

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template**](#_l6ti7uoag22u)1

[**Table of Contents**](#_30j0zll)2

[**Document Revision History**](#_grjogdjh5fi8)2

[**Executive Summary**](#_sbfa50wo7nsh)3

[**Design Constraints**](#_2et92p0)3

[**System Architecture View**](#_ilbxbyevv6b6)3

[**Domain Model**](#_8h2ehzxfam4o)3

[**Evaluation**](#_2o15spng8stw)3

[**Recommendations**](#_m8aleynsvzvc)5

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 9/16/2020 | James Davis | Initial Draft |

**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 wants to develop a web based game that serves multiple platforms on their current game, Draw It or Lose it. Currently, the game is only available on the Android platform. The application must render images from a large library of stock drawings as clues as the users attempt to guess the puzzle. Answers can be in the form of a phrase, title, or thing. Drawings are rendered at a steady rate until fully complete at the 30 second mark.

## [Design Constraints](#_2et92p0)

Currently, the app runs only on Android and locally. Logic and data will need to move server-side to ensure compatibility across user-facing operating platforms. We will have to design an API from the server-side that can be used by each user-facing OP involved.

Storage and memory sufficient to allow images to be stored and drawn must be included. Going with cloud architecture here will be ideal to allow for further expansion. Since the customer has not provided data on whether or not the drawings are vector-based logic rendered ad-hoc or pre-stored images rendered bit by bit, we will need to ask them to ensure we select the appropriate storage and memory capacity for the technique used.

A separate authentication method will need to be used server side that can work with Google, Apple, Facebook, and any other login APIs that the client specifies. Depending on to-be-determined client requirements, we can also create a native user-facing logon protocol with two-factor authentication. Since the app is primarily geared toward mobile users (if only by basic demographics), SMS is an ideal compromise between ease-of-use and security.

## [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 Singleton Tester public testSingleton function is associated with and will be used by the Program Driver main function to test the software.

The GameService class will have as private variables the list type of games, long type of nextGameId, long type of nextPlayerId, nextTeamId, and the private object GameService. The GameService function is private, and getInstance:GameService, addGame, getGame id, getGame name, getGameCount, getNextPlayerId, and getNextTeamid are private.

GameService has a one-to-many relationship with the Game class. The game class has a private teams list type object, and public Game, addTeam, and toString functions.

The Game class has a one-to-many relationship with the Team class, which has a private players list type object, and public Team, addPlayer, and toString functions.

The Team class has a one-to-many relationship with the Player class, which has public methods Player and toString.

Game, Team, and Player classes all inherit from the Entity class. The Entity class has private variables id and name, private function Entity, public function Entity with id name, public function getId, public function getName, and public function toString.

## [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** | Offers no server hardware/software solution. | Cost-effective and adaptable. Encompasses many solutions. | Proprietary, expensive. Possible ease-of-use advantages if the company already has Windows power users. | Not generally used for this function. |
| **Client Side** | Ensure compatibility with the Safari web browser and Mac versions of Chrome and Firefox beyond general browser compatibility. | Minimal to nonexistent. Chrome, Firefox, etc. support website design in a seamless fashion with Windows. | Basic compatibility with modern browsers in the Windows environment. | Significant UI redesign for smaller screens, as well as ensuring website runs code compatibly with iOS and Android built-in and major third party browsers. |
| **Development Tools** | Mac machines to test browser compatibility. | Linux virtual machines to test browser compatibility or write code. | Windows machines to text browser compatibility. | iOS devices, Android devices to test compatibility with newer and older versions of the operating systems that are still in wide use. |

## 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**: We recommend the use of a cloud-based service like Microsoft Azure with granular scaling and the ability to separate processing, security, and data storage for flexibility in pricing and design going forward. This will allow the developer to take advantage of “bundling” or “unbundling” these three separate components as advantageous financially.
2. **Operating Systems Architectures**: A Linux based “server” (really, running on cloud architecture and abstracted across Azure’s hardware) is our best solution. For software this simple, we can design an open-source, low-cost, secure, system-resource sparing environment that lacks unnecessary and expensive features. If upscaling is desired later, the code can be written and compiled in a way that is friendly to other environments by choosing an operating-system agnostic language like Java or Rust. By keeping the code and logic in an environment agnostic language, we make it portable and thus more future-proof.
3. **Storage Management**: Storage management will be scalable with Azure. As historical games data and/or image libraries grow over time, so too can the data storage available. With scalable storage the developer will only pay for what they are using at the time.
4. **Memory Management**: Memory management with Azure will not be a problem. The code is very lightweight per game session and Azure can scale to what is needed as quickly as user demand rises. On the client side the software will be coded such that the image being rendered is by far the largest demand on memory, and only the image required *for that game session* will be in volatile memory – and upon that session ending, the memory storing the image will be released. The images will not be server-side nor present in server-side memory, meaning that the values stored are text or numbers only, not expensive and large images. Memory requirements will be very minimal.
5. **Distributed Systems and Networks**: All distribution will be handled over the Internet exactly as websites and their associated data are handled, via the HTTPS protocol for movement of textual and image data types. This is a browser-based game (each operating system’s client is essentially a specific use-case browser) that stores and pulls data from the server for multi-player as allowed by the code. The game sends user information and hash data for images, not the images themselves. To illustrate, once the game is downloaded, a user could play it on a dial-up modem or 2G cellular connection with the same user experience as someone on a 100MBsec bimodal fiber optic connection or 5g cellular connection. The data that needs to be sent for games to be played is most reasonably measured in kilobits per second, not megabits per second.
6. **Security**: Security will be managed by using appropriate encapsulation coding procedures, length limits on user input fields, and working with Azure to establish two-factor authentication for initial setup of a new device. After that, we will respect the user’s settings on that device (i.e., if they have a PIN lock on their phone, the app will obviously be shielded behind that, but we will not force the user to set a PIN or have one for the game). Server access by developers/game administrators will force at least two-factor (ideally multifactor) access. We recommend using authentication tokens from cellular devices, SMS, and a username/password combination. This will help ensure that users do not share logins and prevent external hacking attempts.   
     
   Insofar as protecting user data between client and server, we will design both the server and client software to establish an encrypted “tunnel” for all subsequent communications – both user login and game data. This will help prevent cheating and misappropriation/man-in-the middling of game data during a session. Breaking encryption during the short, live game sessions will be extraordinarily difficult compared to the potential reward.