

# The Gaming Room — Draw It or Lose It (Web) **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[The Gaming Room — Draw It or Lose It (Web) **CS 230 Project Software Design Template** 1](#_Toc210397036)

[Table of Contents 2](#_Toc210397037)

[Document Revision History 2](#_Toc210397038)

[Executive Summary 3](#_Toc210397039)

[Requirements 3](#_Toc210397040)

[Design Constraints 3](#_Toc210397041)

[System Architecture View 4](#_Toc210397042)

[Domain Model 4](#_Toc210397043)

[Evaluation 5](#_Toc210397044)

[**Recommendations** 9](#_Toc210397045)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | |  | | --- | |  |  |  | | --- | | 09/20/2025 | | Rimon Hamo | Initial submission for The Gaming Room: filled Executive Summary, Requirements, Design Constraints, System Architecture, Domain Model, Evaluation, and Recommendations. |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room wants to expand *Draw It or Lose It* currently Android-only into a **web-based, multi-platform game** accessible from modern browsers on Mac, Linux, Windows, and mobile devices. The game must support multiple teams per game, multiple players per team, and enforce unique names (game, team, player).

**Proposed approach:** build a portable Java domain model (Game, Team, Player) that inherits from a shared Entity base class (id, name) and manage state through a **Singleton** GameService in the prototype. Uniqueness is enforced using the **Iterator** pattern in add/get operations. For production, promote this core to a **3-tier web architecture**: stateless Java service (REST + WebSockets), backed by a relational database with unique constraints and optional in-memory cache. This plan meets current requirements, controls cost and risk and provides a clean path to scale.

## Requirements

***Business***

* *Offer the game on browsers across desktop and mobile platforms.*
* *Simple onboarding and name-availability checks to ensure fair play.*
* *Reliable rounds (1 minute each) with responsive updates.*

***Functional***

* *Create/read* ***Games****,* ***Teams****,* ***Players*** *with system-generated unique IDs.*
* *Enforce* ***unique game names globally****,* ***unique team names within a game****, and* ***unique player names within a team****.*
* *Provide lookups by name to check availability.*

***Technical***

* ***Singleton*** *service instance during the prototype for consistent in-memory state.*
* ***Iterator*** *pattern for list traversal and uniqueness checks.*
* *Separation of concerns: domain logic independent of transport/storage so it ports to REST + DB later.*
* *Basic error handling and concise inline comments for maintainability.*

## [Design Constraints](#_2et92p0)

* **Single in-memory instance (prototype):** only one GameService should exist, simplifying identity and state; later replaced by database-backed constraints in distributed deployments.
* **Concurrency & uniqueness:** duplicates must be prevented; in memory via iteration and case-insensitive comparison; later via **DB unique indexes** and transactional guarantees.
* **Stateless scaling (future):** session state shouldn’t live on a single node; move durable state to a DB and ephemeral round state to a cache (e.g., Redis) to scale horizontally.
* **Portability:** core domain code must remain framework-agnostic so it can run as a console app now and a web service later.
* **Security & privacy (web):** all external traffic must be TLS-protected; validate inputs; authenticate users; protect secrets; log and audit.
* **Performance & cost:** prototype uses lists (O(n) checks) which is fine at small scale; production adds indexes, caching, and CDN where needed.

## [System Architecture View](#_ilbxbyevv6b6)

**Prototype:** single-process Java application with a GameService **Singleton** and in-memory lists.

**Production (3-tier path):**

* **Client tier:** browser UI (desktop/mobile).
* **App tier:** stateless Java service (e.g., Spring Boot) exposing **REST** endpoints for CRUD and **WebSockets** (or SSE) for real-time round events; containerized and scaled horizontally behind a load balancer.
* **Data tier:** **PostgreSQL** (persistent entities with unique constraints) plus optional **Redis** (ephemeral round state, timers, leaderboards).  
  Add CDN for static assets, centralized logging/monitoring, and CI/CD for repeatable builds.

## [Domain Model](#_8h2ehzxfam4o)

**Classes & relationships**

* **Entity** (base): id: long, name: String.
* **Game** extends **Entity**: owns List<Team>, provides addTeam(name) / getTeam(name).
* **Team** extends **Entity**: owns List<Player>, provides addPlayer(name) / getPlayer(name).
* **Player** extends **Entity**.  
  Relationships are compositional: Game 1..\* → Team 1..\* → Player.

**OOP principles**

* **Inheritance:** common identity and naming in Entity.
* **Encapsulation:** collections are private; adds/gets enforce invariants (uniqueness).
* **Composition:** games contain teams; teams contain players.
* **Polymorphism (lightweight):** consistent toString()/identity via base type.

**Design patterns**

* **Singleton:** GameService ensures one authoritative in-memory instance (prototype constraint).
* **Iterator:** used in add/get to traverse collections and enforce uniqueness readably.

**"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)

**Server Side (hosting & licensing)**

**Linux**

* Hosting: First-class choice for production web apps. Native fit for containers/Kubernetes; rich ecosystem (NGINX/Apache, PostgreSQL, Redis).
* Licensing/Cost: Community distros (Ubuntu/Debian) $0 license. RHEL offers paid subscriptions for enterprise support.
* Notes: Easiest path for cloud VMs and managed K8s; vast DevOps tooling and community expertise.

**Windows**

* Hosting: Viable using IIS or containerized services; strong AD/PowerShell/enterprise integration.
* Licensing/Cost: Windows Server licenses + CALs (cost varies by cores/CALs/edition). Typically, higher TCO than Linux.
* Notes: Works fine for Java stacks, but Windows containers/orchestration are less ubiquitous than Linux.

**macOS**

* Hosting: Suitable for development/CI and demos; not recommended as a production web host (limited cloud/mac hosting options).
* Licensing/Cost: No separate “server” SKU; premium Apple hardware required.
* Notes: Great for building/signing Apple artifacts; not for scalable web serving.

**Mobile** Devices (Android/iOS)

* Hosting: Not appropriate for servers (sandboxed, battery/CPU constrained, behind NAT).
* Role: Client only.

**Client Side (browser & device compatibility)**

**All desktop OS (Mac, Linux, Windows)**

* Deliver a responsive web UI (HTML/CSS/JavaScript).
* Test matrix: Chrome/Edge (Chromium), Firefox, Safari (macOS), mixed DPIs/HiDPI/Retina, keyboard + pointer.
* Accessibility: Target WCAG 2.1 AA (color contrast, focus order, ARIA where needed).

**Mac**

* Runs in Safari/Chrome/Firefox. Watch for WebKit quirks and Retina scaling.

**Linux**

* Strong Chrome/Firefox support across distros. Verify fonts/codecs/GPU driver differences.

**Windows**

* Largest desktop base (Edge/Chrome). Validate high-DPI scaling, corporate proxies, and AV restrictions.

**Mobile (Android/iOS)**

* Primary surface for many players. Optimize for touch targets, small screens, and variable networks.
* Implement PWA features (installability/offline cache via Service Workers) as a low-cost “app-like” path.
* Extra QA for device fragmentation (Android versions), and iOS Safari behaviors.

**Cost/Time/Expertise:**

* A single web team delivers one responsive codebase. Time & cost center on cross-browser/mobile QA (Chrome/Edge, Firefox, Safari desktop/iOS, Android Chrome), accessibility (WCAG 2.1 AA), and performance (lazy loading, Lighthouse). Device coverage can be scaled with cloud device labs (e.g., BrowserStack).

**Development Tools (languages, IDEs, licensing, team impact)**

**Common stack (all desktop OS)**

* Languages/Frameworks: Java (server), HTML/CSS/JS (client).
* Server build: OpenJDK (Temurin), Maven/Gradle.
* Client build: Node.js + npm/yarn; optional framework (React/Vue).
* IDE/Editors: VS Code (free), IntelliJ IDEA Community (free) or IntelliJ Ultimate (paid), Eclipse (free).
* Containers: Docker Desktop (free for personal/education; Business subscription may apply for large orgs).
* APIs/Testing: Postman/cURL, JUnit, Jest/Lighthouse.
* Team impact: One cross-functional web team can own browser client + Java service.

**Linux (dev/CI)**

* OpenJDK, VS Code/IntelliJ, Maven/Gradle, Docker or Podman, kubectl/Helm, NGINX, PostgreSQL/Redis.
* Ideal for CI/CD runners and K8s workflows. Tooling largely license-free.

**Windows (dev)**

* OpenJDK, VS Code/IntelliJ, Maven/Gradle, Docker Desktop with WSL2, Git, PowerShell.
* Optional IIS/SQL Server if needed. Edge DevTools for debugging.

**macOS (dev)**

* Same Java/Node/Docker stack; Homebrew for setup.
* Required later if building/signing native iOS apps.

**Mobile (if native later not required for web client)**

* Android Studio (free), Xcode (requires Apple Developer Program $99/yr for device signing).
* Alternative cross-platform: Flutter/React Native (optional).
* Device lab services (e.g., BrowserStack) for broad testing.

**Licensing notes:**

* Docker Desktop (free for personal/education; business plans may apply). IntelliJ Community (free) / Ultimate (paid, optional). Apple Developer Program $99/yr if doing native iOS later.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Suitable for development and demos but rarely used for production hosting. Pros: Unix tooling, stable, easy local setup. Cons: higher hardware cost, few cloud/macOS hosting options, limited automation at scale. Good for CI of Apple builds; not ideal for running web servers long-term. | **Best choice for hosting.** Stable, secure, cost-effective, and the default target for containers/Kubernetes. Huge ecosystem (NGINX, PostgreSQL, Redis), first-class observability. Requires basic Linux admin skills and consistent distro standards, but scales cleanly from single VM to clusters. | Viable with IIS/Windows Server; strong AD/enterprise integration and PowerShell. Works fine for Java stacks, but licensing adds cost and Windows containers/orchestration are less ubiquitous than Linux. Heavier footprint on identical hardware. | Not appropriate as servers (sandboxed, battery/CPU constrained, behind NAT/cellular networks). Use phones/tablets strictly as **clients**; server workloads belong in the cloud. |
| **Client Side** | Modern browsers (Safari/Chrome/Firefox) run the web client natively. Minimal extra cost beyond responsive UI. Do QA for Safari/WebKit quirks and Retina scaling. Optional desktop wrapper (Electron) if needed later. | Chrome/Firefox support is strong across distros. Low incremental effort; just verify fonts, codecs, and GPU driver differences. Great for kiosk/education deployments. | Largest desktop base (Edge/Chrome). Plan QA for high-DPI scaling, corporate proxies, and antivirus restrictions. Otherwise, straightforward web delivery keeps cost/time low. | Primary surface optimize for responsive layouts, touch targets, and variable networks. Consider PWA features (installability/offline cache). Extra QA for device fragmentation (Android versions, iOS Safari behaviors), small screens, and performance (lazy loading). |
| **Development Tools** | JDK (OpenJDK/Temurin), IntelliJ/Eclipse/VS Code, Maven/Gradle, **Docker Desktop**, Node.js + npm/yarn (for front end), Git, Postman/cURL. Homebrew for setup. If you later build native iOS apps, add **Xcode**. | OpenJDK, IntelliJ/Eclipse/VS Code, Maven/Gradle, **Docker/Podman**, kubectl, NGINX, PostgreSQL/Redis, Git, cURL. Ideal for CI/CD runners and Kubernetes-based deployments. | JDK, IntelliJ/VS Code, Maven/Gradle, **Docker Desktop with WSL2** for Linux-like workflows, Git, Postman, PowerShell. Optional IIS/SQL Server for Windows-centric stacks; Edge DevTools for debugging. | For the **web client**: HTML/CSS/JavaScript with a framework (React/Vue), Service Workers for PWA, Lighthouse/Chrome DevTools remote debugging, **Safari Web Inspector** (iOS) and **ADB** (Android). If you later go **native/cross-platform**: **Android Studio** (Java/Kotlin), **Xcode** (Swift), or **Flutter/React Native**; cloud device labs (e.g., BrowserStack) for broad testing. |

**Recommendations**

1. Operating Platform

For production deployment, *Draw It or Lose It* should run on a Linux (Ubuntu Server LTS) platform hosted in a cloud environment such as AWS EC2 or Elastic Kubernetes Service (EKS).  
Linux is ideal because it offers exceptional stability, performance, and open-source flexibility with no licensing cost. Its kernel is optimized for multithreaded workloads and container orchestration, which makes it perfect for Java-based microservices.  
Docker containers orchestrated with Kubernetes allow horizontal scaling, rolling updates, and load balancing across nodes. This ensures consistent uptime and rapid response under high concurrency, while keeping operational costs low.

2. Operating System Architectures

Ubuntu Linux uses a monolithic kernel architecture that integrates process, file-system, device, and memory management inside the kernel space, enabling direct and efficient system calls.  
Key components:

* Kernel Mode: Manages CPU scheduling, virtual-memory paging, and inter-process communication.
* User Mode: Hosts the application services (e.g., the *Draw It or Lose It* backend) inside isolated Docker containers that use Linux namespaces and cgroups for resource limits.
* System Call Interface: Allows safe transitions between user and kernel space for file, network, and memory operations.

This design supports a three-tier microservice architecture:

1. Client Tier: Browser UI communicating via REST API and WebSockets.
2. Application Tier: Java service logic deployed in stateless containers.
3. Data Tier: Persistent and cache layers (PostgreSQL + Redis).

Together, these layers provide strong modularity, portability, and maintainability.

3. Storage Management

A hybrid storage strategy maximizes both durability and performance: Relational Storage (PostgreSQL) stores persistent entities such as players, teams, and scores. PostgreSQL supports ACID transactions, foreign-key constraints, and unique indexes that enforce naming rules (e.g., unique game and team names). In-Memory Cache (Redis) holds transient round data, session tokens, and leaderboards for real-time performance. Expiration (TTL) policies prevent stale data buildup. Object Storage (AWS S3 or Azure Blob) stores static media (avatars, artwork, and logs). Buckets are versioned and replicated for durability. Backup & Recovery: Automated daily snapshots, point-in-time recovery, and verified restore tests guarantee resilience against data loss. This layered approach ensures efficient file-system operations, fast access times, and reliable data consistency.

4. Memory Management

The Draw It or Lose It application leverages Linux’s robust memory management capabilities to ensure efficient performance. On the server side, the Java Virtual Machine (JVM) on Linux utilizes virtual memory with demand paging and caching to optimize I/O operations. The G1 garbage collector automatically reclaims unused Java objects, with heap sizes tuned using -Xms and -Xmx flags to balance memory usage. Connection pools and thread-pool reuse minimize object creation overhead, while Linux cgroups enforce container-level memory caps to isolate workloads and prevent resource contention. The page cache accelerates repeated disk reads, and swap usage is minimized to control latency. On the client side, browsers rely on JavaScript garbage collection and modern memory optimization techniques, avoiding long-lived DOM objects and releasing event listeners when users exit the game. These practices ensure predictable performance and prevent memory leaks or crashes across the application’s multi-platform deployment.

5. Distributed Systems and Networks

To enable cross-platform play, Draw It or Lose It functions as a distributed system connected via the internet. Communication is facilitated through REST/JSON for standard operations, such as creating or retrieving game, team, and player data, and WebSockets for real-time interactions, including drawing, guessing, and score updates. A reverse proxy, such as NGINX or HAProxy, provides load balancing to route traffic evenly among containerized Java services, ensuring scalability. Resilience is achieved through health checks, retry logic, circuit breakers, and exponential backoff to mitigate transient outages. Critical game state, such as player scores and team configurations, is committed to PostgreSQL for consistency, while ephemeral data, like round timers and session tokens, resides in Redis for low-latency access. Fault tolerance is ensured by Kubernetes, which automatically restarts failed containers, maintaining uninterrupted gameplay. This model supports horizontal scaling, allowing millions of concurrent players across browsers and devices with minimal downtime.

6. Security

Security is integral to Draw It or Lose It, ensuring user information is protected across all platforms and for all clients. All traffic is secured in transit using TLS 1.3 for HTTPS and WebSocket Secure (WSS) connections, with HTTP Strict Transport Security (HSTS) and secure cookies enforcing encrypted sessions. Data at rest, including volumes and object storage, is encrypted with AES-256 using managed keys via a cloud key management service (KMS). Authentication and authorization are handled through OAuth 2.0 and OpenID Connect (OIDC) for secure sign-in, with role-based access controls distinguishing players, admins, and moderators. Application-level protections include input validation, parameterized queries, CSRF tokens, and strict CORS policies to prevent injection and cross-site attacks. Credentials and API keys are stored in cloud secret managers, never in source control. Centralized logging and alerts via ELK stack or CloudWatch detect anomalies, while GDPR and CCPA compliance is achieved by minimizing personally identifiable information (PII) and providing user data-deletion controls. These measures ensure confidentiality, integrity, and availability across web, Android, and iOS platforms.

Summary

This architecture leverages Linux, PostgreSQL, Redis, Docker, and Kubernetes to provide a scalable, secure, and efficient environment for *Draw It or Lose It*.  
It meets all client requirements real-time gameplay, multi-platform access, fault tolerance, and user security while ensuring long-term maintainability and cost efficiency.