

# **Draw It or Lose It Web App**

Version 1.0

## Table of Contents

[**Draw It or Lose It Web App** 1](#_Toc203333874)

[Table of Contents 2](#_Toc203333875)

[Document Revision History 2](#_Toc203333876)

[Executive Summary 3](#_Toc203333877)

[Requirements 3](#_Toc203333878)

[Design Constraints 3](#_Toc203333879)

[System Architecture View 4](#_Toc203333880)

[Domain Model 4](#_Toc203333881)

[Evaluation 6](#_Toc203333882)

[Recommendations 7](#_Toc203333883)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/14/25 | Jorge F. M. Jacob | Initial draft of design document completed with Executive Summary, Requirements, Design Constraints, Domain Model, UML diagram, and Java application patterns and purpose comments. |

## [Executive Summary](#_sbfa50wo7nsh)

At the Technology Solutions, we are excited to team up with The Gaming Room to take Draw It or Lose It beyond Android and into everyone’s browser. The current mobile-only application limits the client’s ability to reach a broader audience; by delivering a browser-based implementation (with optional native wrappers for iOS and desktop), CTS will enable cross-platform play and unlock new markets.

This Software Design Document presents a high-level architecture for the new application. It introduces an Entity-based domain model (with Game, Team, and Player classes inheriting common attributes), describes the use of the Singleton pattern to manage a single shared game service instance, and details the Iterator pattern for efficient navigation through game, team, and player collections. Together, these design choices ensure consistency, scalability, and maintainability in a distributed, web-based environment.

## Requirements

* Business Requirements

1. Expand Draw It or Lose It from Android-only to a responsive web app (with optional iOS/desktop wrappers)
2. Preserve feature parity with the existing Android game (drawing canvas, timers, scoring, leaderboards, multiplayer).
3. Integrate with the current Java back end (GameService) to reuse game data and analytics.

* Technical Requirements

1. Authentication: Email plus Google/Facebook login.
2. Real-Time Gameplay: Stroke transmission and guess updates under 200 ms latency.
3. APIs: RESTful endpoints for game operations (addGame, getGame, getGameCount).
4. Browser Support: Latest Chrome, Firefox, Edge on desktop and mobile.
5. Security: HTTPS with TLS 1.2+.
6. Scalability & Performance: Support ≥ 1000 concurrent users with API responses < 200 ms.
7. Accessibility: WCAG 2.1 AA compliance (keyboard, ARIA, contrast).
8. Internationalization: English and Spanish with hooks for more languages.
9. Progressive Enhancement: Offline fallback and low-bandwidth support for the drawing canvas.

## [Design Constraints](#_2et92p0)

1. Platform Compatibility: Must run on modern browsers (Chrome, Firefox, Safari, Edge) and support optional iOS/desktop wrappers.
2. Back-End Integration: All game actions go through the existing Java GameService via REST APIs to preserve data and analytics.
3. Performance & Scalability: Maintain ≤ 200 ms latency for real-time drawing/guesses under normal conditions and support ≥ 1000 concurrent users.
4. Security: Enforce HTTPS (TLS 1.2+), with email and Google/Facebook OAuth login.
5. Accessibility & Localization: Meet WCAG 2.1 AA (Keyboard, ARIA, contrast) and support English/Spanish (Extendable).
6. Offline & Progressive Enhancement: Ensure core drawing features work during intermittent connectivity via service-worker caching.

## [System Architecture View](#_ilbxbyevv6b6)

* Front End:

Players open the game in their web browser (Chrome, Safari, etc.) or in a little wrapper app on their phone or desktop. That layer shows the drawing canvas, the list of games, and lets you type guesses and see scores in real time.

* Back End:

When you are drawing or joining a game, your browser sends a message over secure HTTPS to our Java server. That server runs a single, shared component called the GameService, that handles every request (addGame, getGame, getGameCount). Because there’s only one instance of this service (the “Singleton”), all players stay in perfect sync.

## [Domain Model](#_8h2ehzxfam4o)

* Inheritance Hierarchy

At the center of the model is the Entity base class, which defines the common attributes. All three domain classes (Game, Team, and Player) inherit from Entity. This avoids duplication of id and name fields and ensures a consistent interface for obtaining an object’s unique identifier and display name.

* Aggregation Relationships

1. GameService -> Game

The GameService singleton manages a collection of Game instances:

– games: List<Game>

+ addGame(name: String): Game

+ getGame(...): Game

+ getGameCount(): int

The dashed multiplicity 0…\* beside the association arrow indicates that a single GameService can hold zero or more games.

1. Game -> Team

Each Game can contain multiple teams:

– teams: List<Team>

+ addTeam(name: String): Team

The 0…\* notation shows that a game may have many teams (or none yet).

1. Team -> Player

Teams similarly aggregate zero or more players:

– players: List<Player>

+ addPlayer(name: String): Player

These aggregation arrows (“has-a” relationships) reflect the real-world hierarchy: games are composed of teams, and teams of players.

* Behavioral Classes and Patterns

1. GameService

Marked with – instance: GameService and a + getInstance(): GameService method, this class implements the Singleton pattern, ensuring there is exactly one service in memory managing all games.

1. ProgramDriver

With only + main(args: String[]): void, this class is the application entry point. Its <<uses>> arrow toward GameService shows that it retrieves the singleton instance to initialize and display game data.

1. SingletonTester

Contains + testSingleton(): void and also <<uses>> GameService to prove that any part of the application accessing getInstance() is working with the same underlying object. Inside its test method, it iterates over the games list, illustrating the Iterator pattern in action.

* Object-Oriented Principles in Play

1. Encapsulation: All fields are private (–), accessed only through public getters or methods.
2. Inheritance & Polymorphism: The Entity→(Game, Team, Player) hierarchy lets client code treat all entities uniformly (e.g., storing heterogeneous objects in a single collection).
3. Abstraction: Consumers of GameService and the domain objects need only know the public interfaces, details of data storage and ID generation are hidden inside the classes.
4. Design Patterns:

* Singleton (creational) in GameService
* Iterator: (behavioral) in the loops inside both GameService and SingletonTester

**"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** | • Characteristics: UNIX-based, POSIX compliant • Advantages: native Apache/Nginx, good Java support • Weaknesses: high hardware cost, limited cloud images | • Characteristics: open-source, lightweight • Advantages: free, rock-solid stability, massive hosting support • Weaknesses: requires Linux admin skills | • Characteristics: Windows Server OS • Advantages: tight MS-stack integration (IIS, AD) • Weaknesses: licensing fees, higher resource usage for Java | • Characteristics: Android/iOS OS • Advantages: none for production hosting • Weaknesses: insufficient CPU/RAM, no server runtimes |
| **Client Side** | • Cost: Mac hardware + Safari licenses • Time: moderate browser‐compat testing (Safari) • Expertise: macOS/WebKit quirks | • Cost: minimal (use existing machines or VMs) • Time: test on Firefox/Chrome only • Expertise: Linux CLI familiarity | • Cost: Windows licenses + test lab • Time: test Edge/IE fallbacks • Expertise: Windows-specific CSS/JS fixes | • Cost: device lab or emulators • Time: extensive responsive & touch testing • Expertise: mobile UI frameworks & network variability |
| **Development Tools** | Languages/Frameworks: Java 11+, JavaScript/TypeScript, HTML5/CSS3 • IDEs/Tools: Visual Studio Code, IntelliJ IDEA, Xcode (iOS), Homebrew | •Languages/Frameworks: OpenJDK 11+, Node.js, HTML5/CSS3 • IDEs/Tools: Visual Studio Code, IntelliJ IDEA, Maven/Gradle, Docker | •Languages/Frameworks: Zulu OpenJDK 11+, Node.js, HTML5/CSS3 • IDEs/Tools: Visual Studio Code, IntelliJ IDEA, Maven/Gradle, Git for Windows | Languages/Frameworks: Java/Kotlin (Android), Swift/Objective-C (iOS), HTML5/CSS3 in embedded WebView • IDEs/Tools: Android Studio, Xcode, Chrome DevTools, Cordova/Capacitor |

## Recommendations

1. **Operating Platform**: We recommend deploying Draw It or Lose It on a Linux server environment (for example, Ubuntu LTS). Linux offers rock-solid stability, native support for Java runtimes, and zero licensing costs. It integrates seamlessly with container platforms and cloud providers, making it easy to scale out as user demand grows, exactly what The Gaming Room needs to expand beyond a single Android app.
2. **Operating Systems Architectures**: Containerize the Java REST API and any auxiliary services with Docker and orchestrate them using Kubernetes (or a managed alternative like AWS EKS/GKE). This approach ensures consistent runtime environments from development through production, simplifies rolling updates with zero downtime, and allows horizontal scaling by adding more container replicas behind a load balancer.
3. **Storage Management**: For the prototype, keep game, team, and player data in the in-memory collections of GameService. As you move to production, migrate that data to a managed SQL database (e.g., Amazon RDS for MySQL or PostgreSQL). A relational store offers durable persistence, automated backups, and read replicas to handle high query volumes without altering the existing REST API.
4. **Memory Management**: Run the Java service on a JVM configured for server workloads (G1 or ZGC garbage collector). Allocate an initial heap of 1–2 GB, then monitor and tune via JMX or Prometheus. In containers, enable container-aware memory limits so the JVM respects Docker resource quotas and avoids out-of-memory failures under load.
5. **Distributed Systems and Networks**: Place all API containers behind a TLS-terminating load balancer (Nginx, AWS ALB) to distribute traffic evenly. Use service discovery (Kubernetes DNS or Consul) so microservices can find one another reliably. Incorporate retry and circuit-breaker patterns (via Resilience4j or Spring Cloud) to handle brief network outages gracefully and maintain responsiveness in a distributed environment.
6. **Security**: Protect all client–server communication with HTTPS/TLS 1.2+. Use OAuth2 (Google/Facebook) and issue JWT tokens for session management. Encrypt sensitive data at rest in the database and configure a Web Application Firewall (WAF) to guard against XSS, CSRF, and injection attacks. Isolate services in a private network segment and expose only necessary ports (e.g., 443) to minimize the attack surface.