ChatGPT

Epsilon-POC-2 Backend Architecture

Overview of Architecture and Design Choices

Epsilon-POC-2 is a backend application built with **FastAPI** (Python) and **SQLAlchemy** as its ORM. It exposes a set of RESTful HTTP endpoints and a WebSocket endpoint to support a multiplayer labyrinth game. The architecture is a single-service (monolithic) backend that handles API requests, real-time messaging, database interactions, and even serves a simple front-end interface. Key design choices include:

- **FastAPI for API & WebSockets**: FastAPI provides high-performance async endpoints and built-in WebSocket support. The application creates a FastAPI app (`app = FastAPI()`) and defines various HTTP routes for game session management and labyrinth generation [1]. It also mounts a WebSocket route at `/ws/{session_id}` for real-time updates [2]. This unified approach allows using the same framework for both REST and real-time features.
- **Relational Database with SQLAlchemy**: The backend uses a PostgreSQL database (connection configured via a `DATABASE_URL`) to persist game data [3]. SQLAlchemy's ORM is used to define models for all entities (game sessions, labyrinths, tiles, players, etc.) and to handle database operations [4] [5]. A single metadata Base is shared across all models for schema creation [6]. Design-wise, this provides an easily queryable, relational structure for game state.
- **Pydantic Models for Data Validation**: Pydantic models (via FastAPI) are used for request/response schemas. For example, `GameSessionCreateRequest` and `SessionStatus` classes define the expected input/output structures [7] [8]. This enforces data integrity at the API boundary and simplifies converting DB objects to JSON (leveraging Pydantic v2's `from_attributes` for ORM models [9]).
- **Real-time Client Updates**: The design includes a lightweight real-time messaging system using WebSockets. Instead of a complex message broker, the server holds connections in memory and broadcasts JSON messages to all clients in a game session when certain events occur (e.g., a new player joins or toggles "ready" status) [10]. This choice keeps the POC simple while still enabling instant client synchronization.
- **In-Memory Game State vs. Database**: Persistent game state (sessions, labyrinth layout, players, etc.) is stored in the database, whereas transient state such as **players' readiness** is tracked in-memory for quick access. A global dictionary `session_readiness` holds each session's player readiness flags, guarded by a thread lock for concurrency [11]. This design avoids frequent DB writes for every ready/unready toggle and showcases a simple concurrency solution (FastAPI's async + a `threading.Lock` to protect shared data [12]).
- **Static Seeding of Data**: On startup, the application can initialize the database with static reference data (game entities, equipment, skills, specials) from CSV files [13] [14]. This choice indicates a design for pre-defined game content that the game logic can reference (e.g., character attributes or items). It simplifies initial setup at the cost of embedding seed data loading in the app.
- **Integrated Front-End Serving**: The backend also serves a static front-end (HTML/JS/CSS) for demonstration. The FastAPI app mounts a `frontend/` directory at the root URL, enabling the app to serve an index page and tile images directly [15]. This design choice (likely for the POC phase)

avoids setting up a separate web server; it allows developers to run the backend and get a basic UI in one go.

Overall, the architecture is a **self-contained web service** that manages game sessions and labyrinth generation. It prioritizes simplicity and clarity over advanced concerns (e.g., there's no formal authentication system or scaling beyond a single process, since this is a proof-of-concept). The design decisions reflect rapid development with ChatGPT's help (per the README) and focus on core functionality: creating game sessions with mazes and coordinating multiple clients in real-time.

Project Structure: Folders and Files Breakdown

The repository's backend code is organized into logical modules and folders, each with specific responsibilities. Below is a detailed breakdown of the codebase structure and how the pieces interact:

- **Application Entry Point**:

- `main.py` – The main FastAPI application definition and setup. It creates the `app`, configures startup events, and mounts routes. In `startup` event, it initializes the database and optionally re-initializes the schema/data [16]. It also registers the WebSocket routes and includes the API router [17]. A few REST endpoints are defined here as well (e.g., listing game sessions, creating a session, generating a labyrinth, deleting sessions) as part of the core functionality [18] [19]. Finally, it mounts static file routes to serve the front-end UI and tile images [15].

- **Configuration**:

- `config.py` – Contains configuration constants and initialization logic. It defines the database connection string (`DATABASE_URL`) for a PostgreSQL database (in this POC, a static URL with credentials is used) and sets up the SQLAlchemy engine [3] [20]. An `init_db()` function is provided to create all tables based on the ORM models [21]. The config also defines an `ASSET_PATHS` dictionary for file locations (e.g., portraits, backstories) to support the game's asset management [22]. This central config makes it easy to change the database or asset paths if needed (though in a real deployment, one would externalize secrets like the DB URL rather than hard-coding them).

- **Database Session Management**:

- `db/session.py` – Sets up the database session factory and dependency for use in FastAPI endpoints. It creates a `SessionLocal` using SQLAlchemy's sessionmaker, bound to the engine from config [23]. It also defines `get_db()`, a dependency that yields a database session and ensures it's closed after each request [24]. This file encapsulates the pattern of "dependency per request" for DB access, so that endpoints can use `db: Session = Depends(get_db)` to get a transactional session.

- `db/init_data.py` – Contains logic to **load initial seed data** into the database. This is used on application startup if the database is being re-initialized. The `load_data(engine, df_entities, df_equipment, df_skills, df_specials)` function reads data from pandas DataFrame (CSV content) and populates the corresponding tables [25] [26]. It

first inserts all `Entity` records, commits them, then inserts `Equipment`, `Skill`, and `Special` records with proper foreign key links to those entities [27] [28]. The function wraps inserts in a try/except to catch errors: on any exception it rolls back the session and logs the error [29]. This modular seeding approach separates data loading from the main logic and makes it easy to adjust how initial data is provided (e.g., different files or data sets for different environments).

- **Data Models (`models/` folder)**: All persistent data structures are defined as SQLAlchemy ORM models in this folder. Each model corresponds to a database table. They all inherit from a common Base class (SQLAlchemy declarative base) defined in `models/base.py` [6]. Key model files include:

- `models/base.py` – Defines the `Base` object via SQLAlchemy's `declarative_base()` [6]. All other model classes extend this Base, which collects their metadata for table creation.
- `models/labyrinth.py` – Defines the **Labyrinth** model, representing a generated maze. Fields include an `id` (UUID primary key), `size` (dimension of the square maze), `seed` (string used for random generation reproducibility), `start_x` and `start_y` (the starting coordinates in the maze), and a `created_at` timestamp [4]. Relationships: a Labyrinth has a one-to-many relationship with `GameSession` (a maze can be associated with sessions) and with `Tile` (the collection of tiles/cells that make up the maze) [30]. Deleting a labyrinth will cascade to its tiles and game sessions (via `cascade="all, delete-orphan"` and foreign key on-delete rules).
- `models/tile.py` – Defines a **Tile** model, which represents a single cell in the labyrinth grid. Key fields: `labyrinth_id` (foreign key link to the Labyrinth it belongs to), coordinates `x` and `y`, `type` (tile category, e.g., corridor, turn, dead_end), and `open_directions` (which directions from this tile are pathways, stored as a string or JSON-encoded list) [31]. There is also a boolean `revealed` flag (whether the tile has been revealed to players, default false) [32]. Each Tile is linked back to its Labyrinth via a relationship [33].
- `models/game_session.py` – Defines the **GameSession** model, which represents an active game session (a playthrough instance of a labyrinth). Fields include an `id` (UUID primary key), `seed` (the seed of its associated labyrinth), `labyrinth_id` (foreign key to a Labyrinth), `size` (size of the maze for convenience), `start_x` / `start_y` (the starting position in the maze), and `created_at` timestamp [34]. Relationships: a GameSession links to one Labyrinth (`labyrinth` relationship) and has many `Player` and `MobileClient` entries associated [35]. In other words, each session knows which labyrinth it's using, which players are in it, and which client connections are attached.
- `models/player.py` – Defines a **Player** model, intended to represent a player character within a game session. Fields: an `id` (UUID primary key), `game_session_id` (foreign key to GameSession), `player_x` / `player_y` coordinates (the player's position in the maze), and an optional `username` [36]. It relates back to the GameSession via `game_session` relationship [37]. (In the current logic, player records may be created or updated as the game progresses, though the POC's focus is on session setup; movement or gameplay updates are not yet implemented in the endpoints.)
- `models/mobile_client.py` – Defines a **MobileClient** model, used to track a client device/connection associated with a session. Fields: `id` (UUID primary key), a `client_id` string (unique identifier for the client device or user connection), `connected_at` timestamp, and `game_session_id` (foreign key linking to the session) [38]. This model essentially logs which clients are currently connected to which session. Its relationship back to GameSession (`game_session`) allows quick access to all connected clients for a given session [39]. When a client joins or leaves, this table is updated.

- `models/game_entities.py`, `models/equipment.py`, `models/skills.py`, `models/specials.py` – These models represent static game content (e.g., characters and their attributes) loaded from seed files.

    - **Entity** (in `game_entities.py`) is a general game entity with fields like `name, type, age, role`, plus file paths for backstory and portrait images [40]. It uses a string primary key (possibly an ID from the CSV). Entity has one-to-many relationships with Equipment, Skill, and Special – i.e., an Entity can have a list of equipment items, a list of skills, and a list of special abilities [41].
    - **Equipment** (`equipment.py`) represents an item belonging to an Entity. It stores an `id`, `entity_id` (FK to Entity), `name`, and `description` [42]. There's a back-populated relationship to the Entity [43].
    - **Skill** (`skills.py`) represents a skill/ability of an Entity with fields `id`, `entity_id` (FK), `name`, and `description` [44], and relates back to Entity [45].
    - **Special** (`specials.py`) represents a special attribute or power of an Entity, with `id`, `entity_id` and a `description` [46], relating back to Entity [47].

    These four model types (Entity, Equipment, Skill, Special) are loaded at application start from CSV files (see **Data Initialization** below). They are mainly used to enrich the game session with character data (though the current POC endpoints do not directly expose them, they are part of the domain model for potential future use). All of them share the same Base and are created in the database on startup [13].

- **API Routes**:

- `routes/api.py` – Contains a collection of **API endpoints (routes)** organized with a FastAPI `APIRouter`. By convention, these endpoints are all under an `/api` URL path prefix (the router is included into the main app without a prefix in code, but each route path defined here starts with `/api/…`) [48]. This module defines higher-level game session management endpoints, such as creating a session, joining/leaving sessions, listing sessions or clients, and toggling player readiness. By separating these routes into a dedicated file, the project keeps the main application file cleaner and groups related endpoints logically. The router uses `Depends(get_db)` for DB access and includes necessary error handling for missing sessions, etc. We will detail these endpoints in the **REST API Endpoints** section, but in summary this file handles the core gameplay API (session lifecycle and player coordination). It also utilizes utility functions (like `generate_labyrinth` and `broadcast_session_update`) and the global state for readiness to fulfill requests [49] [50].

- **Real-time Communication**:

- `realtime.py` – Implements the **WebSocket communication layer** for real-time updates. It maintains a mapping of active WebSocket connections per session in memory (`active_connections: Dict[session_id, List[WebSocket]]`) [51]. Key functions include:

    - `connect_to_session(session_id, websocket)`: Accepts a WebSocket connection and adds it to the list for the given session, initializing the list if necessary [52].
    - `disconnect_from_session(session_id, websocket)`: Removes a WebSocket connection on disconnect; if a session's list becomes empty, that session key is removed [53].

- `broadcast_session_update(session_id, message: dict)`: Sends a JSON message to every active WebSocket client for the specified session [10]. (It catches exceptions from any failed sends and simply ignores them, ensuring one bad connection doesn't crash the loop.)
- `mount_websocket_routes(app)`: A function that registers the WebSocket endpoint with the FastAPI app [54]. It creates a router with a single websocket route `/ws/{session_id}` and includes it into the app. The websocket route uses the above connect/disconnect functions: when a client connects to `/ws/{session_id}`, the server accepts the connection and enters a loop waiting for messages [2]. (The server doesn't process incoming messages beyond reading them; the loop `await websocket.receive_text()` is essentially a keep-alive/read loop. If the connection closes, a `WebSocketDisconnect` is caught and the server cleans up by removing that socket [55].)

This design means clients subscribe to a session channel and the server pushes out updates (like "player X is ready/not ready") to all subscribed clients via `broadcast_session_update`. The WebSocket handling is kept separate from the main logic for clarity.

• **Utility Modules**:

• `utils/corrected_labyrinth_backend_seed_fixed.py` – Contains the **Labyrinth generation algorithm** and related helper functions. The primary function here is `generate_labyrinth(size: int, seed: Optional[str], db: Session)` which programmatically creates a randomized maze of given size [56]. Notable aspects of this function:

- It uses a depth-first search (DFS) maze generation: starting from a random cell, it recursively "drills" passages in random directions ensuring a perfect maze (no loops, one connected component) [57]. It tracks visited cells and opens paths in random order [57].
- It determines the tile type based on how many directions are open (e.g., 1 open direction = dead end, 2 = straight corridor or turn, 3 = T-intersection, 4 = crossroad) [58].
- If no seed is provided, a random UUID hex string is generated as the seed [59]. This seed is used to initialize Python's random generator to make the maze repeatable given the same seed.
- The function builds an in-memory `tile_map` of all cells and their open directions via the DFS. It picks a random start cell within the grid as the entry point of the labyrinth [60] (this becomes `start_x, start_y`).
- It then creates a Labyrinth ORM object, saves it to the DB (so it gets an ID), and iterates through the `tile_map` to create a Tile ORM object for each cell [61] [62]. Before insertion, it assigns each tile a type and formats the open directions as a JSON string if multiple directions (or a single letter if only one) [63] [64]. All tiles are added to the session and then committed in bulk [65].
- Simultaneously, it prepares a `tiles_response` list of dictionaries containing the essential info for each tile (x, y, type, and an image filename for that tile's graphic) [66]. Filenames are chosen via `get_image_filename()` based on tile type and open directions (for example, a corridor with North-South openings gets `"tile_corridor_NS.png"`) [63] [67].
- The function returns a tuple `(labyrinth, tiles_response)` [68]. The `labyrinth` is a SQLAlchemy object (with id, seed, etc.), and `tiles_response` is a ready-to-use JSON-like structure for client consumption. This separation is important: the labyrinth object contains relationships and other fields not directly serializable, so the tiles_response provides the

needed map layout in plain data. *(The code comments explicitly note not to assign the tiles list to the labyrinth object to avoid accidental serialization issues [68].)*

This utility encapsulates the core maze logic, keeping the API endpoints simple. The design choice here was likely to generate and persist the maze in one go, so any client can join later and the maze is already stored in the DB for reference (e.g., if we had an API to fetch the maze by session).

- **Schema Definitions**:

- `schemas.py` – Defines **Pydantic models** for request and response bodies. This helps decouple the API data schemas from the database models. Key classes include:
  - `ClientJoinRequest` with a single field `client_id` (string) [69] – used when a client wants to join a session.
  - `GameSessionCreateRequest` with `size: int` and optional `seed: str` [70] – used to create a new game session (mirrors what the labyrinth generation needs).
  - `PlayerStatus` with `client_id: str` and `ready: bool` [71] – represents an individual player's ready state, used in readiness toggle and status.
  - `SessionStatus` with `players: List[PlayerStatus]` and `all_ready: bool` [72] – represents the overall readiness state of a session (the list of players and a flag if everyone is ready). These schema definitions are used as types in function signatures (FastAPI will automatically parse JSON into these, and also use them to format response JSON). For example, the toggle readiness endpoint returns a `SessionStatus` Pydantic model [73], which FastAPI will serialize to JSON matching that schema. The separation of schemas also allows easier changes to API output without affecting the database layer.

- **Global State**:

- `state.py` – Holds shared in-memory state and synchronization primitives. Specifically, it defines `session_readiness: Dict[str, Dict[str, bool]] = {}` – a nested dictionary mapping session IDs to a mapping of client IDs and their "ready" status (True/False) [11]. This is used to track which players in a session have marked themselves as ready. It also defines `lock = threading.Lock()` [11], a global lock used to ensure thread-safe updates to the `session_readiness` structure. Since FastAPI could run under multi-threaded servers or handle multiple requests concurrently, this lock is used whenever the readiness dict is modified or read in a non-atomic way (e.g. toggling a status) [74]. The design is simple: use a standard mutex to avoid race conditions on shared data.

- **Static Frontend**:

- `frontend/` **directory** – Contains static files served by the backend (this is not Python code, but worth noting in the architecture). Notably, `frontend/index.html` is a simple HTML/JavaScript file that provides a UI for interacting with the backend. It has forms and buttons to create game sessions, generate labyrinth previews, join sessions, etc., and uses the Fetch API to call the backend routes. For example, it calls the `/generate-labyrinth` endpoint to get a maze and then renders it, and uses `/api/game_sessions` endpoints to manage sessions and clients (e.g., `fetch("/api/game_sessions")` to list sessions) [75] [76]. The front-end also connects to the

WebSocket endpoint for real-time updates (not shown in the snippet above, but presumably the JS would open `new WebSocket("ws://.../ws/{session_id}")` when a client joins a session to listen for updates). The `frontend/tiles/` subdirectory contains image assets for the maze tiles (PNG images referenced by the `image` field in the labyrinth response). The backend mounts the `frontend/` folder as static files at the root path, meaning any request to "/" that doesn't match an API route will serve `frontend/index.html` by default [15], and the `/tiles` path serves the images [15]. This integration is mainly for convenience in this POC architecture; in a production scenario the front-end would likely be a separate application or served differently.

- **Startup Script**:

- `startup.sh` – A shell script provided to run the application (likely used in deployment or development). This script appears to reset and start the server. It uses a Flask CLI command to drop and recreate the database tables, then launches the app with Gunicorn:
    - It runs `flask shell` to import the `db` from `main` and call `db.drop_all()` and `db.create_all()` [77]. (This suggests that at one point the app might have used Flask-SQLAlchemy; however, in the current FastAPI implementation, this part may be a leftover. The intention is to ensure a clean database on each start in the POC.)
    - Then it starts the FastAPI app using Gunicorn on 0.0.0.0:5000 by pointing to `main:app` [78]. Gunicorn likely uses an ASGI worker (such as uvicorn worker) since FastAPI is ASGI. The presence of `uvicorn` in requirements and this command indicates deployment via Gunicorn/Uvicorn.
    - This script highlights the POC nature: it force-resets the DB each time and doesn't handle data persistence across restarts. It also uses a fixed port. For a production setting, you'd remove the DB reset and maybe use environment variables for configuration instead of a fixed script.

In summary, the codebase is structured to separate concerns: **models** define the data schema, **database modules** handle connections and seeding, **routes** implement the API, **utils** contain game logic (maze generation), **state** and **realtime** manage in-memory session state and WebSockets, and the **main** ties it all together. The interactions are mostly via function calls and shared objects: for instance, the `routes/api.py` functions call into `utils.generate_labyrinth` to create mazes and use `realtime.broadcast_session_update` to notify clients [49] [79], and they manipulate the `state.session_readiness` structure to track players [74]. All components come together within the FastAPI app context.

Functional Flows Explanation

This section describes how key functionalities are implemented and how data flows through the system for each use case. We cover the major flows: **game session lifecycle**, **player (client) lifecycle including a rudimentary authentication approach**, **messaging (real-time updates)**, and **data initialization/cleanup**. Each flow explanation ties together the relevant components from above.

1. Game Session Lifecycle and Labyrinth Generation

Creating a Game Session (and Maze): A game session represents a playable instance of a maze that players can join. The typical flow to create a session is:

- A client (or the front-end UI) sends a request to create a new session. This can be done via the **REST API** (either `POST /create-game-session` or `POST /api/game_sessions/create`). The request includes the desired maze size and an optional seed. For example, the front-end calls `fetch("/api/game_sessions/create", {... body with size/seed ...})` when the user clicks "Create Game Session" [75].
- The backend endpoint handling session creation receives a `GameSessionCreateRequest` (with `size` and optional `seed`) and begins the process. In the `routes/api.py` implementation, this is the `create_game_session` function [80]. It calls the labyrinth generator utility: `labyrinth, _ = generate_labyrinth(request.size, None, db)` [81]. (If a seed was provided in the request, it would pass that; currently it passes `None` or the request's seed accordingly. The main difference between the non-`/api` endpoint and the `/api` endpoint is that the former uses `request.seed` directly [82], while the latter example passes `None` if no seed – both ultimately allow random seeding or fixed seed.)
- **Labyrinth Generation**: Inside `generate_labyrinth(size, seed, db)`, the function creates a new Labyrinth record and the associated Tile records in the database as described earlier. It uses a DFS algorithm to carve out a maze and inserts records: first a Labyrinth row (with generated `seed`, random start coordinates, etc.), then each cell as a Tile row [61] [62]. The function commits these to the database, so at this point the new labyrinth and all its tiles are persisted. It then returns the Labyrinth object and a list of tile info dictionaries.
- Back in the endpoint, after receiving the `labyrinth` object from the generator, a new `GameSession` object is created. The session is populated with: a new UUID for its ID, the `seed`, `labyrinth_id` (linking to the created labyrinth), the `size`, `start_x`, `start_y` (copied from the labyrinth), and a timestamp [83]. This session object is added to the DB and saved (committed) [84]. Now the database has: one Labyrinth, many Tiles, and one GameSession referencing that labyrinth.
- The endpoint returns a response confirming the session creation. In the `/api` route, it returns a JSON with fields like `session_id`, `seed`, `labyrinth_id`, `size`, `start_x`, `start_y` [85]. In the non-`/api` route defined in `main.py`, it uses a Pydantic model `GameSessionResponse` to serialize the session, which includes similar fields and the creation timestamp [86] [18]. The front-end will typically use this response to display the new session info or possibly automatically join the session.

Key design aspects in this flow: - **Decoupling and Reuse**: The labyrinth generation is decoupled from session creation. This allows the application to also expose a standalone maze generation endpoint (`/generate-labyrinth`) for previewing a maze without creating a session [87]. The flow for `POST /generate-labyrinth` is similar, except it does not create a GameSession; it just calls `generate_labyrinth`, then queries the created tiles to build a response structure with tile images [88] [89], and returns the maze details (seed, start, tiles list) [90]. This is useful for a user to see a randomly generated maze before deciding to start a session. - **Data Integrity**: The labyrinth and tiles are created inside a database transaction. By committing the labyrinth before tiles, and using the labyrinth's ID for each tile, referential integrity is maintained. If any error occurred in generating tiles, the code would raise and the session creation would not finalize. (Note: There's a validation in `generate_labyrinth` that size must

be between 4 and 10, otherwise it raises a `ValueError` [56]. Currently this error isn't caught and would cause a server error response – improving this by returning a 400 Bad Request could be a future refinement.) - **Design Choice**: The game session stores a `size` and a duplicate `seed` field even though it can get those from the labyrinth. This redundancy simplifies query and response – the session can be displayed with its essential info without always joining with the labyrinth table. It's a denormalization trade-off for convenience in this POC [34]. - After creation, the game session is effectively "open" for players to join. At this point, no players are associated yet, and no WebSocket connections exist for it. The labyrinth and session will persist in the DB until cleaned up (explicitly or by a DB reset).

Destroying Game Sessions: For testing or restarting scenarios, the backend provides a way to clear all sessions: - A client (or developer) can call **DELETE** `/destroy-all-sessions` (defined in `main.py`). This endpoint simply deletes all records from the `game_sessions` table in the DB [91] and commits. Because of the `ondelete="CASCADE"` on the `game_sessions.labyrinth_id` foreign key [92] and the relationships, deleting a session can also delete related `Player` and `MobileClient` records (via their foreign keys). However, note that the Labyrinth and Tile records are not automatically removed by deleting sessions (no session->labyrinth cascade in the current schema). In the current design, this endpoint is primarily to quickly wipe play sessions; orphaned labyrinths/tiles may remain in the DB until a full reinit. This is acceptable for a dev/test environment where the DB is frequently reset entirely, but would be tuned in a full application (e.g., to also remove unused labyrinths or reuse them). - The front-end has a "Destroy All Sessions" button which triggers this (though the JavaScript function for it wasn't fully shown in the snippet, presumably it calls this DELETE endpoint and then refreshes the session list).

2. Client Connection and Session Join Flow (Pseudo-Authentication)

The application does **not implement traditional user authentication** (no login, tokens, or OAuth in this POC). Instead, it uses a simple mechanism to identify clients: a client provides a unique `client_id` (string) when joining a session. This could be any identifier (for example, the front-end might generate a UUID for the device or ask the user for a name). The `client_id` is then used to track that client's connection. Here's how the join flow works:

- A client decides to join an existing game session. The client must know the `session_id` (which could be shared out-of-band or obtained from the list of sessions). The front-end UI lists available sessions and their IDs [93], and could provide a way for the user to pick one to join.
- The client sends a **POST** request to `/api/game_sessions/{session_id}/join` with a JSON body containing their `client_id` [94]. For example, `{ "client_id": "player123" }`.
- The server (in `routes/api.py`) handles this in the `join_game_session` function:
- It first looks up the game session in the database by the provided `session_id`. If not found, it returns a 404 error (FastAPI raises an HTTPException) [95].
- It then checks if a MobileClient entry already exists with the given `client_id` (regardless of session) [96]. This ensures a client cannot join twice with the same ID. If such a record exists, it simply returns a message indicating the client is already connected [96] (and does nothing else).
- If the client_id is unique and session exists, it creates a new **MobileClient** record in the DB: setting the `client_id`, linking it to the `game_session_id`, and recording the current time as `connected_at` [97]. This insertion is committed to the database so that the session's connected clients list is updated persistently.

- Next, the function updates the in-memory readiness state. This is done in a thread-safe way by locking the global `lock` and then:
  - Ensuring an entry exists in the `session_readiness` dict for this session (if this is the first client for that session, it creates an empty dict for it) [12].
  - Adding the new client to the `session_readiness[session_id]` map with a default ready status of `False` (not ready upon joining) [98].
  - It then prepares a `SessionStatus` object which includes all players' ready states. It creates a list of `PlayerStatus` for each client currently in the session's map and marks `all_ready=False` since a new client is unready [99]. This SessionStatus essentially captures the updated state of the lobby.
  - While still under the lock, it calls `broadcast_session_update(session_id, session_status.dict())` to send this updated readiness info to all connected WebSocket clients for that session [100]. This means if other players were already connected and listening, they get notified that a new player joined and is currently not ready.
- After releasing the lock, the server responds to the join request with a JSON message containing: a success message, the session details (session_id, map seed, labyrinth_id, start coordinates, and size) [101]. This lets the joining client know they have successfully connected and provides them the data needed to begin the game (e.g., they now know the maze's seed and starting position).
- On the client side, after posting to join, the client would typically open the WebSocket connection to `/ws/{session_id}` to start receiving real-time updates (if not already connected). The response includes the `session_id` which they likely already had, but also confirms they're in. The design assumes trust in the provided client_id (no duplicate name checking beyond uniqueness and no authentication token to prove identity – it's purely an ID string).

Note on Authentication: This flow is essentially a replacement for a login in this POC. The `client_id` acts like an authentication identity, but it's not secure. There's no password or token – any user who knows a client_id could impersonate that client by calling join or toggle readiness with that ID. In a real application, this would be handled with proper user accounts or unique tokens per connection. For the POC's purposes, this simplistic approach was chosen likely to avoid the complexity of an auth system. All REST endpoints are **unprotected** (no auth required), meaning anyone with network access could create or join sessions. This is acceptable in a closed test environment but not in production.

Leaving a Session: The counterpart is the leave flow: - A client that is connected can leave (disconnect) from a session by calling **POST** `/api/game_sessions/leave` with their `client_id` in the body [102]. No session_id is needed here because the assumption is a given client ID can only be in one session at a time (the MobileClient table design allows a client to be associated with at most one session). - The server finds the MobileClient record by client_id. If not found, it returns 404 (meaning the client was not connected) [103]. If found, it deletes that record from the database and commits [104]. - It responds with a simple message `'Disconnected successfully'` [104]. - Notably, the current implementation does **not** update the `session_readiness` or broadcast the departure to others. This might be an oversight; ideally, upon leaving, the server should remove the client from the `session_readiness` dict (and possibly broadcast an update so remaining players know someone left). Because that isn't done, the in-memory state might temporarily still think that client is present (with whatever their last ready status was) until perhaps a restart or if someone toggles ready and the departed client is still in the dict. This is a minor inconsistency in the POC. It doesn't affect the database (which correctly no longer lists the client), but it's a detail to refine. In practice, since the WebSocket disconnect handler will remove the connection [55], no further broadcasts will

be sent for that client, and if `get_session_status` is called later, it will still list the client in players unless manually removed. For a POC, this is tolerable.

- When a client actually closes their WebSocket (like the app is closed), the `realtime.py` logic will remove their socket from `active_connections` but it has no hook to remove them from `session_readiness`. So the state.py dict may retain stale entries until server restart or if a leave API was called. The design is focused on demonstrating the join/ready concept and could be extended to handle disconnects more thoroughly (e.g., by integrating the leave API call or adding a timeout for readiness).

Client State Recovery: There is a helper endpoint **GET** `/api/game_sessions/client_state/{client_id}` that helps a client (or the UI) determine if a given client_id is currently connected to a session, and if so, which one [105] [106]. This is useful if, say, a user refreshes their app – they can call this endpoint with their client_id to see if the server still thinks they're in a session. The endpoint checks the MobileClient table for that ID, and if found, returns the session details (same info as join response) [107] [108]. If not, it returns that `connected_session: None`. This isn't a real "login" but it's a way to resume state in a stateless HTTP manner. The design choice to include this suggests the developers considered app reconnect scenarios, given there's no token to remember the session otherwise. Essentially, the client_id is the key to get back in.

3. Player Readiness and Real-Time Messaging Flow

Once players have joined a session, they typically need to coordinate before the game starts. The concept of **"ready"** state is introduced: each player can signal they are ready/unready, and when all are ready, the game could commence. The backend manages this via an in-memory structure and broadcasts updates using WebSockets. Here's how that works:

Toggling Readiness: When a player is in a session (i.e., has joined via the above flow), the UI will likely show a "Ready" checkbox or button. Toggling that sends a request to **POST** `/api/game_sessions/{session_id}/toggle_readiness` with the client's ID and their new ready status. For example, `{ "client_id": "player123", "ready": true }`. The front-end can obtain the `session_id` from when they joined or from stored state, and call this endpoint (in a POC, they might call it directly via some UI control).

- The server's `toggle_readiness` handler in `routes/api.py` receives the session_id (as a path param) and a `PlayerStatus` payload (client_id + ready bool) [109].
- It acquires the global lock to safely update shared state [74]. Within the locked block:
- If the session_id is not yet in the `session_readiness` dict, it initializes it (this covers any case where a session might not have been in memory, though normally it should exist after a join; this is a safeguard) [74].
- It then sets `session_readiness[session_id][client_id] = payload.ready` to the new value [110] (overwriting whatever was there).
- It builds a list of `PlayerStatus` objects for all players in that session from the updated dict (so it collects every `client_id` and their ready status) [110].
- It computes `all_ready = all(p.ready for p in players)` – a boolean flag indicating if every player in the session is now ready [111].

- It creates a `SessionStatus(players=..., all_ready=...)` object representing the new state [79].
- Then it calls `await broadcast_session_update(session_id, session_status.dict())` to push this new status to all connected clients' websockets [112]. This means every player in the session will immediately receive an updated list of who's ready.
- The lock is released, and the endpoint returns the new `SessionStatus` as the HTTP response as well [113]. The response is typically not as critical because the clients will get the info via WebSocket, but it could be used by the caller to update itself if needed or to confirm the action.

Real-Time Update Delivery: When `broadcast_session_update` is called, the function in `realtime.py` goes through all active WebSocket connections for that session and sends the JSON message. The message is effectively the content of `SessionStatus.dict()`, which looks like:

```
{
  "players": [
    {"client_id": "player123", "ready": true},
    {"client_id": "player456", "ready": false}
  ],
  "all_ready": false
}
```

(for example, if one player is ready and another is not). This lets each client's front-end script update the UI (e.g., show checkmarks next to ready players, or enable a "Start Game" button only when `all_ready` becomes true). Because FastAPI's WebSocket send is asynchronous, and the code awaits it, it will run concurrently to sending the HTTP response.

Getting Session Status: If a client joins mid-way or wants to refresh the readiness status, the **GET** `/api/game_sessions/{session_id}/status` endpoint provides the current `SessionStatus` snapshot [73]. This is a read-only operation that also uses the lock to safely read the `session_readiness` dict and construct the players list and all_ready flag [114]. It doesn't broadcast anything. The front-end might call this once after joining (in case it joined after other players toggled ready) to catch up with the current state. After that, WebSocket messages will keep it updated.

Flow of a Ready Toggle Example: 1. Player A clicks "Ready". The front-end calls `toggle_readiness` with `ready=true`. 2. Server sets A's status to true in memory, sees maybe players A (true) and B (false), thus all_ready = false. It broadcasts this state. 3. Both A and B's clients (via WebSocket) receive the message and update their displays (A sees B not ready, B sees A ready). 4. Player B then clicks "Ready". The server sets B's status to true, now players A and B are true -> all_ready = true. Broadcast occurs. 5. Both clients get the message that now all_ready=true. The UI could now indicate the game can start (the POC front-end might not have the next step implemented, but logically that would be the signal). 6. If any player toggles back to false, all_ready goes false and a broadcast goes out similarly.

Messaging Design Discussion: - The use of WebSockets here is to ensure **low-latency updates**. Instead of each client polling the `/status` endpoint repeatedly, one client's action immediately triggers updates on all other clients. This is critical in multiplayer scenarios to synchronize state (like a lobby readiness or even positions during gameplay). - The decision to ignore any messages sent *from* clients on the WebSocket

(`await websocket.receive_text()` is called in a loop but the content is not used [115]) means the server is currently treating the WebSocket as a pure broadcast channel. Clients don't send gameplay info via WebSocket in this design – they use HTTP endpoints for actions (join, ready, etc.), and only listen on the WebSocket. This simplifies server logic (no need to parse messages) but slightly limits interaction (e.g., you couldn't easily send a chat message or real-time movement without extending the WebSocket handling to process incoming data). - The global `session_readiness` dict means the readiness state is ephemeral – it is lost if the server restarts. However, since the POC resets on each run and game sessions themselves likely wouldn't persist long, this is acceptable. If persistence of readiness was needed (not usually, since if the server restarts all players would have to rejoin anyway), it could be stored in the database. The choice to keep it in memory was likely for speed and simplicity. - By broadcasting the entire list of players and their status on each change, the client logic is simplified (it can redraw the whole list). This is fine given the small number of players typically in a session (maybe a handful). If scaling up, one might broadcast only the deltas (e.g., "player X became ready") but that adds complexity to the client.

4. Data Initialization and Lifecycle Management

This flow concerns what happens to the data outside of the normal gameplay actions – specifically at application startup and during cleanup.

Startup Initialization: Each time the application starts (server launch), it runs the `startup` event handler in `main.py` [16]. The flow is:

- `init_db()` from `config.py` is called, which will create any tables that don't exist in the database (via `Base.metadata.create_all`) [116]. This ensures the schema is up-to-date with the models.
- If the flag `FORCE_REINIT_DB` is set to True (which in the code it is by default [117]), the app will **reinitialize the entire database**:
- It prints a warning message in logs and then calls `EntityBase.metadata.drop_all(bind=engine)` to drop all existing tables [118]. Here `EntityBase` is an alias for the same Base (imported from `models.game_entities`), so this drops all tables for all models (game sessions, labyrinths, tiles, players, mobile_clients, entities, equipment, skills, specials, etc.).
- Then `EntityBase.metadata.create_all(bind=engine)` is called to recreate all tables fresh [119].
- It proceeds to load seed data from CSVs: it reads four CSV files (`entities.csv`, `equipment.csv`, `skills.csv`, `specials.csv`) into pandas DataFrames [14]. These files contain initial data for the game entities and their attributes.
- It calls `load_data(engine, df_entities, df_equipment, df_skills, df_specials)` to populate the tables with this data [120]. As described, this will insert all Entities then their related Equipment/Skills/Specials with proper foreign keys [25] [26].
- The net result is that on startup, the database is wiped and then pre-populated with a known set of entities (e.g., characters or monsters) and their gear/abilities. This is done every time (since FORCE_REINIT_DB is True) so the POC always starts from a clean slate.
- This design is very useful during development (no need to manually reset DB or handle migrations of new columns – starting the app always yields a fresh schema). However, it means any game sessions or labyrinths created in a previous run are lost on restart. In a production scenario, you would likely turn off `FORCE_REINIT_DB` to preserve data, and run migrations for schema changes instead.

- The use of pandas for CSV reading and the printouts of inserted IDs [121] shows this is a one-time bulk load, not something intended to run repeatedly in a long-running server. It's primarily for demonstration content.

Data Persistence During Runtime: During the server runtime, all created game sessions, labyrinths, tiles, players, and connected clients are stored in the PostgreSQL database. This means they survive as long as the database isn't reset. If `FORCE_REINIT_DB` were False, you could stop and start the server and still have previous sessions in the DB (the code would then only do create_all without dropping, so tables would persist). However, the current setting effectively ties the lifecycle of data to the lifecycle of the app process (for sessions and such).

Cleanup/Clear Flows: - As mentioned, **DELETE** `/destroy-all-sessions` is an explicit cleanup of game session data during runtime. This is useful to clear out old sessions without restarting the server or wiping static data. It leaves the static reference data (entities, etc.) intact, since those might be needed for any new sessions. - There is no specific endpoint to clear out players or labyrinths individually, but they would be removed by cascade if their parent session is removed (players and mobile clients cascaded via session deletion, labyrinths not automatically removed unless we drop them). - The **MobileClient** entries are removed when a client leaves, or they would be removed if the entire session is deleted (MobileClient has ondelete cascade via game_session_id as well [122], so deleting a session should also drop its MobileClient records).

Error Handling and Data Consistency: - The code is careful in some places to catch errors (e.g., seeding data) and ensure the DB is not left in a partial state by using rollback on exception [29]. - For gameplay flows, it uses HTTP exceptions to signal invalid operations (join nonexistent session, leave nonexistent client, etc.), which translates to appropriate HTTP status codes. - One potential data consistency issue to be aware of: if multiple clients rapidly toggle readiness or join/leave concurrently, the global lock ensures the in-memory state stays consistent. The database operations for join/leave are simple and atomic per request. There isn't a scenario where a game session's core data would be corrupted by concurrent access in this design (since writing to the same session from two endpoints either happens sequentially with locks for shared state, or if it's just DB writes, they're separate records). - If the server did not use a global lock, two clients toggling at the same time could intermix reads/writes to the dict; the design wisely avoids that by using `with lock:` around those critical sections [12] [74]. This is essentially a concurrency design choice: using a simple thread lock instead of more complex async locks or database-backed state.

In summary, the data lifecycle is straightforward: start fresh (or from seeds), create sessions and associated data during runtime, optionally reset or clear as needed, and tear down (losing in-memory state) on shutdown. Persistence is mainly for the game content and session records; transient coordination info is kept in memory.

REST API Endpoints Documentation

The backend exposes a variety of RESTful endpoints for managing game sessions, players, and labyrinths. Below is a comprehensive list of all endpoints, including their purpose, inputs, and outputs. (Note: Some endpoints exist in two forms – a base path and an `/api/...` path – due to the project evolving. The `/api` endpoints are the primary ones for client use, but the older base endpoints are still present for certain actions like labyrinth generation and session listing.)

Game Session and Labyrinth Endpoints (HTTP REST)

- **GET** `/game-sessions` – *List all game sessions.* Returns a list of existing game sessions (in JSON). Each session includes fields such as id, seed, labyrinth_id, start_x, start_y, and created_at [18]. This endpoint is defined in `main.py` and uses a Pydantic response model list, but effectively returns the same data structure as the `/api/game_sessions` endpoint. Example response (200 OK):

```
[
  {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "seed": "abcdef123456...",
    "labyrinth_id": "123e4567-e89b-12d3-a456-426614174000",
    "start_x": 3,
    "start_y": 5,
    "created_at": "2025-05-12T18:40:00",
    "size": 7
  },
  ...
]
```

(Size is included in the /api variant; in this main version, `GameSessionResponse` model includes it since `from_attributes=True` allows Pydantic to pull all fields [123].)

- **POST** `/create-game-session` – *Create a new game session (with a new labyrinth).* Expects a JSON body with `{"size": <int>, "seed": "<string>"}` (seed is optional). Creates a labyrinth of the given size and uses it to initialize a new game session. Returns the details of the created session as JSON [82] [85]. On success (201 Created), the response includes: `session_id`, `seed`, `labyrinth_id`, `size`, `start_x`, `start_y`, and possibly a message. For instance:

```
{
  "message": "Game session created successfully",
  "session_id": "550e8400-e29b-41d4-a716-446655440000",
  "seed": "abcdef123456...",
  "labyrinth_id": "123e4567-e89b-12d3-a456-426614174000",
  "size": 7,
  "start_x": 3,
  "start_y": 5
}
```

This is the `/api/game_sessions/create` variant's output [85]. The non-`/api` version returns a `GameSessionResponse` model which has the same fields without the wrapping message.

- **POST** `/generate-labyrinth` – *Generate a maze layout without creating a session.* Expects JSON `{"size": <int>, "seed": "<string>"}` (seed optional). Returns a **LabyrinthResponse**

containing the maze's seed, start coordinates, and a list of all tiles with their type, open directions, and an image filename for that tile [90]. This is used to preview or display a maze. Example response:

```
{
  "seed": "abcdef123456...",
  "start_x": 3,
  "start_y": 5,
  "tiles": [
    { "x":0, "y":0, "type":"dead_end", "open_directions":["E"],
"image":"tile_dead_end_W.png" },
    { "x":1, "y":0, "type":"corridor", "open_directions":["W","E"],
"image":"tile_corridor_EW.png" },
    ...
  ]
}
```

Each tile's `open_directions` is an array of directions (N,S,E,W) and `type` indicates the shape. (Under the hood, this endpoint calls `generate_labyrinth` and then queries the DB for each tile to format the open_directions properly and assign images [88] [89].) If the size is out of bounds (<4 or >10), this will error (500 with ValueError message) since no custom error handling is added for that in this endpoint.

- **DELETE** `/destroy-all-sessions` – *Delete all game sessions (and their players/clients).* This is a bulk-destructive operation for convenience. No input body needed. It will wipe all rows in the `game_sessions` table [91]. In the database, this cascades to `Player` and `MobileClient` records via foreign keys. It returns a confirmation message on success, e.g. `{"detail": "All game sessions deleted"}` [19]. Use with caution, as it doesn't ask for confirmation. (There is no `/api` equivalent for this in the code; it's only in main.)

- **GET** `/api/game_sessions` – *List all game sessions (API variant).* Returns a JSON object with a `sessions` key, whose value is a list of session objects [124]. Each session object includes id, seed, size, labyrinth_id, start_x, start_y, created_at, and related fields. (Since it's using the ORM models directly in the response, it might include relationships if converted to dict; but FastAPI likely pydantic-ifies them. The front-end expects just the basic fields as seen in usage [93].) Example:

```
{ "sessions": [ { "id": "...", "seed": "...", "size": 7, "labyrinth_id":
"...", "start_x": 3, "start_y": 5, "created_at":
"2025-05-12T18:40:00" }, ... ] }
```

This is similar to the GET `/game-sessions` output but wrapped in an object, which the front-end then iterates over [125].

- **POST** `/api/game_sessions/create` – *Create game session (API variant).* Same functionality as `/create-game-session` above. Accepts `size` and optional `seed` in JSON [80]. Returns a JSON

with session details and a `"message": "Game session created successfully"` [85]. This is used by the front-end UI for creating sessions [126].

- **POST** `/api/game_sessions/{session_id}/join` – *Join a specific game session.* Path parameter is the session's UUID. Body is `ClientJoinRequest` JSON: `{"client_id": "<your-id>"}` [94]. On success, the response contains:

- `"message": "Connected successfully"` and
- The session info: `session_id`, `map_seed` (alias for seed), `labyrinth_id`, `start_x`, `start_y`, `size` [101].
  Example success response:

```
{
  "message": "Connected successfully",
  "session_id": "550e8400-e29b-41d4-a716-446655440000",
  "map_seed": "abcdef123456...",
  "labyrinth_id": "123e4567-e89b-12d3-a456-426614174000",
  "start_x": 3,
  "start_y": 5,
  "size": 7
}
```

If the session ID doesn't exist, you get a 404 `{"detail": "Session not found"}` [95]. If the client_id was already connected, it returns a 200 with `{"message": "Client already connected"}` (and does not duplicate the entry) [96]. This call also triggers a WebSocket broadcast internally (no indication in HTTP response, but other players will be notified of the new client). The client calling join should next open the WebSocket connection to start receiving updates.

- **POST** `/api/game_sessions/leave` – *Leave the current session.* Body is `{"client_id": "<your-id>"}`. This removes the client from whatever session they were in. Success response: `{"message": "Disconnected successfully"}` [104]. If the client_id wasn't found (not connected), 404 `{"detail": "Client not connected to any session"}` [103]. Typically called when a player exits the lobby or game. (No session_id is needed since the server can derive it from the client record.)

- **GET** `/api/game_sessions/{session_id}/clients` – *Get list of connected clients in a session.* Path param is session ID. Returns `{"clients": [ ... ]}` where each item has a `client_id` and `connected_at` timestamp [127] [128]. For example:

```
{
  "clients": [
    { "client_id": "player123", "connected_at":
"2025-05-12T18:45:10.123456" },
    { "client_id": "player456", "connected_at":
```

```
"2025-05-12T18:46:05.789012" }
  ]
}
```

If the session doesn't exist, 404 is returned. This endpoint is used by the front-end to populate the "Connected Clients" column in the sessions table for each session [129]. It queries the `MobileClient` records related to the session and formats them.

- **GET** `/api/game_sessions/client_state/{client_id}` – *Check a client's current session.* Path param is the client's ID. Returns information about the session that this client is connected to, if any. On success (client is connected):

```
{
  "client_id": "player123",
  "connected_session": "550e8400-e29b-41d4-a716-446655440000",
  "session_details": {
     "session_id": "...same as above...",
     "labyrinth_id": "123e4567-e89b-12d3-a456-426614174000",
     "seed": "abcdef123456...",
     "size": 7,
     "start_x": 3,
     "start_y": 5
  }
}
```

If the client is not connected to any session, it returns their ID and `connected_session: null` (with `session_details: null`) [107] [108]. This endpoint does a DB lookup on MobileClient and then GameSession to gather details. It's useful for client reconnection logic as discussed.

- **POST** `/api/game_sessions/{session_id}/toggle_readiness` – *Toggle or set a player's ready status.* Path param is session ID. Body is `{"client_id": "<your-id>", "ready": <true/false>}` conforming to `PlayerStatus` schema [109]. This updates the readiness state in memory and broadcasts to others. The response is the updated `SessionStatus` object for the session [79] [113]. For example:

```
{
  "players": [
    {"client_id": "player123", "ready": true},
    {"client_id": "player456", "ready": false}
  ],
  "all_ready": false
}
```

The client who made the request will get this response, and all clients (including the requester) will also get the update via WebSocket. This endpoint always returns 200 OK (there's no case in code where it 404s – if a session_id wasn't in memory, it just initializes a new entry). If a client_id that isn't actually in the session calls it, the readiness dict will get an entry anyway (this could lead to inconsistency, but that scenario is unlikely if the UI only lets joined players call it). In practice, one would secure this or verify the client is part of the session.

- **GET** `/api/game_sessions/{session_id}/status` – *Get current readiness status of a session.* Path is session ID. Returns a `SessionStatus` JSON similar to the toggle response, showing all players and whether all are ready [114]. If the session has no entry in the readiness dict (no one joined yet or server restart), it will return `players: []` and `all_ready: false`. This is a safe way to poll the state if needed, but typically the WebSocket pushes make this mostly unnecessary except on first load.

WebSocket Endpoint and Communication Pattern

- **WebSocket** `/ws/{session_id}` – This is not an HTTP GET/POST endpoint, but a persistent connection upgrade. Clients connect to this endpoint to receive real-time updates about the session with ID `session_id`. Once connected, the client remains in the socket and the server can send messages at any time. Key points of this WebSocket interface:
- To connect: the client (likely the front-end JavaScript or a mobile app) opens a WebSocket to `ws://<server>/ws/<session_id>`. The server, on accept, will add that socket to the internal list for the session and keep the connection alive [52] [2].
- **No explicit authentication on socket** beyond the URL containing a session (in a more secure setup, you might include a token or verify the client's identity when upgrading).
- After connection, the client should expect JSON messages broadcast by the server. These messages correspond to `SessionStatus` updates currently. For example, whenever someone joins or leaves ready state, the server sends the latest `{"players": [...], "all_ready": ...}` as shown above. The server may also send other types of messages in the future (e.g., if game moves or chat were implemented, they could be sent similarly). But for now, it's specifically used for readiness/lobby updates.
- The server's implementation of the socket (`realtime.websocket_endpoint`) does not handle any messages from the client (it just reads and discards any text the client sends [130]). This means the communication is effectively one-way (server -> client) for this POC. The client might send pings or some keep-alive, but nothing is processed.
- **Disconnect**: If a client closes the socket (or loses connection), the server removes them from the active list [55]. It doesn't automatically notify others of this event via the readiness mechanism, but their disappearance could be inferred (for example, if the front-end wanted, it could call the `/clients` endpoint periodically to see who is still connected).

The WebSocket is crucial for providing a responsive UX: players see each other's status changes in real time. Without it, clients would have to poll `/status` or `/clients` constantly to detect changes, which is inefficient and slow. This publish-subscribe pattern on a per-session basis is a common design in multiplayer backends.

WebSocket Usage Example: - When Player A joins session X via the HTTP join endpoint, the server (as part of join) broadcasts an update. If Player B is already connected to session X's WebSocket, Player B's client

receives a message like: `{ players: [ {id A, ready: false} , {id B, ready: whatever} ], all_ready: false }`. Player A at this point would also connect its WebSocket and likely call `/status` to get the current players (which would show A and B). - When Player A toggles ready, the server broadcasts the new status. Both A and B get it through the socket and update their UI immediately. Perhaps the UI shows a green checkmark by A's name for B, and by A's own name as well. - If a new player C joins, A and B get a broadcast adding C (ready: false). When C connects their socket, they might get the current status either from the join response or an immediate call to status. From then on, they receive broadcasts too.

In short, the WebSocket endpoint is the **real-time bridge** that complements the REST API. The pattern is to use REST for actions (which also modify DB and state) and WebSockets to notify all clients of those actions' effects.

Additional Technical Insights

Finally, we cover some additional technical considerations in the backend solution: middleware, configuration, deployment, and error handling, which though not part of the core logic, are important for a complete understanding of the system.

Middleware and Static File Serving

The application does not employ custom middleware beyond what FastAPI and Starlette provide out-of-the-box. Common middleware like CORS, authentication, etc., are not configured explicitly (likely because the front-end is served from the same origin, and security was not a primary concern in this POC). If this were to be accessed from a different domain, one might add `CORSMiddleware`.

One key "middleware-like" component is the **StaticFiles mounting**. Using `app.mount("/", StaticFiles(directory="frontend", html=True), name="frontend")` and a second mount for the tiles images [15] effectively acts as a middleware that intercepts requests to certain paths (the root and `/tiles`) and serves files. This is provided by Starlette under the hood. The static file mount with `html=True` means that requests to undefined routes will return the `index.html`, enabling client-side routing if any. In this app, it simply ensures that going to the base URL opens the interface.

No other middleware (like sessions or compression) is shown. FastAPI's exception handler middleware is by default in place to catch unhandled exceptions and return HTTP 500.

Configuration and Environment

All configuration is centralized in `config.py`. As noted, `DATABASE_URL` is hard-coded for now [3] – it points to a Postgres instance on Render.com, which suggests the app was likely deployed to Render at some point. In a production scenario, one would use environment variables or a secrets manager to provide this URL. The config also sets up SQLAlchemy with `echo=True` for SQL logging (printing all SQL statements to console) [20], which is useful for debugging but would be noisy in production.

There are also asset path configurations (`ASSET_PATHS`) which imply the backend might serve or use static files for portraits/backstories [22], though the current code doesn't actively use these paths. This

forward-thinking addition indicates the architecture anticipates serving or processing static content related to game entities (perhaps the front-end could request an image by ID and the backend knows the path).

The app expects certain files and directories to exist (like `assets/seed/*.csv` and `frontend/`). If deploying, these need to be bundled with the app. The **requirements.txt** lists exact versions for dependencies [131], ensuring reproducibility of the environment. It includes FastAPI, Uvicorn, SQLAlchemy, Psycopg2 (DB driver), Pandas, and the websockets library. Notably, Gunicorn is not listed in requirements, which might mean on Render it was provided or installed separately, or they rely on Uvicorn for local runs. In any case, to run the app locally, one might use Uvicorn (`uvicorn main:app --reload`) which will serve on the default port 8000, whereas in production the `startup.sh` uses Gunicorn on port 5000 [78].

No configuration for logging is explicitly set up; the app uses simple `print` statements for logging important events (DB init, data load, etc.) [13] [132]. In a more advanced setup, one would integrate Python's logging module.

Deployment Strategy

The presence of `startup.sh` and the specific database URL suggests the app was deployed as a web service (possibly on Render.com). The deployment strategy is likely container-based or a simple Python environment where:

- The database is a cloud Postgres instance (the URL has a host `dpg-cvmm...` which looks like a managed DB host).
- The web service runs the FastAPI app. Render.com supports deploying FastAPI either via Gunicorn or their own mechanism. The script indicates using Gunicorn binding to all interfaces on port 5000, which is the default for some platforms.
- Gunicorn is started without specifying workers or an ASGI worker class in the script; possibly Render's default for FastAPI is to use `uvicorn.workers.UvicornWorker`. If not, this might actually run the app in a synchronous WSGI mode which FastAPI (ASGI) doesn't directly support. However, given that Uvicorn is installed, it's likely configured properly by the platform. In a custom Docker deployment, one would run something like `gunicorn -k uvicorn.workers.UvicornWorker -w 4 -b 0.0.0.0:5000 main:app`.
- There is no Dockerfile in the repo, so the deployment was possibly done by connecting the repo to the platform and specifying the start command (`bash startup.sh`). This script takes care of DB migrations (by dropping/creating – not typical but works for POC) and launching the server.
- No horizontal scaling considerations are mentioned. With a single instance, the global in-memory state for readiness is fine. In a multi-instance scenario, that state would need to be externalized (to a shared DB or cache) or else different clients could be connected to different instances and not share readiness info or WebSocket broadcasts. That's beyond the scope of this POC's architecture, which assumes one running instance.

Error Handling and Validation

Error handling in this backend is minimal but present in key places: - For expected error cases (resource not found, or duplicate actions), the code uses FastAPI's `HTTPException` to return appropriate HTTP status codes. For example, joining a session with a bad ID returns a 404 Not Found [95], and trying to leave when not connected returns 404 [103]. These ensure the client gets a clear error response instead of a generic 500.

- The labyrinth generation function validates input size and raises a Python `ValueError` if out of allowed range [56]. Currently this isn't caught and translated to an HTTP 400, so the client would receive a 500 with an internal error message. This is a minor gap; in a full implementation, you'd catch that in the endpoint and respond with a 422 or 400 telling the user the size is invalid. - Transactions: The use of session commit and rollback in `init_data.load_data` shows an effort to handle errors during the bulk insert [29]. If any insertion fails (say the CSV has bad data), it will rollback and print an error. The rest of the app doesn't explicitly catch DB errors (like failure to insert a duplicate primary key, etc.), but those are unlikely in normal use (client IDs are meant to be unique, session IDs are UUIDs generated to be unique). - The WebSocket operations are wrapped in try/except to catch disconnects or send errors [133]. The code simply passes on exception, meaning if a send fails (maybe the socket closed unexpectedly), it does nothing further. A possible improvement is to remove that socket from active connections if send fails consistently, but since `disconnect_from_session` will handle removal on actual disconnect, this is not critical. - There is no global exception middleware configured (FastAPI does have default handlers). If something truly unexpected happens in an endpoint (like a KeyError in code), FastAPI will return a 500 with a generic message by default. For development, the debug mode would show a traceback; in production, just an internal server error. Given this is a POC, this default is fine.

Security Considerations

Even though not explicitly asked, it's worth noting: - There is **no authentication or authorization** on any endpoint. All API calls are open. This means any user who can reach the service could potentially create or join sessions at will. In a contained environment (like a closed test or single-player scenario) this is acceptable, but in a multi-user scenario it could be problematic. Typically, adding at least a simple auth (even if just a password on session join or an API key) would be considered. - The `client_id` is the sole identifier for players and it's trusted blindly. For now, collisions are only handled in that duplicate join is prevented, but there's nothing stopping two different people from using the same client_id on different machines and effectively being treated as the same player by the server. A robust solution would tie client_id to a user account or a session cookie. - The database credentials are in plain text in the repo [3]. This is obviously not secure for a real deployment, but in a POC or if the database is ephemeral, it was convenient. Typically, these would be environment variables. (We redact credentials here, but in code they are visible which is a practice to avoid in real projects.)

Extensibility and Future Work

Though not part of the current implementation, the architecture sets the stage for future extensions: - **Game Play Logic**: With players, sessions, labyrinths, and readiness in place, the next step would be to handle actual game turns or moves. This could involve new endpoints or WebSocket messages for moving a player (which would update the `Player.player_x, player_y` in the DB and broadcast the move), or attacking, etc. The current structure could accommodate this by adding new routes (e.g., `/api/game_sessions/{session_id}/move`) and perhaps leveraging the WebSocket to broadcast moves. The use of SQLAlchemy and Pydantic means adding such features would fit in naturally. - **Scaling**: If needed, the app could be scaled by moving the global state to a shared store (like Redis) so multiple instances have the same view of readiness, and using a pub/sub mechanism for WebSocket broadcasts across instances. Alternatively, one might stick to a single instance given the scope. The architecture is monolithic but modular, so pulling out components (say, running the database separately, which it already does) or even separating the real-time component (though not necessary) could be possible. - **Improved Session Management**: Currently, session cleanup (destroy-all or manual deletion) doesn't remove labyrinths. One

could add a scheduled task or a cascade such that if a labyrinth has no sessions, it's removed. Also, if many sessions accumulate, listing all might become heavy; pagination or filters could be added to `/api/game_sessions`. - **Error and Validation**: Adding more thorough input validation (e.g., ensure seed format is valid hex or short string, ensure size is int, etc. – FastAPI/Pydantic already covers types but could do bounds) and better error messaging would improve robustness.

Conclusion

The mkssystems/Epsilon-POC-2 backend demonstrates a clean architecture for a multiplayer game backend prototype. It opts for a straightforward design using FastAPI for both synchronous REST and asynchronous WebSocket handling, a normalized relational database for persistence, and global in-memory state for ephemeral coordination. Each part of the codebase has a clear role, making it easy for developers to locate functionality (e.g., DB models vs. API routes vs. game logic). While certain POC shortcuts are present (no auth, DB resets, in-code secrets), the overall solution is well-structured for a technical audience to understand and extend. The documentation above should serve as a map of the system's components and how they work together to enable the game's core functionality: generating labyrinths and managing game sessions with multiple players in real time.

---

1 9 13 14 15 16 17 18 19 82 86 87 88 89 90 91 117 118 119 120 123 main.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/main.py

2 10 51 52 53 54 55 115 130 133 realtime.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/realtime.py

3 20 21 22 116 config.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/config.py

4 30 labyrinth.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/labyrinth.py

5 34 35 92 game_session.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/game_session.py

6 base.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/base.py

7 8 69 70 71 72 schemas.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/schemas.py

11 state.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/state.py

12 48 49 50 73 74 79 80 81 83 84 85 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 124 127 128 api.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/routes/api.py

23 24 session.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/db/session.py

25 26 27 28 29 121 132 init_data.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/db/init_data.py

31 32 33 tile.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/tile.py

36 37 player.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/player.py

38 39 122 mobile_client.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/mobile_client.py

40 41 game_entities.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/game_entities.py

42 43 equipment.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/equipment.py

44 45 skills.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/skills.py

46 47 specials.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/models/specials.py

56 57 58 59 60 61 62 63 64 65 66 67 68 corrected_labyrinth_backend_seed_fixed.py
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/utils/corrected_labyrinth_backend_seed_fixed.py

75 76 93 125 126 129 index.html
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/frontend/index.html

77 78 startup.sh
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/startup.sh

131 requirements.txt
https://github.com/mkssystems/Epsilon-POC-2/blob/35e28f8356880a7b32c78af2701888e5e5f8b707/requirements.txt