Part 1 — Multithreading in the Simulator

Overview

The simulator is fundamentally concurrent: multiple airports generate flight requests, planes depart, travel, land, and are serviced simultaneously. To achieve realism without blocking the JavaFX UI thread, I used a combination of fixed and cached thread pools, BlockingQueues, and careful synchronization.

Thread Creation

- SimulationController is the orchestration class.
    - It creates a fixed thread pool with one dispatcher thread per airport.
        - These dispatcher threads continuously poll their airport's request queue and attempt to acquire an available plane.
        - Once a plane is found, they schedule a "flight" task.
    - It also creates a cached thread pool for servicing tasks.
        - Each plane that lands is passed to StandardPlaneServicing.service(...), which runs asynchronously in this pool.
- JavaFX Application Thread is reserved for rendering (UI, logs, stats). No background work touches JavaFX objects directly, updates are marshalled back via Platform.runLater(...).

Thread Communication

Threads communicate indirectly using shared blocking queues:

- Each airport has:
    - A BlockingQueue<FlightRequest> (produced by the simulator, consumed by its dispatcher thread).
    - A BlockingQueue<Plane> (produced when a plane becomes available, consumed by dispatchers when scheduling a flight).
- When a dispatcher selects a plane and a request, it updates global simulation state (plane enters "in flight" map). Once the flight completes, it enqueues the plane back into the servicing pool.

Thus, communication happens through queues and concurrent data structures, not direct message passing.

Avoiding Race Conditions and Deadlocks

- All shared state (Map<Plane, Airport> inFlight, airport queues) uses thread-safe collections (ConcurrentHashMap, BlockingQueue).
- The design avoids nested locks entirely; no manual synchronization is required beyond what the concurrent collections provide.
- Flight state transitions are atomic: a dispatcher thread removes from one queue, inserts into another. No partial state is ever visible.

Thread Termination

- The End button signals SimulationController to:
    - Interrupt dispatcher threads via shutdownNow() on the fixed thread pool.
    - Stop accepting new servicing tasks.
    - Planes already in servicing complete naturally.
- A stop() method is also bound to the JavaFX stage close handler, ensuring clean shutdown before exit.
- Importantly, termination leaves the UI snapshot frozen with the last known state (airports and planes still drawn).

Part 2 — Extending into a Multiplayer Game

Important Non-Functional Requirements (NFRs)

1. **Scalability**
    - Many players online simultaneously, each controlling an airport.
    - Backend must handle thousands of concurrent flight requests and transactions.
2. **Reliability / Availability**
    - Downtime directly affects players' experience and revenue.
    - Continuous uptime and graceful failover are critical.
3. **Security**
    - Protect user accounts, in-game assets, and financial transactions (real-money purchases).
    - Prevent cheating (tampering with flight schedules, free purchases, etc.).
4. **Performance / Responsiveness**
    - Flights, purchases, and multiplayer interactions must feel real-time.
    - Long delays (e.g., 5s before a plane shows on another player's screen) would ruin gameplay.
5. **Maintainability / Extensibility**
    - New aircraft models, facilities, and routes released over years.
    - Architecture must allow incremental feature deployment without disrupting running services.
6. **Regulatory / Compliance**
    - Real-money transactions require compliance with financial and data protection regulations (e.g., PCI-DSS, GDPR).

Architectural Approach

I propose a service-oriented architecture (SOA) with the following elements.

- **Game Clients (Desktop/Mobile)**
  Render UI, handle input, send commands (e.g., "buy aircraft", "dispatch plane") to backend via APIs.
  Lightweight - no direct control over shared game state.
- **Game Server Cluster**
  - **Simulation Service**: Manages global state of airports, planes, routes. Uses event queues (similar to current simulator design, but distributed).
  - **Player Service**: Handles player authentication, profiles, leaderboards.
  - **Economy/Store Service**: Manages purchases, balances, and monetisation.
  - **Messaging Service**: Publishes game events (e.g., flight departed) to subscribed clients in real-time (WebSockets or pub/sub).
- **Database Layer**
  - **Relational DB** for transactions, player accounts, purchases.
  - **In-memory stores (Redis)** for fast state updates (current flights, positions).
- **Payment Gateway Integration**
  - Secure APIs for handling real-money transactions.

Architecture Diagram

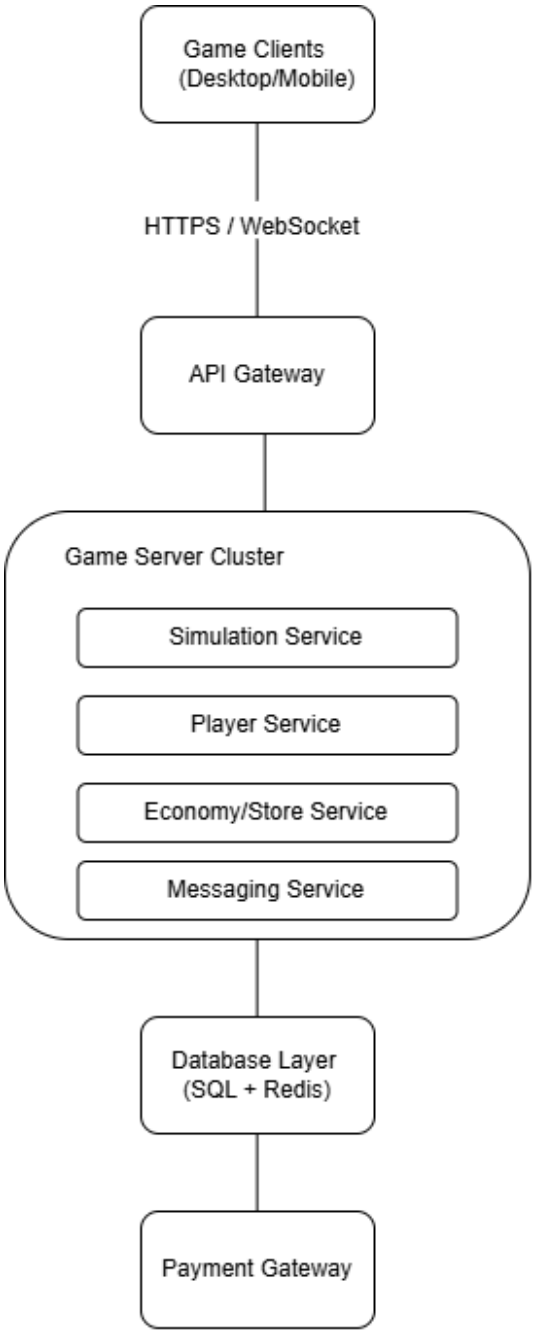

Figure 1

How the Architecture Addresses NFRs

- **Scalability**: Independent services can scale horizontally (e.g., multiple simulation servers sharded by region).
- **Reliability**: If one service fails (e.g., store), others remain operational. Messaging ensures recovery after transient outages.
- **Security**: Clear separation of financial subsystem (Economy/Store Service) reduces attack surface.
- **Performance**: In-memory caches and event-driven updates keep latency low.
- **Maintainability**: New aircraft or features = new microservices or APIs, no need to rewrite whole system.

Disadvantages

- **Complexity**: Distributed systems require expertise (orchestration, monitoring, deployment pipelines).
- **Latency risk**: Cross-service communication adds overhead.
- **Cost**: Running and maintaining multiple services, databases, and real-time infra is expensive.