

Draw It or Lose It — Web-Based Edition

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc208932256)

[Table of Contents 2](#_Toc208932257)

[Document Revision History 2](#_Toc208932258)

[Executive Summary 3](#_Toc208932259)

[Requirements 3](#_Toc208932260)

[Design Constraints 3](#_Toc208932261)

[System Architecture View 4](#_Toc208932262)

[Domain Model 4](#_Toc208932263)

[Evaluation 5](#_Toc208932264)

[Recommendations: 7](#_Toc208932265)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/16/25 | Angelo Salas | Initial completion of template for Project One (Executive Summary, Requirements, Design Constraints, Domain Model, Evaluation, Recommendations). |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room (TGR) will expand its Android title, Draw It or Lose It, into a web-based, multi-platform application. The solution uses a centralized, thread-safe GameService (Singleton) to manage Games, Teams, and Players, guaranteeing unique names and single-instance coordination. Uniqueness checks use Iterator traversal of in-memory collections. The first release runs as a stateless web service with browser clients; future phases will add persistence, authentication, and horizontal scaling. This design minimizes coupling, maximizes portability, and allows CTS to deliver a functional prototype quickly while keeping a clean path to production.

## Requirements

Business

* Support one or more teams per game
* Support multiple players per team
* Ensure game and team names are unique and discoverable
* Allow only one authoritative instance of the game coordinator

Technical

* Provide a web-based interface accessible on common desktop and mobile platforms
* Enforce name uniqueness across the system
* Expose simple APIs to create and query games, teams, and players
* Be thread-safe for concurrent users
* Prefer portable, standard technologies suitable for later deployment behind a Java web stack and a database

## [Design Constraints](#_2et92p0)

* **Environment:** Distributed web setting with stateless, concurrent requests and shared server state.
* **Constraints:**
* Single instance of **GameService** (**Singleton**) for identity and rule enforcement
* Concurrency control for mutations and **atomic ID generation**
* **Unique naming** — O(n) iterator search in prototype; enforce **DB constraints** later
* **Low latency** targets → in-memory data for the prototype
* **Portability** across macOS/Linux/Windows servers and modern browsers
* **Security** phased in: input validation, authentication, transport encryption (TLS)
* **Implication:** Favors a **service-oriented design** with clear separation between **presentation (web client)**, **application service (GameService)**, and **domain model (Entity/Game/Team/Player)**.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

## [Domain Model](#_8h2ehzxfam4o)

The UML defines an abstract base class Entity with id and name. Game, Team, and Player each extend Entity. A Game composes one or more Teams; each Team composes multiple Players. Inheritance captures shared identity/name attributes and behavior; composition models whole–part relationships. Encapsulation keeps collections private and mutations funneled through service methods. The Singleton GameService centralizes creation, uniqueness checks (Iterator pattern), and lookups. These OO principles allow reuse, clear responsibilities, and efficient enforcement of system rules.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## 

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Capable of hosting Java/Node/Ruby web apps (e.g., via Homebrew). Strengths: developer-friendly UI, Unix base. Weaknesses: costly hardware, limited headless/server SKUs, and uncommon in production; fewer ops automations compared to Linux. Best for development/demo, not primary production hosting. | Industry-standard for servers. Strengths: performance, stability, broad tooling (systemd, containers), first-class support for Java/Node, rich CI/CD ecosystem, low cost. Weaknesses: admin expertise required; distro variance. Recommended for production hosting at scale. | Strong IIS/.NET support and good ops tooling (AD/Group Policy). Java/Node run well too. Strengths: enterprise integration, GUI admin options. Weaknesses: licensing cost, heavier resource footprint than minimal Linux builds. Appropriate where Windows ecosystem is required. | Not suitable for hosting the server. Mobile devices are resource- and connectivity-constrained and sandboxed; they should operate as clients consuming the web API. |
| **Client Side** | Supporting Mac clients means ensuring modern browser compatibility (Safari, Chrome, Firefox). Costs are modest if the UI is web-based; extra cost arises only if a native macOS app is also demanded. | Linux client support focuses on Chrome/Firefox compatibility. Minimal extra cost with a standards-based web UI. Native desktop apps are optional and would increase complexity. | Windows clients primarily use Edge/Chrome/Firefox. A responsive web app minimizes cost and time. Optional native app (UWP/Win32) increases scope and maintenance. | Mobile clients should be supported via responsive web UI for lowest cost. If native features (push, offline, sensors) are needed, consider a cross-platform framework (Flutter/React Native) or a PWA to balance effort and capability. |
| **Development Tools** | Languages: Java (JDK 17+), JavaScript/TypeScript. Tools: IntelliJ IDEA/Eclipse/VS Code, Maven/Gradle, Node.js for front end, Docker for containerization. Safari/Chrome dev tools for testing. | Languages: Java, JS/TS. Tools: OpenJDK, Maven/Gradle, Node.js, Nginx/Apache for reverse proxy, Docker/Podman, Kubernetes for orchestration, Git, CI/CD runners (GitLab/GitHub Actions). | Languages: Java, JS/TS, optionally .NET for enterprise integration. Tools: IntelliJ/Eclipse/VS Code, Maven/Gradle, Node.js, IIS or Nginx, Docker Desktop/WLS2, PowerShell, Azure DevOps. | Web-first approach requires HTML/CSS/JS with responsive design. For native or cross-platform: Android Studio (Kotlin/Java), Xcode (Swift) or Flutter/React Native toolchains. PWAs for push/offline. |

## [Recommendations:](#_2o15spng8stw)

1. Operating Platform:  
     
   Linux-based servers are recommended for production deployment due to cost efficiency, stability, and first-class support for web application frameworks and containerization. Windows Server remains a valid choice in enterprise Microsoft environments but carries licensing costs. macOS should be limited to development/demos, not production hosting. Mobile devices are strictly client-side consumers of the web application.

Use a **Linux-based, containerized web platform** for production (e.g., Docker containers orchestrated by Kubernetes or a managed PaaS). This gives The Gaming Room portability, scalability, and low operating cost while supporting a standards-based REST/WebSocket API that serves **all** client environments (Windows, macOS, Linux, iOS, Android) via the browser.

1. Operating Systems Architectures:  
     
   Adopt a three-tier, stateless web architecture:  
     
   • Client tier (browser/PWA): Responsive web UI that runs on any OS with a modern browser.  
   • Application tier (Java/Dropwizard): Stateless REST API with optional WebSockets for real-time updates. GameService singleton coordinates logic across multiple server instances behind a load balancer.  
   • Data tier: Managed relational database (PostgreSQL) with enforced constraints, with Redis as optional cache.  
     
   This separation supports scalability, portability, and minimized vendor lock-in.

Adopt a **three-tier, stateless web architecture**:

* **Client tier (browser/PWA):** Responsive web UI that runs on any OS with a modern browser.
* **Application tier (Java):** A stateless REST API with WebSocket/SSE for live game updates. Core domain logic is in a GameService singleton within the process; multiple app instances can run behind a load balancer.
* **Data tier:** Managed relational database (e.g., PostgreSQL) with unique constraints for game/team names; optional Redis for caching session/lightweight game state.  
  This architecture separates concerns, supports horizontal scaling, and minimizes OS lock-in.

1. **Storage Management**:

* **Primary store:** **PostgreSQL** with normalized schemas for Game, Team, and Player, enforcing **unique indexes** on names and **foreign keys** for relationships.
* **Caching:** **Redis** for hot keys (game lobbies, active rounds, leaderboards) to reduce DB load.
* **Backups & retention:** Automated daily snapshots, point-in-time recovery, and environment-specific retention policies.
* **Migrations:** Versioned schema migrations (Liquibase/Flyway) integrated into CI/CD.

1. **Memory Management**:

* **JVM (Java 17+):** Use the default generational GC (e.g., G1) which handles many short-lived objects typical of request/response traffic.
* **Design practices:** Keep entities lightweight, prefer IDs over large object graphs, and clean up references when games end to prevent leaks.
* **Tuning:** Right-size heap per pod/container, monitor GC metrics, and set upper memory limits to avoid container OOM kills.
* **Concurrency:** Use immutable DTOs where possible and thread-safe collections only where needed to minimize contention.

1. **Distributed Systems and Networks**:

* **Protocol:** Expose REST/JSON for CRUD and **WebSockets or Server-Sent Events** for real-time drawing/guess updates.
* **Scalability:** Run **multiple stateless app instances** behind a load balancer; sticky sessions not required if state is externalized (DB/Redis).
* **Resilience:** Health checks, timeouts, retries with back-off, and circuit breakers; graceful degradation when Redis/DB is slow.
* **Connectivity:** Use CDN for static assets; consider edge caching for latency. Handle intermittent client connectivity by resyncing state on reconnect and idempotent server endpoints.

1. **Security**:

* **Transport:** Enforce **TLS (HTTPS)** end-to-end.
* **AuthN/AuthZ:** OAuth2/OIDC for user authentication; role-based access for admin/ops endpoints.
* **Data protection:** Store minimal PII; hash & salt passwords if stored (or outsource to an identity provider). Encrypt secrets at rest (KMS) and use secrets managers for config.
* **Input & session safety:** Validate/sanitize all inputs; protect against XSS/CSRF; use secure cookies with SameSite/HttpOnly; rate-limit login and game actions.
* **Ops hygiene:** Least-privilege IAM for services, regular patching, dependency scanning, audit logging, and anomaly alerts (SIEM).

**Recommendations:**

**Operating Platform:**

Linux (Ubuntu LTS) is recommended as the primary server platform, deployed on a managed cloud environment (AWS, Azure, or GCP). This provides cost efficiency, high stability, containerization support, and wide compatibility. Containers (Docker + Kubernetes) enable scalable, portable deployments that can serve all client platforms (Windows, macOS, Linux, iOS, Android).

**Operating System Architectures:**

The Linux kernel uses a monolithic but modular design with support for loadable modules, process scheduling, memory management, and advanced networking. Its Virtual File System (VFS) allows compatibility with multiple file systems. Namespaces and cgroups support container isolation for scalability and resource management. The proposed architecture will follow a three-tier model:  
• Client tier: Web-based responsive front end accessible from any modern browser.  
• Application tier: Java/REST API layer with optional WebSocket for real-time updates.  
• Data tier: Managed PostgreSQL with Redis caching for session and leaderboard data.

**Storage Management:**

A managed PostgreSQL database will store persistent game data (users, games, teams, scores), while Redis will handle temporary or frequently accessed data such as session tokens and live scores. Object storage (e.g., S3) will host image assets and static content. Backup schedules, encryption at rest, and replication across availability zones will protect data integrity and availability.

**Memory Management:**

Linux uses virtual memory, paging, and process-level memory isolation. The JVM’s G1 garbage collector efficiently manages short-lived objects from frequent user interactions. Containers will have defined memory limits via cgroups to prevent resource contention. Object pooling and caching will further improve memory performance for Draw It or Lose It.

**Distributed Systems and Networks:**

Draw It or Lose It will operate as a distributed web application using REST APIs and WebSockets for communication. Multiple stateless application instances will run behind a load balancer to ensure scalability and resilience. Redis pub/sub will handle real-time updates. Network reliability is maintained through retries, health checks, and redundancy across availability zones. Client reconnect logic will recover game state after connectivity loss.

**Security:**

All communication will be protected with TLS 1.3 encryption. Authentication and authorization will use OAuth2/OIDC standards. User data will be encrypted at rest using managed key services. Input validation, rate limiting, and Web Application Firewall rules will prevent injection and denial-of-service attacks. Least-privilege IAM policies, secrets management, and regular vulnerability scanning will ensure platform-wide security.