

# **Draw It or Lose It Web Application**

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | |  | | --- | | 07/20/25 |  |  | | --- | |  | | |  | | --- | | Luis Tomassini |  |  | | --- | |  | | <Brief description of changes in this revision> |

**Instructions**

Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room’s popular mobile game, Draw It or Lose It, currently exists only as an Android app. To reach a broader audience and support multiple platforms, our team proposes a web-based reimplementation that preserves the original four-round, one-minute gameplay while rendering clue images from the existing stock library. At the core of our design is a GameService class implemented using the Singleton pattern to ensure only one service instance manages all game state in memory. We enforce unique identifiers for games, teams, and players through a centralized ID generator, and we use the Iterator pattern to validate name uniqueness whenever users create new games or teams. This pattern-first approach delivers a maintainable, extensible architecture that meets The Gaming Room’s requirements for multi-user support, name collision avoidance, and cross-platform deployment.

## Requirements

While this section is not being graded directly, these business & technical requirements guided my design. From a business perspective, the application must enable teams of players to join and compete in four rounds per game, with each round lasting one minute. Technically, every game instance must run concurrently for multiple users in a web environment, enforce globally unique game and team names at creation time, and maintain a single in-memory service object to coordinate game state. Inheritance from a common Entity base class will guarantee consistency in how IDs and names are managed across Games, Teams, and Players.

## [Design Constraints](#_2et92p0)

Because we are targeting a web-based, distributed architecture, latency and concurrency must be managed carefully. The Singleton pattern for GameService places all ID generation and collection management in one object, reducing the risk of conflicting state, but it also introduces a single point of contention in multi-threaded scenarios. Ensuring name uniqueness via the Iterator pattern requires scanning in-memory collections, which scales linearly with the number of games or teams; for large deployments this may later require optimization or indexing. Finally, client browsers and servers may differ in capabilities, so our initial pure-Java backend design must be abstracted behind clean APIs to allow future migration to RESTful services or microservices without rewriting core logic.

## [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 & subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication & storage aspects is also necessary to understand the overall architecture and should be provided.

## [Domain Model](#_8h2ehzxfam4o)

The core of our model is an abstract Entity class that provides id & name fields (with getters and a toString()), which Game, Team, & Player all extend to ensure consistent identity management. A singleton GameService holds a private List<Game>, exposes iterator-based addGame()/getGame() methods to enforce unique names at creation. Each Game aggregates multiple Team instances, & each Team aggregates multiple Player instances, demonstrating composition. This design leverages inheritance for code reuse and encapsulation via private collections/iterator patterns for name-checked access.**"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)

Using your experience to evaluate the characteristics, advantages, and weaknesses of each operating platform (Linux, Mac, and Windows) as well as mobile devices, consider the requirements outlined below and articulate your findings for each. As you complete the table, keep in mind your client’s requirements and look at the situation holistically, as it all has to work together.

In each cell, remove the bracketed prompt and write your own paragraph response covering the indicated information.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS offers a UNIX‐based foundation with excellent developer tools and a familiar graphical interface, making setup and local testing straightforward. However, as a production server platform it comes with high hardware costs, limited cloud‐hosting options, and fewer enterprise management utilities compared to Linux or Windows Server. macOS servers typically see less community support for large‐scale deployments, so while ideal for rapid prototyping, they are less common in high-traffic, multi-user environments.  [Part 2]  **macOS** can host web services via the built-in Apache/Nginx stack or with commercial macOS Server (a $19.99 App Store purchase) on Apple hardware. True macOS VMs in the cloud are rare, so most deployments rely on on-premise Mac minis or Mac Pros—hardware costs are high and enterprise management tools are limited. This makes macOS a fine choice for prototyping but uncommon for large-scale, multi-user production environments. | Linux dominates the web-server landscape thanks to its open-source model, lightweight footprint, and rock-solid stability. Distributions like Ubuntu Server or CentOS provide mature package ecosystems, strong security defaults, and seamless integration with containers (Docker) and orchestration tools (Kubernetes). The steepest challenge is the initial learning curve for command-line administration, but once configured, Linux excels at handling concurrent connections, automatic updates, and resource-efficient operation without licensing fees.  [Part 2]  **Linux** is the dominant server platform for web applications—it runs on virtually any commodity x86\_64 hardware or cloud VM, with no licensing fees. Distributions such as Ubuntu LTS or CentOS offer mature package ecosystems, built-in web stacks (Apache/Nginx, OpenJDK), and strong container/orchestration support (Docker, Kubernetes). Its only drawbacks are the initial command-line learning curve and the need for manual patch management in some distros. | Windows Server delivers tight integration with the .NET framework, Active Directory, and Microsoft SQL Server, making it a top choice for shops invested in the Microsoft ecosystem. It features a familiar GUI for administrators who prefer graphical tools over the command line, and built-in IIS for easy website hosting. Drawbacks include higher licensing costs, a larger system footprint, and historically more frequent patch cycles. Windows Servers remain popular for line-of-business applications but are less common for open-source web stacks.  [Part 2]  **Windows Server** provides IIS, .NET integration, and Active Directory out of the box, making it attractive for shops already invested in Microsoft ecosystems. It supports both on-premise and Azure VM deployments, but each instance requires a Windows Server license plus Client Access Licenses (CALs), driving up costs. While Windows’ GUI eases administration, patch cycles can require frequent reboots. | Smartphones and tablets are not optimized as headless web servers; their CPUs, memory, and power‐management schemes prioritize battery life and user-focused interfaces over sustained, multi-client workloads. While lightweight HTTP servers can run on Android or iOS for testing or peer-to-peer demos, they lack the reliability, network stability, and security controls required for production. In a pinch you could prototype on a mobile device, but they are unsuitable for hosting a robust, multi-user web application.  [Part 2]  **Mobile devices** (Android/iOS) are not designed as headless servers. Although lightweight HTTP servers exist for demos or peer-to-peer apps, their CPU throttling, limited RAM, and battery-saving power profiles make them unsuitable for hosting a robust, multi-user web service. For production you must choose a true server OS rather than a mobile device. |
| **Client Side** | Supporting macOS clients requires ensuring our web UI renders correctly in Safari (WebKit) and Chromium-based browsers. This demands time for cross-browser testing and occasional workarounds for WebKit quirks. Expertise in responsive CSS and JavaScript debugging is essential, but licensing costs are minimal since testing can be done on existing developer machines.  [Part 2]  On **macOS** desktop, you must test in both Safari (WebKit) and any Chromium-based browser. That requires extra cross-browser QA and occasional WebKit-specific workarounds (CSS prefixes, feature detection). Developer tools (Safari Web Inspector, Chrome DevTools) are free, and the macOS environment is stable, so licensing costs are limited to existing developer machines. | On Linux desktops, users will primarily use Firefox or Chrome—both follow modern web standards closely, so fewer rendering issues arise. The main investment is in setting up test VMs or containers to validate behavior across distributions. A moderate level of Linux command-line familiarity is needed for browser automation (e.g., Selenium) but no special licensing is required.  [Part 2]  **Linux** desktop support focuses on Firefox and Chrome, which both adhere closely to web standards and require minimal polyfills. You’ll need Linux VMs or containers for automated testing (Selenium, headless Chrome), and devs must be comfortable with the command line for scripting. All tools—browsers, Node.js, VS Code—are open source with zero licensing fees. | Windows clients use Edge (Chromium) or, occasionally, legacy IE11, so our front-end must be tested in both. Time must be budgeted for patch-cycle updates and Polyfills to support older engines. Developers should be comfortable with PowerShell or WSL for automation. Microsoft’s free Edge DevTools and remote debugging tools keep costs low.  [Part 2]  Supporting **Windows** clients means validating in Edge (Chromium) and, if needed, in legacy IE11 for enterprise customers. This adds time for polyfills and conditional code paths. Development teams may use WSL or PowerShell scripts for build automation. All required tools (Edge DevTools, VS Code, Node.js) are freely available, though Windows Server CALs apply only to server usage. | Mobile support adds the highest complexity: we must ensure responsive layouts and touch interactions across iOS Safari and Android Chrome/WebView. This requires additional QA time for emulators and real devices, plus expertise in CSS media queries and fast event handling. While provisioning devices has a hardware cost, much testing can be offloaded to cloud-based device farms.  [Part 2]  **Mobile** support is the most complex: you must design a responsive, touch-friendly UI that works in iOS Safari and Android Chrome/WebView. That means maintaining emulators and real devices for QA, writing media queries, and optimizing JavaScript event handling. While on-premise device labs have hardware costs, cloud-based device farms (BrowserStack, AWS Device Farm) can offset that expense. |
| **Development Tools** | For backend Java work, we use Eclipse or IntelliJ IDEA on macOS, along with Maven or Gradle from Homebrew. Front-end teams leverage VS Code plus Node.js/npm for building and bundling assets. Safari’s Web Inspector and Chrome DevTools handle client-side debugging.  [Part 2]  On **macOS**, Java back-end work uses Eclipse or IntelliJ IDEA (both free Community editions) with Maven or Gradle installed via Homebrew. Front-end teams use VS Code plus Node.js/npm for bundling. Native tools like Safari Web Inspector and Chrome DevTools handle client-side debugging at no cost. | On Linux servers and dev boxes, Java developers use IntelliJ or Eclipse with Maven/Gradle. Front-end code is managed with Node.js/npm and VS Code or Vim. Shell scripting, Git, Docker, and headless browser tools (e.g. Puppeteer) automate builds and tests.  [Part 2]  On **Linux**, developers use IntelliJ or Eclipse with Maven/Gradle for Java, and VS Code (or Vim/Emacs) with Node.js/npm for front-end. Shell scripting, Git, Docker, and headless browser tools (Puppeteer) automate CI/CD pipelines with zero licensing fees. | Windows developers can choose Eclipse or IntelliJ for Java, with Maven/Gradle in a PowerShell or WSL environment. VS Code covers front-end work under Node.js/npm. IIS Express may be used for quick local testing, and PowerShell scripts handle deployment tasks.  [Part 2]  On **Windows**, Java IDEs run under Eclipse or IntelliJ (Community editions) within PowerShell or WSL. Front-end builds use VS Code and Node.js/npm. IIS Express can be used for local web testing, and PowerShell scripts manage deployments. All these tools are freely available for developers. | Though the app is web-based, when debugging on Android or iOS we use Android Studio’s emulator and Xcode’s iOS Simulator. Chrome DevTools remote-debugs WebViews, and Safari Web Inspector attaches to iOS Safari. These tools ensure parity between desktop and mobile experiences.  [Part 2]  Although the application itself is web-based, debugging on **mobile** requires Android Studio (free) for Android emulation and Xcode (free) for iOS simulation. Remote-debugging capabilities in Chrome DevTools and Safari Web Inspector connect to WebViews and mobile browsers to ensure feature parity across platforms. |

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

1) Operating Platform

Recommendation: Deploy the server side of Draw It or Lose It on Ubuntu Server LTS (x86\_64) running in a cloud environment (AWS, Azure, or GCP) or on equivalent on-prem virtual machines.

Why: Ubuntu LTS provides long-term security updates, a vast package ecosystem, excellent Java/OpenJDK support, and first-class container/orchestration tooling (Docker, Kubernetes). It’s license-free, widely supported by cloud providers, and the prevailing choice for web workloads—minimizing cost and maximizing portability.

Alternatives considered: Windows Server offers strong .NET/AD/IIS integration but higher licensing cost and heavier patch cycles. macOS is excellent for development but isn’t commonly used for scalable, production web servers or cloud VMs.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) Operating Systems Architectures

The recommended platform’s key architectural characteristics and how we’ll leverage them:

64-bit, SMP Kernel with Preemptive Multitasking: Fully utilizes multi-core CPUs during peak rounds. Threads from the Java process (game API and WebSocket servers) are scheduled across cores for low-latency rendering/updates.

Process & Namespace Isolation (cgroups + namespaces): We’ll containerize services. CPU, memory, and I/O limits protect the system from runaway processes and give dependable performance even under load spikes.

Virtual Memory & Paging: Demand paging and page cache increase effective memory. Transparent Huge Pages can improve throughput for large Java heaps (we’ll benchmark and enable only if it reduces GC pause times).

Modern Networking Stack: High-performance TCP/IP, epoll, and kernel offloads optimize concurrent connections for REST and WebSocket traffic.

Filesystem Layers: We’ll use ext4 or XFS on block storage for application code/logs, and object storage (e.g., S3/Blob) for static drawings (images).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3) Storage Management

We separate structured data, static media, and caching:

Relational Data — PostgreSQL:

Stores games, teams, players, scores, and audit logs.

ACID transactions ensure consistency for name-uniqueness checks and round timing.

Indexes on name fields prevent duplicates and keep lookups fast.

WAL (write-ahead log), point-in-time recovery, and streaming replication provide durability and failover.

Retention & Backups: daily full backups plus 15-minute WAL shipping; keep 30 days; test restores monthly.

Static Assets (Drawings Library) — Object Storage + CDN:

Store clues as immutable objects (bucket/key).

Serve globally via CDN for low latency and bandwidth offload.

Versioning and lifecycle policies allow differential retention (e.g., keep the last N versions, expire old).

Filesystem Components & “Differential” Data:

Block devices → filesystem (ext4/XFS) → directories/inodes → files for app/runtime/logs.

For incremental change tracking: ZFS/Btrfs snapshots (where available) or cloud incremental snapshots of volumes; in the database, WAL provides differential replay to a restore point.

Logs rotate with logrotate (size/time policies) and ship to centralized storage (e.g., CloudWatch/Stackdriver/ELK).

Caching — Redis (In-Memory):

Speeds up frequently read metadata (game state snippets, scoreboard).

Can hold HTTP/WebSocket session state if sticky sessions are not used.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4) Memory Management

OS Level (Ubuntu):

Demand-paged virtual memory and page cache reduce disk I/O.

Tunables: vm.swappiness (keep low on memory-rich nodes), vm.max\_map\_count (for large JVMs), and THP (enable/disable based on GC profiling).

Container cgroups enforce hard memory caps per service; OOM killer events are surfaced to alerts.

JVM Level (Java/OpenJDK):

Set heap explicitly (-Xms, -Xmx) to avoid dynamic resizing.

Use G1GC (or ZGC for ultra-low latency) with pause-time targets appropriate for 1-second drawing cadence.

Off-heap (direct buffers) or Redis for hot metadata to reduce GC churn.

Monitor with JFR/JMX; alert on GC pause percentiles and heap occupancy.

Application Patterns:

The server remains stateless for REST, keeping minimal in-memory state per request.

Real-time features (WebSockets) maintain lightweight session state; heavier state is externalized to Redis or DB.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5) Distributed Systems and Networks

To support thousands of concurrent players across platforms:

Service Topology:

API Gateway/LB (NGINX/HAProxy) → Stateless REST Services (game lifecycle, auth) → WebSocket Gateway (real-time push) → PostgreSQL / Redis / Object Storage.

Containerized services orchestrated by Kubernetes (rolling updates, health checks, autoscaling, pod anti-affinity for resilience).

Real-Time Messaging:

WebSockets for push of countdowns, clues revealed at 30s, and turn changes.

Optional message broker (RabbitMQ/Kafka) to decouple producers (game logic) from consumers (broadcast services, analytics).

State & Sessions:

REST stays stateless; sessions (if required) stored in Redis (or JWT on clients). No dependence on sticky sessions at the LB.

Resilience Patterns:

Circuit breakers, retries with exponential backoff, timeouts, and idempotency keys to handle partial failures.

Multi-AZ database replicas, read replicas for scale, automatic failover for HA.

CDN reduces latency globally; anycast DNS enables region failover.

Networking:

Private subnets for app/data tiers; public subnets only for the gateway.

Security groups / network ACLs strictly control east-west and north-south traffic.

L7 WAF and rate limiting to mitigate abuse/DDoS.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6) Security

Security spans in transit, at rest, identity, secrets, and operations:

Transport Security:

Enforce TLS 1.3 everywhere (HSTS on public endpoints).

Mutual TLS for internal service-to-service traffic where feasible.

Identity & Access:

OAuth 2.0 / OIDC for user auth; JWT access tokens with short TTLs and refresh tokens.

RBAC for players/admins; server-side authorization checks on every call.

Least privilege IAM for services; per-service identities and scoped credentials.

Secrets & Keys:

Manage secrets in KMS/Secrets Manager/Vault; rotate regularly.

No secrets in code or images; mount at runtime.

Data Protection at Rest:

AES-256 encryption on PostgreSQL volumes, object storage buckets, and backups/snapshots.

Row-level encryption for sensitive PII if required by policy; strict retention limits (GDPR/CCPA aware).

Application Security:

Input validation, parameterized queries, CSRF protection for form posts, CORS set to trusted origins only.

WAF rules for common attacks (SQLi/XSS), rate limits and captcha for abuse endpoints.

Monitoring & Compliance:

Centralized logs (API/WebSocket/db/audit), immutable storage, access auditing.

Automated vulnerability scans, dependency checks (OWASP Dependency-Check), and container image signing.

Patch pipeline for OS and runtime; CIS benchmarks where available.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Implementation Roadmap (condensed)

Phase 1: Stand up Ubuntu LTS, container registry, CI/CD; deploy stateless REST and WebSocket gateways; PostgreSQL primary + read replica; Redis cache; S3 + CDN for drawings.

Phase 2: Autoscaling, global CDN, message broker for analytics/broadcast; multi-AZ failover testing; observability (metrics, tracing, logs).

Phase 3: Security hardening: WAF, mTLS internal, secret rotation, compliance reporting; chaos drills and restore tests.

Risks & Mitigations (high-value)

Peak load spikes → HPA autoscaling + pre-warmed capacity; backpressure and graceful degradation.

Name uniqueness race conditions → DB unique indexes on names + transaction retries.

WebSocket fan-out → Sharded gateway pods; broker-backed pub/sub for large rooms.

Data loss → PITR backups, cross-region snapshots, periodic restore drills.

Credential exposure → KMS-managed secrets, short-lived tokens, zero secrets in repos/images.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Summary:

Ubuntu Server LTS with a containerized Java stack, PostgreSQL for core data, object storage + CDN for drawings, Redis for caching, and Kubernetes for orchestration best satisfies The Gaming Room’s performance, reliability, and cost goals. The architecture supports real-time gameplay across browsers and mobile, scales horizontally during peak rounds, and enforces strong, modern security controls to protect users and the platform end-to-end.