同一份評價資料。

✓1 **server/main.py 啟動後，程式是如何開始運作的？**

你的後端 Server 整體架構通常包含：

- **Developer Server（5501）**
- **Lobby Server（5502）**
- （以及動態 Game Server，由房主啟動）

server/main.py 的作用，就是 **一次啟動這兩個核心 Server**。

---

🔍 **server/main.py 內部流程（啟動順序）**

**流程概述**

1. **載入設定（config + runtime_ports）**
2. **啟動 Developer Server（例如 port 5501）**
3. **啟動 Lobby Server（例如 port 5502）**
4. **兩個 server 以 asyncio 永久監聽，等待 client 連線**
5. **main thread 進入 event loop，不會退出**

**舉例**

```
from dev_server import start_dev_server
from lobby_server import start_lobby_server
import asyncio

async def main():
    await asyncio.gather(
        start_dev_server(),
        start_lobby_server(),
    )

if __name__ == "__main__":
    asyncio.run(main())
```

☞ **說明：**

- asyncio.gather() 會「同時」開兩個 socket 伺服器。
- Developer Server 用來接收 developer_client 的請求（上架 / 更新 / 下架）。
- Lobby Server 用來接收 player_client 的請求（商城 / 房間 / 評分等）。

---

✓2 **Developer Client / Player Client 啟動後是如何連線到 Server 的？**

★ **共同邏輯：**

兩個 Client 都會在檔案載入最開始執行 _pick_target()

用來決定 **我要連線到哪一個 IP:port**。

---

🔧 **Developer Client（developer_client.py）連線流程**

**Client 啟動時（模組 import 時）**

DEV_HOST, DEV_PORT = _pick_dev_target()

這行會：

1. 檢查 DEV_CONNECT_HOST / DEV_CONNECT_PORT 環境變數
2. 若沒有 → 讀取 config.json
3. 若 main.py 寫入 runtime_ports → 讀取動態 port

確定 developer server 的位置，例如：

127.0.0.1:5501

---

**當 client 想發請求時**

例如登入：

resp = await send_req({"kind":"login", "username":u, "password":p})

send_req() 會做：

reader, writer = await asyncio.open_connection(DEV_HOST, DEV_PORT)

writer.write(JSON + "\n")

resp = await _read_json_line(reader)

✓ **這裡正式連線到 Developer Server 的 socket**

---

🎮 **Player Client（lobby_client.py）連線流程**

啟動時同樣會：

LOBBY_HOST, LOBBY_PORT = _pick_target(...)

然後所有玩家操作（登入、商城、房間）都會統一使用：

resp = await send_req({"kind": ...})

send_req() 會：

reader, writer = await asyncio.open_connection(LOBBY_HOST, LOBBY_PORT)

writer.write(JSON + "\n")

resp = await reader.readline()

✓ **這就是 player → lobby server 的 socket 連線**

---

🔥 **3️⃣ 那 Game Server 是怎麼啟動的？（Bonus）**

當玩家在房間中按下「開始」，Lobby Server 會：

1. fork 一個新的 Python process
2. 執行該遊戲 entry_server.py
3. 分配一個動態 port
4. 回傳給所有玩家

Player Client 自動讀取 join_info：

GAME_HOST = join_info["host"]

GAME_PORT = join_info["port"]
並在本地啟動：
subprocess.Popen([python, "start_client.py"], env=vars)
完成連線。

---

🎯 總結（非常簡短）

| 啟動物件 | 程式流程 |
| --- | --- |
| **server/main.py** | 啟動 Developer Server + Lobby Server，掛在 event loop 等連線 |
| **developer_client.py** | _pick_dev_target() 取得 IP → send_req() 用 socket 連線 Developer Server |
| **player/lobby_client.py** | _pick_target() 決定 IP → send_req() socket 連線 Lobby Server |
| **game server / client** | 房間建立後，由 Lobby Server fork 新 process，Client 用 env 變數連線 |

✓ **1. main.py 內部 server 是如何 accept 連線？**

你的 server/main.py 裡，不管是 **Lobby Server** 或 **Developer Server**，它們都是用 **asyncio.start_server()** 建立 TCP 伺服器。

一旦 server 啟動：

server = await asyncio.start_server(handle_client, host, port)

其中：

- **handle_client(reader, writer)** 就是每次有 client 連上來時，被自動呼叫的處理函式
- asyncio 自動處理 accept，不需要你手寫 socket.accept()

也就是：

★ **流程**

1. client 端 open_connection() → 對應 TCP connect
2. asyncio server 背後呼叫 accept()
3. 成功後呼叫你提供的 handle_client()
4. reader.readline() 讀一行 JSON
5. server 根據 kind 分派給不同邏輯處理

➞ **你並沒有手動寫 socket.accept()，全部由 asyncio server 幫你做。**

---

✓ **2. Lobby Server 內部如何管理房間？**

Lobby Server 使用 **rooms.json + 一個 Python dict（記憶體態）** 管理房間資料。

核心結構大概像：

```
rooms = {
    "tetris-1234": {
        "game": "tetris",
        "version": "1.0.0",
        "host": "...",
        "port": 54321,
        "players": ["AAA", "BBB"],
        "ready_players": [],
        "status": "waiting",
        "start": { ... }
    }
}
```

而流程：

★ **建立房間**

kind=create_room

→ server 建立 dict entry

→ 寫回 rooms.json
→ 回傳房間 ID 給 player
★ **加入房間**
kind=join_room
→ 更新 players
→ SSE 推播給所有訂閱者
★ **房間更新**

- ready/unready
- propose_start / respond_start
- 房主同意後啟動遊戲 server

每次更新：

1. 修改記憶體中的房間 dict
2. **寫入 rooms.json**（持久化）
3. **推送 SSE 給訂閱中的玩家**

➡ **房間是存在於 server 記憶體（dict）中，並同步寫回 JSON 檔。**

---

✅ **3. Developer Server 內部如何寫入 DB？**
Developer Server 所有資料都透過：
server/db.py
使用 JSON 當成資料庫。
流程：
★ **上傳遊戲 (kind=upload_game)**

1. client 送 zip + manifest
2. server 在 db["games"] 中新增或更新：

```
db["games"][game_name] = {
    "author": developer_name,
    "versions": { version: manifest },
    "latest": version,
    "status": "active"
}
```

3. 使用 db.save() → 寫回 games.json

★ **下架遊戲**

```
db["games"][name]["status"] = "removed"
db.save()
```

★ **Developer Server 不使用 SQL**
它就只有：

- 一个 Python dict 做 cache
- 一个 JSON 檔做持久化

➞ **Developer Server 更新資料全部靠 db.py，把 dict 寫回 JSON。**

---

**✓ 4. 動態 game server 是怎麼被 fork 出來的？**

當房主 & 房客全部 agree 開始遊戲時：

Lobby Server 做：

subprocess.Popen([python, "start_server.py"], env=new_env)

具體做法：

1. Lobby 產生新的 port（動態 port）
2. 把遊戲 server 所需設定塞進環境變數：
    - ROOM_ID
    - GAME_HOST
    - GAME_PORT
    - GAME_NAME
    - GAME_VERSION
3. 呼叫 start_server.py（位於 developer 上傳的遊戲資料夾）

遊戲 server 就被 **獨立 fork 成一個新 process**，不會跟 Lobby server 共享記憶體。

**★ 動態 game server 實際上是：**

✓ 由 python subprocess 啟動

✓ 每款遊戲都是一個獨立的 server 程序

✓ port 由 Lobby 管理並寫入 runtime_ports.json

**➞ 所有 game server 都是透過 subprocess.Popen fork 出來的獨立程序。**

---

**✓ 5. player 的 SSE（訂閱房間）是怎麼串起來的？**

PlayerClient 的 AsyncRoomUI 在加入房間後會：

```
reader, writer = await asyncio.open_connection(lobby_host, lobby_port)
writer.write(json.dumps({"kind": "subscribe_room", "room_id": ...}))
```

Lobby Server 做的事情：

1. 收到 subscribe_room
2. 把這個玩家的 writer 存入某個 list：

```
subscribers[room_id].append(writer)
```

3. 當房間有任何更新

    → server 廣播給所有 writer：

```
writer.write(json.dumps({"event":"room_update", "room":room_data}))
```

4. Player 端 update_loop() 不斷執行：

```
while True:
    msg = await reader.readline()
    # 收到 room_update 就重新 render UI
```

🔌 **也就是：**

✓ player 每個房間各開一條 TCP 連線

✓ lobby server 做 "pseudo-SSE"：一行一行 JSON 推送

✓ player 持續 read 更新

➞ **SSE 是用 TCP + readline 的自製簡易版 server-push 不是 HTTP SSE，但概念相同。**