

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.2

## 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 | 11/9/23 | Justin Phillips | Created the document |
| 1.1 | 11/25/23 | Justin Phillips | Fleshed out evaluation |
| 1.2 | 12/10/23 | Justin Phillips | Recommended an operating system for server use, produced information on storage management, memory management, distributed systems, and security functionality. |

**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 seeks to develop a multiplatform web app for their game *Draw It or Lose It*. This game will be hosted on a server(s). It will allow multiple games (infinitely scalable) with multiple teams and multiple players on each team. Each instance of a game, team, and player should be unique by taking advantage of singleton and iterator patterns. The game must be accessible through various platforms.

Draw It or Lose It already exists as an Android app, written in Java.

## Requirements

Draw It or Lose It must be hosted on the Google Play Store on Android devices and the Apple App Store on iOS devices, as well as a web app.

The API must be tailored then to work between these platforms. Teams must be able to join a game hosted from any of these platforms.

The names of the game lobby and the teams must be unique.

If a name fails to be unique, the team leader should be notified, and a new team name may be entered to retry.

Users should be able to create an account, log into the account through authentication, and join a game and team.

Web deployment is a critical feature, so Java will be used to engineer the app taking advantage of tools such as the REST API.

Since the app exists on Java, the hooks offered by the API need to be made portable for various OS environments.

## [Design Constraints](#_2et92p0)

The game must take keyboard input from a physical or virtual keyboard, i.e. from a phone or tablet. The game must be written in a platform agnostic manner to make the user experience as seamless as possible between iOS, Android, or web. The game must be kept lightweight, responsive, and have optimized packets.

The game must have a permission system to determine team leadership permissions, end user permissions, and operator (i.e. administrator) permissions through logging in.

The game must be multiplatform. The game server application should be written in Java and using REST API.

## [Domain Model](#_8h2ehzxfam4o)

**"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.**ProgramDriver is the class of the program that holds and runs main(), the body of the program that actually executes the code. This class uses SingletonTester, a class that checks whether the GameService is instantiated.

The Entity class is a parent class that holds an id and name that are publically available. The name and id can be retrieved. The Entity can be initiated with an id and name as a long and String. The Entity can output a descriptive string with the name and id.

Game, Team, and Player all extend the Entity class.

Team can have zero to many players and players can be added. Team holds a list of players.

Game can have zero to many teams. Teams can be added to the Game. Game holds a list of teams.

Game also uses GameService, which has a singleton pattern to see if a specific instance exists of a game or if that instance does not exist. If a GameService instance exists, it will return that. If not, it will create a new one. There is a nextGameId, nextPlayerId, and nextTeamId to keep track of the games, teams, and players managed. A game can be added. A game can be retrieved. The number of games can be counted which returns an int. Zero to many Games can exist in a GameService, but only one GameService can run.

## 

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS server has been discontinued. Apple advises using third party services to webhost. | Linux is free and open sourced and is a transparent OS solution for web-based hosting. The client can host their own site without needing to worry about expensive licensing fees. Scalable, large support community on forums, compatible with pretty much any desktop or server hardware. Can self host (high up front cost + needs maintenance) or use a third party service (costs more over time + maintenance outsourced). Can be good at interfacing with other OSes due to open source nature. Can use Red Hat, Kubernetes, and other distributions and tools for optimal setup. | Windows offers dedicated servers and Windows is very secure but developer friendly. Licensing is expensive for this. GUI walks the client through setup. Similar to Linux, can self host with same pros and cons or use a third party with same pros and cons. Windows is generally compatible but may be less compatible with other platforms and OSes, unlike Linux. | Mobile devices can host but are severely power limited. They cannot be expanded easily, they tend not to have optimal core count/speed, and custom tools would need to be developed to make this work properly.  A botnet may be able to be deployed, but the alternative of one centralized hosting machine makes much more sense. |
| **Client Side** | Apple has a user friendly developer environment but requires a Mac to develop software for their platform. The Swift programming language must be used. This is expensive and requires a Swift programmer. | Development time and familiarity with Python are the largest expenses and requirements in developing for Linux. There are many libraries available to make development easier, and many dev tools available. This is inexpensive and requires somewhat knowledgeable entry-level programmers. | Visual Studio developing an application that utilizes .NET framework would be optimal. The developer must be familiar with developing applications for Windows. There are many dev tools and libraries available and C++ redistributable, Visual Studio, etc can all be taken advantage of. This is inexpensive and requires somewhat knowledgeable entry-level software engineers. | Developers should have app development experience. User interaction and how things are displayed need to be tailored to be adaptive between a variety of screen sizes and the landscape and portrait layouts. Typically would use Java for Android and Swift for iOS.  This would take a certain level of specialization and familiarity with mobile platforms, and would require both knowledge of Swift and Android, which may mean hiring separate teams for either platform or a specialist. This has potential to be expensive. As well, the Gaming Room must get their app approved on the Apple app store. |
| **Development Tools** | Mac is equipped with iCode, with programming done in the Swift language.  Otherwise, Eclipse can be installed for macOS, and would be a great option for developing the server side Java app or the client side Java app for other platforms.  Xcode is a great IDE for Swift development for Mac and iOS, and SwiftUI can be used to quickly build apps. Alternatively, AppKit can be used to build GUI interfaces for Mac and iOS apps. | Linux has native Python support. PyCharm is an ideal IDE. Otherwise, there is a large programmer/developer presence and community on Linux, and tools for just about anything, typically open-source software or packages able to be installed via pip pipeline.  Similarly to Mac, Eclipse IDE can be installed for Java development, making it a great multiplatform IDE for the server and client apps. | Visual Studio can be used to create a native Windows app in C++ or C#. For more complex development, Linux can be installed to Windows, and there are tools for various uses such as IDEs for nearly any language. Developing this app would most likely use easy accessible Windows development tools.  Visual Studio is great for programming webapps and mobile apps, so it is a good IDE to use for developing client-side apps for Windows, Android, and Linux. There could be dedicated teams for each of these platforms. Eclipse is also available for Windows, making it a good multiplatform Java IDE. | Android Studio can be used to write the app in Java on a desktop for Android. Mac can be used to write an iOS app with Swift and iCode. Java cannot be used on iOS – Only Swift and Objective-C are able to be ran. Like MacOS, Xcode on a Mac can be used as an IDE for Swift, and tools like SwiftUI and AppKit can help with UI design. |

## 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**: Linux seems like the best server hosting operating system. An Ubuntu server using Kerbernetes can be employed. Overall the license cost is significantly lower than any other potential hosting platform, and the option to self-host is nice for future expansion, security, and control over the server side of the app. However, outsourcing hosting will still be relatively inexpensive. There is more control over the security end of development with Linux as well.

A Windows server would be a good second contender for hosting. Windows makes hosting a server and managing it fairly intuitive, though there is a steep licensing MSRP of $501 for the barebones Essentials package which is considered ideal for small businesses with 25 users and 50 devices, or the Standard package which does not have as many limitations on an ideal user and device setup but is meant for minimally virtualized environments.

Linux and Windows both allow the use of nearly any language, app, or virtual environment to run a hosting application, making them both strong and versatile candidates for hosting the servers, though Linux allows even more flexibility due to the open source nature and the high level of contribution from the community around it and boasts the benefit of cheaper licensing for server intended distributions such as Ubuntu, Fedora, or Arch Linux.

I would not recommend the use of mobile devices or MacOS for the server. These platforms are limited in their hosting potential, as mobile devices do not have much processing power for hosting, and MacOS Server no longer exists while older versions of the software which still exist have steep licensing fees and require code written in Swift or Objective-C.

I recommend Linux as the most appropriate operating system for The Gaming Room to expand Draw It or Lose It to other computing environments. The networking port 443 for HTTPS can be used for sending packets including images and text from the game to the user and messages from the user to the server and the port NTP at 123 can be used for time scheduling and syncing between the client and server (Sharma, 2022). Hosting any server will likely require database software, such as MySQL, and the use of webserver software, such as the open-source Apache (Sohail, 2023). Linux has commands to use for creating backups as well, such as rsync and cronopete, which can be ran as sudo commands from Linux’s command line, which allows the use of Python to interface with the system (Echo, 2023).

1. **Operating Systems Architectures**: In general, an operating system allows the user to more easily interface with low level functions such as file management, processor management, task scheduling, hardware interaction, error detection and correction, creating the general environment for higher level software to run in, and input and output management. An aspect of Linux that is an advantage is that Linux as the Ubuntu Server distribution only requires 1GB of RAM, a 2.5GB minimum allocation on disk, and a 1Ghz or better processor, which is extremely lightweight (Ubuntu, 2023). Other systems, such as Windows Server, are more demanding. Windows Server requires 1.4Ghz of clock speed, NX and DEP technologies, error correcting code RAM, 2GB pf storage (but a bare minimum of 512 MB), and support for various technologies such as NX, DEP, and Second Level Address Translation (Jenks, 2021).

Linux is a stable and secure environment to host the servers. It is scalable and can take advantage of Kerbernetes. 64 bit should be used for expandability. The server should handle front end rendering of the graphic, the team management, game management, and user management, and operations involving user authentication and permissions. If the app is browser based, development between operating systems for the client side can mostly depend on things such as HTML and IP. Otherwise, in dedicated apps across platforms, an intermediary language or API may be used to simplify development, or individual per-OS apps can be developed that all hook into the API of the server to get information and send requests. A client cache could take some of the server’s drawing and store it locally in an encrypted file for smoother rendering to protect against things like latency or connectivity issues.

In terms of self-hosting server physical architecture, modern server RAM for a DDR4 server can be up to 3200Mhz fast with error correction code, and Intel Xeon line processors would be ideal for hosting a high number of games at once due to the sheer quantity of cores and the high clock speeds and the error correction support they provide. A lightweight operating system such as a Linux distribution would allow the OS to take up less processing power and the actual game to have more resources available, generally being more efficient than a modern Windows installation, which can be significantly advantageous for budget servers or servers trying to fully take advantage of their hardware. A x64bit OS architecture distribution of Ubuntu Server is the obvious choice for the sake of the larger memory address maximum size and the efficiency of a 64 bit processor overall.

1. **Storage Management**: A RAID system that allows back-ups and performance striping should be employed, allowing photos to be saved and backed up in case of data loss, and for fast asset access. Perhaps an NVMe or some other SSD can be used as a cache drive for potential assets currently in use by a game, but the limited read and write life of an NVMe might mean it’s more cost effective to primarily use HDDs. I would advise the use of a 10000rpm HDD and a service such as MIStore which allows the use of SSD caches for maximum performance. Using the hardware RAID configuration, maybe a two-bay dedicated USB-3.0 or 3.1 RAID dock for future expansion, the hardware load can be fully deferred from the OS environment which utilizes no system resources to create fast backups of the server.

As described in my previous report, [*Memory and Storage Management*](../Documents/Memory%20and%20Storage%20Management.docx), a 30Gb physical disk partition should be plenty of storage space for the application and its data, though a generous disk size would be fairly inexpensive and may provide longevity and flexibility to the system. I would recommend a 15000 RPM primary disk for the sake of balanced longevity and efficiency, with a sufficient cache size and enough room for a page file to be hosted to ensure optimal performance throughout the various game session objects hosted in the JVM. The drive I recommended in that report was a 600Gb drive, which is plenty of space for future expansion with the small application. Even allocating 40Gb of storage for a page file would leave 530Gb free after all the data was stored and reached the reasonably defined maximum capacity from that report. Specifically, a 22.58Gb registry of user log-in information holding the records for 177,418,816 players will be kept at most to begin with.

The server software, photo library, operating system, and peripheral utilities/software will likely take up less than the 30Gb suggested allocation of primary partition storage as well. A 10000 RPM hard drive can be used in the RAID configuration as a backup drive to ensure that data is safe, even in the event of a primary disk failure. A 1Tb capacity will allow several backups to be held of the server’s 30Gb primary partition simultaneously, as well as the 22.58Gb registry file, perhaps stored on its own encrypted partition for the sake of security (Phillips, 2023). Alternating two 30Gb partitions leaves us the space for 16 backups of the pairs of partitions and some space left over, 40Gb, for other software and working space.

1. **Memory Management**: It is assumed a Java virtual machine will host the server application. A garbage collector should be employed using the JVM to manage memory and destroy games and their assets at the end of the sessions. Using a custom JVM for the server may also yield better memory management, such as using a distribution like Adoptium. 128Gb of error correcting DDR4 RAM should provide enough room to host 14,357 games holding a cumulative 7,350,784 players simultaneously even with a 6Gb allocation specifically for handling the authentication and logging in services up to 46,875 simultaneous calls (Phillips, 2023). It should also be observed that using a page file will add additional wiggle room for other software to be used, games that are using abnormal amounts of memory, or edge cases where allocation of memory is taking abnormally long. Since the authentication of a user will be dealing with encrypted json packets being sent from client to server and server to client, and the registry file for usernames and passwords will be a fairly large encrypted database, there may be a slower allocation and de-allocation of memory than other processes performed such as hosting the actual game. It may be possible to cache the password registry or regions of it in the page file as well. In Ubuntu, the page file is referred to as ‘swap space’, and can be configured through sudo commands or the user interface. Maintaining a section of dedicated swap space can prevent the premature killing off of processes such as game sessions and add more flexibility and durability to the game manager, improving the end user’s experience by keeping it seamless and preventing memory related crashes (phoenixNAP, 2023).
2. **Distributed Systems and Networks**: The server handles a majority of the processing for the game. The servers need to have backup power supplies and backup drives for redundancy. The photos should be pooled when a game is instantiated and then synced between the client devices as the game runs, allowing seamless background loading, or use the aforementioned cache system. Either way, the client’s received photos should be encrypted or stored in some manner where cheating by looking at files in advance is incredibly difficult or impossible.

The client should primarily act as a user interface to the game’s server’s API and should be designed to facilitate visual information to the client. The client should use the designated communication ports that the server uses to keep information transmissions secure. The client should also be sent the photo being drawn at the start of the round, though kept secure to prevent cheating. The photo should also have a timer associated with its rendering and the render pattern should be transmitted by using a seed that comes with the photo. This will allow synchronization for rendering between players who have worse or better, more or less consistent, connections. All player interaction activity can be kept up through faster and more direct ports, i.e. temporary chat rooms, because game chat is important to keep up to date. However important calls like login requests need to be sent through secure ports and held at higher priority by the server.

The REST API provides a solid client/server relationship foundation for software engineers building the project. Its stateless data system allows the storing of variables per-session and the use of role-based interaction and permissions. Ubuntu Server also has native support for OpenSSL 3.0, which allows modern, general use cryptography and keeps communication secure. Network acceleration is present natively with SmartNIC support, and there is support for several system architectures and common networking processes and tools.

1. **Security**: The player should have no authority other than to play the game. The team leader should have the ability to manage the team, ie. Change name/set name. The game host should be able to remove teams and players and create and end the game. Finally, there should be an administrative role outside of the scope of the end users who may create a game, create a team, or join a team in order to manage the game and enforce guidelines. The admin should be able to access all information handled by the server, while the end user roles (game host, team leader, player) should only have limited access to the server in order to communicate with the game instance, team instance, or gameplay loop.

User permissions should be sorted by the role of the user. A registry should be kept that is exclusively server side recording the unique username key, the role, and the encrypted password of all users. Users should default to basic users, but when making a game or a team move to a leadership role within that game or team, allowing management of exclusively their game or team. Some users who are part of the Gaming Room’s staff should be administrators of the game, and allowed full permissions to end a game session, ban an image used in the game, or even ban a user if required for a certain duration or indefinitely for those exceptionally problematic. The client permissions for a default user should be solely to browse games, create games, join games, or create teams within the game.

To create an account, the user should send the server an attempt at a unique username. If the server determines the username to be a unique (not matched by another key) name in the user database, it should confirm that to the user and allow the user to create a password that meets certain requirements. If either the username test or the password test fail, the user should be prompted to try again.

To log into a game server, the user must send a packet with a username and an attempt at a password to the server, which should then verify the information with a database using the username as a unique key. Failure in this authentication should be reported to the client to allow the client to try again. Multiple failed attempts should result in a locking-out of the account. The user should be able to send a request to the server to reset the account, which the server should forward to the registered email address of the username key.

For the cybersecurity element of hosting the server, a protective firewall should be employed, and all packets not utilizing a port dedicated to the game should be barred from entering the game’s network, barring maintenance ports which should be heavily restricted. Firewalld is an open source, host-based firewall which can be employed for this, as well as the use of a whitelist for the ports (Amoany, 2022). It can be installed, updated, and maintained through sudo commands on the Linux server host. The packets for user authentication should certainly all be encrypted and use HTTPS as the port. .json files should be used because they already have an encryption for the JVM host to parse. The file for the image sent out to the client also needs to be encrypted to prevent cheating. The user protection and security capabilities of Linux are mostly in the way it keeps different processes separate from each other. Each process has its own address space allocations, preventing issues from occurring such as unauthorized read, write, and overwrite by other applications. The Ubuntu distribution also features policy features such as keeping all ports closed, whitelisting ports, password hashing, livepatches, automatic security updates, encryption for virtual machines, and read-only data sections (Ubuntu, 2023).

# References

Amoany, E. (2022, July 12). *How to configure a firewall on Linux with firewalld*. Retrieved from Red Hat: https://www.redhat.com/sysadmin/firewalld-linux-firewall

Echo. (2023, March 9). *How to Backup and Restore Linux Server Easily in 7 Ways?* Retrieved from Vinchin Blog: https://www.vinchin.com/en/blog/backup-and-restore-linux-server.html

Jenks, A. (2021, December 23). *Hardware requirements for Windows Server*. Retrieved from Microsoft.com: https://learn.microsoft.com/en-us/windows-server/get-started/hardware-requirements

Phillips, J. (2023). [*Memory and Storage Management*](../Documents/Memory%20and%20Storage%20Management.docx)*.*

Sharma, S. (2022, Sep 8). *Common Networking Port Numbers in Linux*. Retrieved from linuxhandbook.com: linuxhandbook.com/common-ports

Sneyers, J. (2016, July 20). *One pixel is worth three thousand words*. Retrieved from Cloudinary: https://cloudinary.com/blog/one\_pixel\_is\_worth\_three\_thousand\_words

Sohail, M. (2023, April 24). *How to Host Web Server: A Beginner's Guide*. Retrieved from LinuxAndUbuntu: https://www.linuxandubuntu.com/home/how-to-setup-a-web-server-and-host-website-on-your-own-linux-computer

Ubuntu. (2023, May). *Basic installation*. Retrieved from Ubuntu Server: https://ubuntu.com/server/docs/installation