CS307 – Software Engineering
Design Document

Educational Programming through Interactive Media

Team 2

Members:

Anant Goel, Don Phan, Jason Rahman, Jon Egeland, Josh Selbo, Levi Linville

Table of Contents

1. Purpose
2. General Priorities
    - 2.1. Performance
    - 2.2. Usability
    - 2.3. Maintainability
    - 2.4. Security
    - 2.5. Hosting, Development and Scalability
    - 2.6. Appearance
    - 2.7. Grading Isolation
3. Design Outline
    - 3.1. Overview and Components
    - 3.2. Architecture
    - 3.3. Application Server
    - 3.4. Client
    - 3.5. Grading Server
4. Design Issues
5. Design Details
    - 5.1. Components
        - 5.1.1. Client
        - 5.1.2. Application Server
        - 5.1.3. Database Server
        - 5.1.4. Grading Server
    - 5.2. Game Logic (Client)
    - 5.3. Game Logic (Grading Server)
    - 5.4. Security
    - 5.5. Session Initialization
    - 5.6. Session Persistence
    - 5.7. Roles
        - 5.7.1. Student
        - 5.7.2. Instructor
        - 5.7.3. Administrator
    - 5.8. Domain Classes
        - 5.8.1. User
        - 5.8.2. Class
        - 5.8.3. Role
        - 5.8.4. Level
        - 5.8.5. Challenge
        - 5.8.6. Badge
    - 5.9. System Interactions
    - 5.10. Creating a new user
    - 5.11. Resetting a user’s password
    - 5.12. Enrolling in a class
    - 5.13. Making a code submission
6. UI Mockups

Purpose

Currently, there are many tools that attempt to teach middle to high school students how to program. However, very few tools aim to teach computer science concepts in addition to programming skills, and few teach these concepts in an engaging, interactive medium like a video game. The purpose of our project is to develop a game which teaches students both programming skills and relevant topics in computer science, such as simple algorithms and programing logic.

The following items the major areas of functionality within the application. These areas capture the key user-facing elements of the application.

- **Account management:** Students, instructors, and administrators must be able to manage their own and other user accounts for which they have the appropriate permissions. 1) Students may manage their own account and enroll in classes. 2) Instructors may create and manage classes. These classes create a learning environment tailored to their students. 3) Administrators have the ability to perform the tasks needed to maintain the site, including managing users and classes.
    - As a student, I want
        - to register for the site as a student
        - to reset my password if I forget it
        - to edit my account information (name, email, etc.)
        - to enroll in classes
        - to start playing the game
            - for a class (programming language already specified)
            - in Free Play mode and choose a programming language
    - As an instructor, I want
        - *to have all capabilities listed above*
        - to register for the site as an instructor
        - to create a public or private class with a specific programming language
        - to approve users' requests to join private classes I have created
        - to manage class settings for classes I have created
        - to remove users from a class I have created
        - to delete a class I have created
        - to view all submissions for all users enrolled in classes I have created (see **Instructor-Student Interaction**)
    - As an administrator, I want
        - *to have all capabilities listed above*
        - to register for the site and be manually elevated to an administrator role
        - to edit the account information of any user
        - to reset the game progress of any user in any class
        - to manage class settings for any class
        - to delete any class

        - to delete any user account
- **Game Interaction:** Users enrolled in a class will interact with the game world as they learn programming and computer science skills. Users will have certain opportunities to customize their character and game experience.
    - As a user, I want
        - to choose my character's name and gender
        - my progress to be saved and restored when I return to the game
        - to interact with the game world by
            - moving around the map
            - speaking to non-playable characters in the game world [time permitting]
            - solving programming puzzles to make progress in the game
            - solving optional challenges to acquire bonus items or badges
            - receiving badges for solving certain puzzles and performing certain tasks
        - to view all badges I have received and all available badges
        - to view all users in a class I am enrolled in and the badges they have received
- **Pedagogy**: The overarching goal for this application is to teach basic programming and computer science concepts. Therefore it is important that the gameplay mechanics are entertaining, engaging, and accessible to all students who may or may not have experience playing games.
    - As a user, I want
        - to play a level
        - to view my progress
        - to read tutorials to help me with the gameplay
        - to have the option to view hints for tough problems
        - detailed feedback about my syntax and coding mistakes
        - to play additional challenges to sharpen my skills
        - to see a demo of my executed code before I submit it as a move [time permitting]
- **Instructor-Student Interaction:** Instructors play a key role in the learning process. Our aim is to provide transparency to instructors in the class by providing the functionality needed to understand the progress of their students.
    - As an instructor, I want
        - to view my students' game progress
        - to view my students' badges
        - to view my students' puzzle and challenge submissions

General Priorities

The general priorities of the application are focused on ensuring the success of the application by prioritizing the user experience, and providing quality functionality to the students and instructors who will use the software.

Performance
To ensure a seamless and fluid experience for students, the server response time should be less than 500ms and ideally less than 250ms. On the client-side, Javascript execution should be less than 1s (with a variable timeout). The overall goal for performance is to provide a responsive gameplay environment for the user.

Usability
The application must provide a clean and clear interface for the necessary operations a user must perform. The presentation of options must be kept simple to avoid confusing our users (students, instructors, and administrators) which would detract from the experience.

Maintainability
To facilitate rapid application development, we will be using the Rails framework for Ruby for the majority of the web site. Rails automatically handles page requests and data management, in addition to many other minor details, allowing us to focus strictly on writing the application.

Rails also employs the “Convention over Configuration” paradigm, meaning that only non-conventional features need to be specified in the code. Everything else will be handled in a default manner as determined by the framework. This drastically reduces the size of the codebase, improves clarity, and helps enforce best-practices throughout the application.

In other parts of the application, we intend to abide by web standards and best practices, such as implementing standard REST-style interfaces and standards-compliant CSS, HTML and JavaScript.

Security
Security is an important but not crucial feature for this application. The consequences of security breaches are relatively low, since no sensitive information such as banking or medical records are stored. What is important is that the service remains robust against malicious or naive student code, which is handled by our isolated grader. Account passwords will be encrypted using standard password-encryption techniques; if time permits, OpenID will be utilized for authentication.

Interaction between clients and both servers shall be encrypted using SSL/TLS to avoid basic session hijacking attacks. Taken as a whole, we adopt the philosophy of utilizing simple, easy to implement security measures that provide a high benefit to cost ratio. A provably secure system is not required and is not worth the effort needed.

Hosting, Development and Scalability
The application will be developed and deployed using virtual machines, with different virtual machines fulfilling different roles (database, Ruby on Rails backend, remote code

execution/grading, etc). Using virtual machines with well defined roles for each VM, the application will be easier to scale out to a larger number of users if needed. For example, as the user count increases, more machines can be added to run student code input and perform grading.

Appearance

The application should appeal to the target audience in a positive and engaging way. Game characters and environments will be designed to be gender neutral to ensure the game appeals to a broad audience and encourages greater diversity in computer science. Music in the game will generally be friendly and upbeat. Sprites will be inspired by classic 8-bit art but may have modern touches. OpenGameArt.org will be a resource for finding open source classic video game art.

Grading Isolation

All submitted code must be run in a restricted sandbox to avoid attacks by malicious users and unintentional problems created by faulty student submissions. This is key to ensure a good user experience for the students, instructors, and administrators of the application. Our game logic will be implemented through a custom C/C++ library layer to provide additional isolation. The abstraction of this layer will also simplify the task of adding new languages, so long as they support an interface to C/C++.

Design Outline

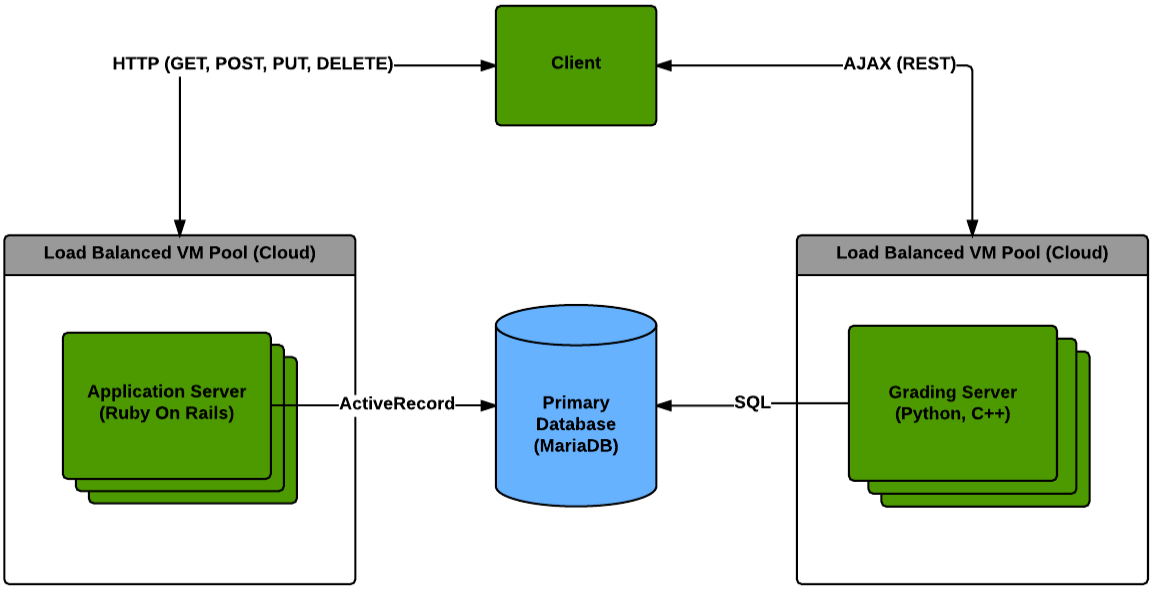
Overview and Components

The system will implement a client/multi-server architecture with the following components

- Application server
- Grading server
- Database server
- Client (web browser)

Architecture

The application will use a common client-server paradigm to provide separation of concerns between the client and server. This will facilitate easier division of effort between the development team members and allow the UI and backend components to evolve in a flexible manner.



Application Server

All of the user and class management functionality will be located within the application server component. This functionality will be implemented using Ruby On Rails within a load balanced set of machines. The application server will be responsible for serving dynamically generated pages to the clients.

Client

The client will be written in HTML and CSS and be mostly generated by the application server component of the project. A primary goal for the client is to be simple, easy to use, visually attractive, and responsive to users' needs. The client will interact with the application server for various management tasks, and will communicate with the grading server for game play mechanics.

Grading Server

The grading server will handle the responsibility of running submitted code in an isolated environment to determine the results of the student's code submission. A score or outcome will be determined, and a move summary will be sent to the client to be replayed for feedback to the user.

Design Issues

The following are some of the design choices that were made during the course of the planning process.

- **Design Issue #1** - Single role per-user or single role per-user per-class?
    - **Option 1**: Every user has a single role assigned across all classes
    - **Option 2**: Every user has a role assigned for each class they are involved with
    - **Choice**: We chose **Option 2** as reflected in the class description
    - **Discussion**: **Option 1** is unacceptable since we cannot give a single user such broad access rights across every class. It also prevents a user from being a student in one class, and an instructor in another class. Clearly, **Option 2** gives us the granularity we need for access control.
- **Design Issue #2** - Should students be allowed to use direct keypad input?
    - **Option 1**: Allow up/down/left/right keypad input to move the character
    - **Option 2**: Use submitted code to move the character
    - **Choice**: We chose **Option 2** for reasons details below
    - **Discussion**: **Option 2** was chosen because of the significant amount of additional complexity that **Option 1** would have introduced into the design. **Option 1** would also undermine gameplay mechanics designed to encourage more elaborate programming (e.g. utilization of for loops, path-finding algorithms).
- **Design Issue #3** - Self-enrollment or selected enrollment?
    - **Option 1**: Allow students to enroll in a class potentially based on instructor permission
    - **Option 2**: Require instructors to explicitly invite student via a name lookup
    - **Choice**: **Option 1** is our choice for the application
    - **Discussion**: **Option 2** allows for an instructor to choose who can enroll in their class, but requires modification to allow for certain special cases. For example, we plan on there being an open catch-all class for students who wish to practice and learn, but are not part of a formal program and thus have no instructor. **Option 1** allows this by letting an instructor invite students to closed classes by name, thus limiting the class to those chosen students.
- **Design Issue #4** - Should students be allowed to code in multiple languages?
    - **Option 1**: Hard code all functionality to one language and not allow other languages
    - **Option 2**: Create plug ins to parse student input to a common language which is run through the game logic
    - **Option 3**: Create a class in each language with bindings to a C++ library containing the game logic
    - **Choice**: We chose **Option 3** for reasons detailed below
    - **Discussion**: If we were to hard code the parsing of student code, as considered in **Option 1**, we would be locked into our decision. This is too limiting, as we have hopes for adding support for additional languages. **Option 2** requires creating our own scripting language to translate the source into. We chose to simplify this into

Option 3, which involves no parsing, only direct calls to a “glue” class which forwards them to a library call. This allows for additions of features or languages by creating or modifying these intermediate classes, of which there is only one per language.

- **Design Issue #5** - How should the game engine be designed - with or without a game library?
    - **Option 1**: Design a JavaScript game engine from scratch
    - **Option 2**: Utilize an existing JavaScript game engine framework
    - **Choice:** We chose **Option 2** for the following reasons.
    - **Discussion:** Due to the time constraints placed on this project, **Option 2** was decided to be the best choice. **Option 1** would allow greater flexibility but require significantly more development time to implement common game mechanics, such as sprite layout, z-ordering, animations, and textures. JavaScript game engine frameworks provide these features out of the box. Of these frameworks, Phaser was chosen due to its popularity, wide user base, and positive acclaim among developers.
- **Design Issue #6** - How are challenges related to levels?
    - **Option 1**: A Challenge is a type of level
    - **Option 2**: A Challenge is linked to a level
    - **Option 3**: Challenges are entirely separate from levels
    - **Choice**: We chose **Option 3** for this issue
    - **Discussion**: Challenges’ actions are very similar to those of levels. However, challenges are not necessarily tied to the same resources (tilemaps, storyline, etc.) as levels, and therefore may require different handling in some cases. Therefore, **Option 3** was chosen, as it allows us to be more flexible with the inclusion of challenges across the application.

Design Details

Components

Client

A browser-based front end to the system, to provide ease-of-use in management of the system, as well as the game interface.

The client will be implemented using HTML, CSS, and JavaScript. As supplements, we will use the jQuery library to simplify the JavaScript programming, the Phaser game engine library to help support the game interface, and the Twitter Bootstrap framework to improve the site interface.

The majority of the content will be delivered to the client in response to standard HTTP requests to the Application Server. Code submissions for levels and challenges will performed through AJAX calls to the grading server to provide responsiveness.

The game interface will be implemented using the Phaser JavaScript library to display visually appealing maps for users to interact with through code entered into the user interface. Phaser creates an HTML5 canvas painted with either Canvas 2D or WebGL API calls. Game layout and user interaction occurs on the client. When the user submits code, the code is sent to the Grading Server using AJAX through a REST API and a list of execution steps is returned.

The client uses these execution steps to animate and update the game state in the canvas. After a student successfully complete a level, the results will be recorded into the database for permanent record keeping. When a user logs into a class, the game state is fetched from the Application Server and loaded. The web application will make a best effort to synchronize the game state on the client and the game state stored in the database.

Application Server

The Application Server will be hosted on one or more machines (virtual or physical, with multiple machines fronted by a load balancer) that serve content to the client.

This server will be implemented using Ruby. It will use the Rails framework to handle general web-server functionality and data management, the HAML language to simplify View markup, and the SASS language to simplify CSS style definitions.

As extensions to Rails, we will use multiple Ruby gems, including the Devise gem for user authentication, the Rolify gem to add roles to users, the Cancancan gem for permissions management, the Thin gem as the Rails server core, and multiple other gems to aid in development, testing, and production efficiency.

Database Server

A single highly performant physical or virtual server to store static data such as user accounts, roles, classes, game levels, and user submissions. We will use MariaDB for the database as it is open source and similar to MySQL. Adapters for both Python and Ruby On Rails are available for MariaDB, so no significant compatibility issues are expected.

Grading Server

The Grading Server will consist of multiple virtual machines fronted by a load balancer. Each virtual machine is dedicated to executing user submissions and returning a result back to the user.

This server will be implemented using Python with the Flask framework and individual Docker containers will be created to host user sessions. Docker containers are a lightweight form of virtualization that utilize a number of Linux kernel features to create the illusion of a new virtual machine, but are significantly more lightweight than a virtual machine. Each grading server instance will control many Docker containers to run code within a resource and security controlled environment.

The Grading Server will associate a given user session with a specific Docker container instance. For capacity purposes, multiple user sessions can live within a single container, but will be kept separate by running each session with different read-only user accounts within the container. Grading requests will be sent into the container, along with additional data such as the level layout, etc, and the code will be run to evaluate the success or failure of the submission. The results will be returned from within the container and be processed by the grading server before being returned to the client.

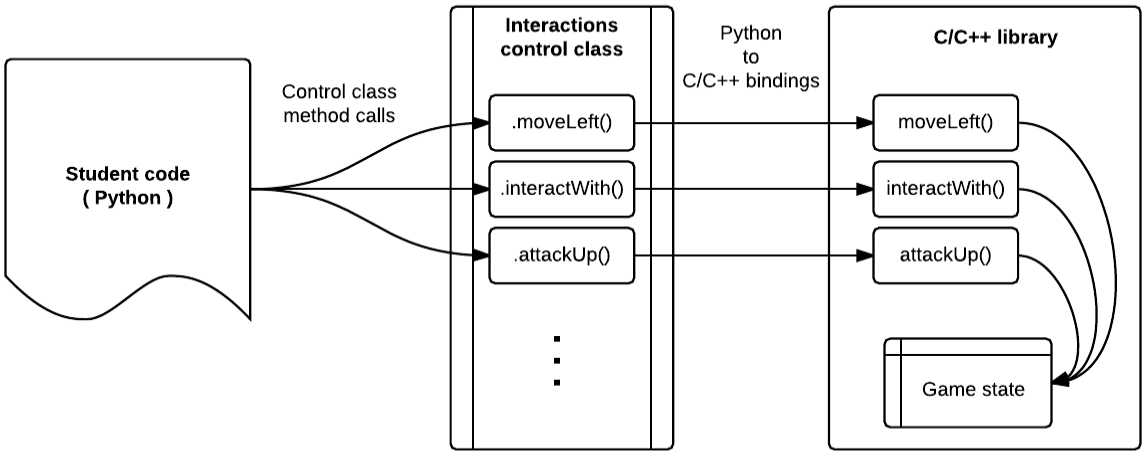
Game Logic (Client)

The user will be presented with various programming puzzles and challenges as he or she progresses through the game. To interact with these, the user will use the “Console” which slides in and out from the right side of the screen. The Console has two major sections: the explanation section and the coding section. The explanation section will contain text which introduces and explains the puzzle. The coding section provides an editor-like text box in which the user types the code which solves the puzzle.

There may or may not be starter code (stored in the database) already given to the user depending on the level. Small buttons will be available to select hints if a user needs help. When the user is satisfied with the code, he or she presses the “Submit” button to execute the code. The code is passed to the Grading Server; either an error or a list of execution steps is returned. If an error is returned, the user is notified with specific feedback on how to fix the code. If a list of execution steps is returned, the client animates and updates the game state appropriately.

Game Logic (Grading Server)

After a student submits their code to the Grading Server, the code will first pass some basic sanity and validity check (Mainly to filter extremely large submission and confirm request parameters). After those checks, the code will be submitted to the container where it will run. The code the students write will interact with the game level through a custom Python library. This custom interaction library will wrap around a C/C++ library containing the actual gameplay logic and game state. Abstracting the game logic implementation into a C/C++ library gives several advantages.



First, it allows us to reuse the exact same library to add support additional languages, provided they have a sufficiently capable binding to C/C++. The allows for rapid extension to new languages based on customer requirements. Another benefit of this design is that it helps provide additional isolation.

Python and many other programming languages are very “dynamic” and don’t offer good ways of hiding state that should be kept hidden. This means that if we implemented our game logic directly in Python, then students could “cheat” or break the game by directly accessing and modifying the game state. The C/C++ API library would not suffer from that same problem, since students cannot directly access or corrupt the game logic state.

Security

Security is important for user trust and retention, but is not a high priority feature. Basic common sense security features will be implemented, such as SSL connections and secure user session support.

Session Initialization

The Devise gem for Ruby on Rails will handle the creation and maintenance of users' sessions as they access the application. This gem provides full functionality out-of-the-box, including user creation, authentication (with options for OmniAuth and OpenID), persistence, and more.

Session Persistence

The Devise gem (mentioned above) also allows for session information to be stored in the database. By passing a user's credentials between different parts of the application, we can validate their session at each point in the application's flow.

Primarily, this will be used to validate users' submissions to the grading server. Once the grading server has validated a session, it can correlate the submission to the user and save the resulting record to the database without relying on the application server.

Roles

To provide appropriate access control to the various features and settings, a role based access control mechanism will be implemented using the Devise and Rolify gems for Ruby On Rails. Roles are assigned on a per-user, per-class basis. More simply as an example, a user can be a student in one class, and an instructor in another. Additionally, a site-wide instructor or site-wide administrator status can be granted to select users.

Student

The student role grants access to the levels within a given class, including viewing game progress, customizing his or her character, and viewing earned badges for all students enrolled in the class.

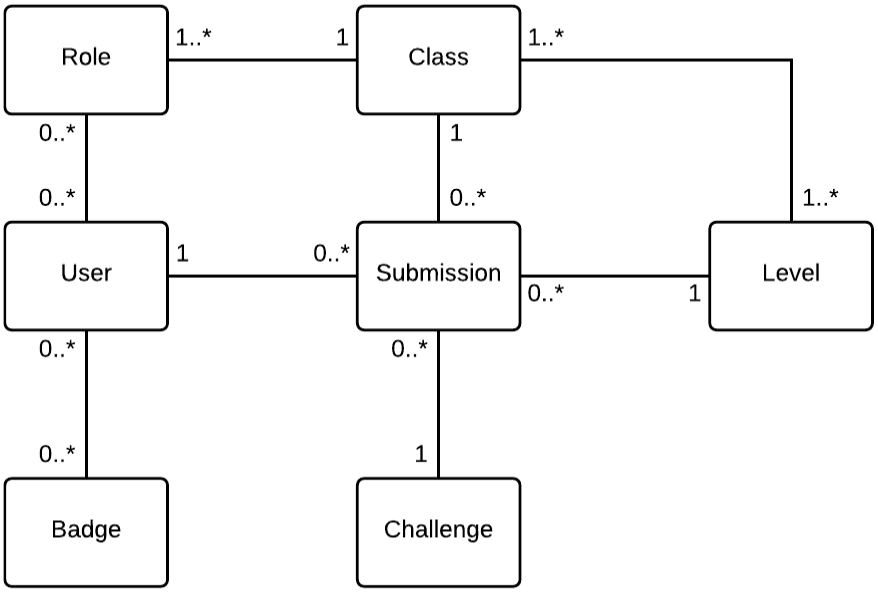
Instructor

The instructor role gives a user the ability to create new classes, modify class settings, view student submissions within the class, approve or deny student enrollment in a private class, and manage other class settings.

Administrator

The administrator role gives a user the power to add or remove classes and students from the website at will. The role of administrator can only be granted by another administrator. With great power comes great responsibility.

Domain Classes



User

A User represents everything related to a user of the application, including their name, email address, password hash, etc.

Class

A Class represents a set of levels by a given set of instructors. Instructors have full control of the class. Users can also enroll in classes as Students. For Classes with closed enrollment, Students initiate an enrollment attempt, and the instructor will authorize the attempt.

Role

A Role represents the role of a User in a Class. Possible roles are Student, Instructor, and Administrator, each with their own permissions. Roles are not universal, meaning a User can be a Student in one Class, and an Instructor in another.

Level

A level represents some portion of a Class, generally including a coding activity and part of a continuous storyline for the class. Levels are required to include a tilemap definition that will be shown to the user while they play the level.

Challenge

A challenge is a special coding activity, either included as part of a level, or separate from a level that does not use the graphical game grid. Challenges are more involved and difficult than levels.

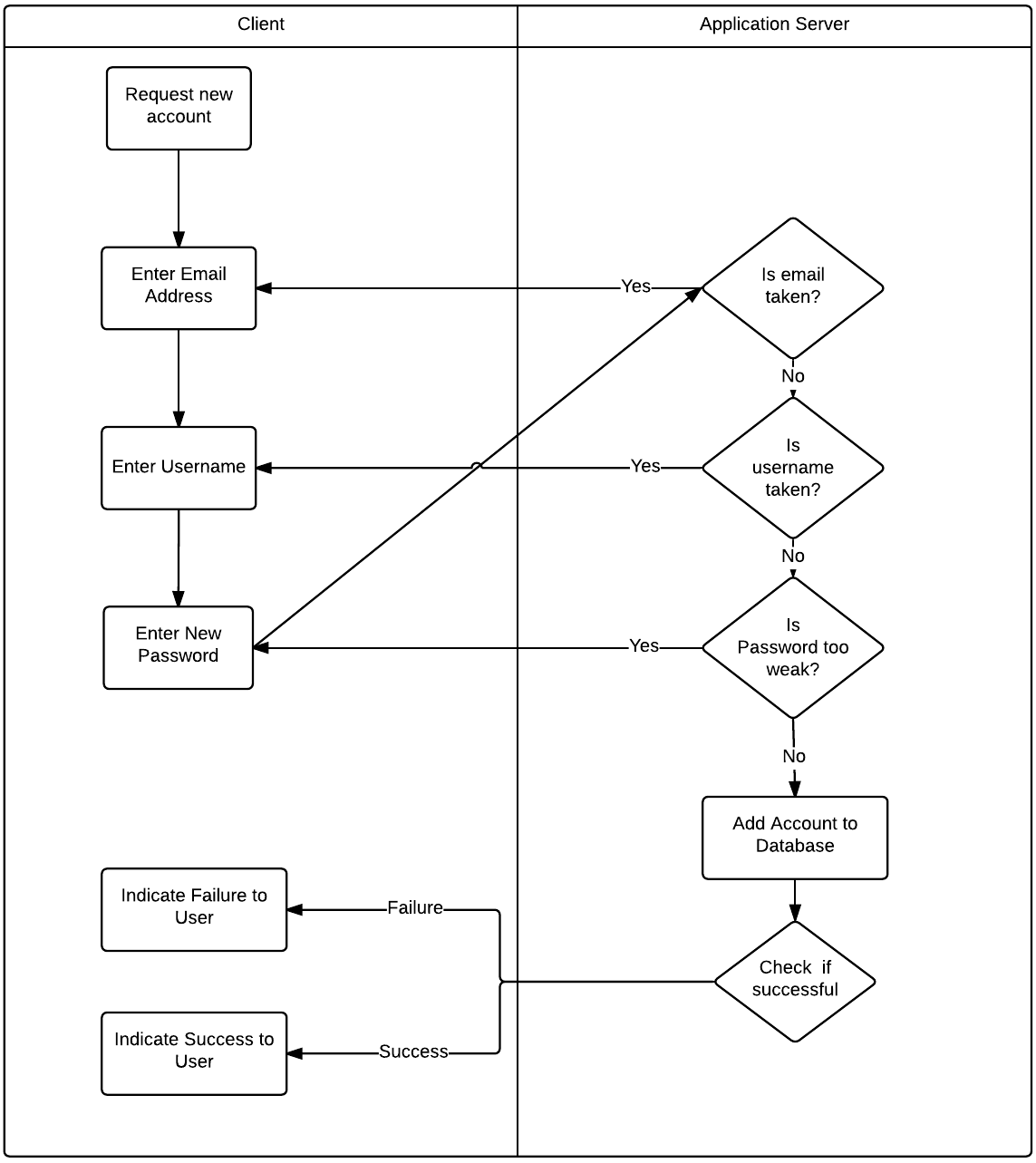
Badge

A Badge represents a reward for a user. There is an established list of badges that can be earned, and a user unlocks certain badges as certain tasks are completed or certain levels are completed.

System Interactions

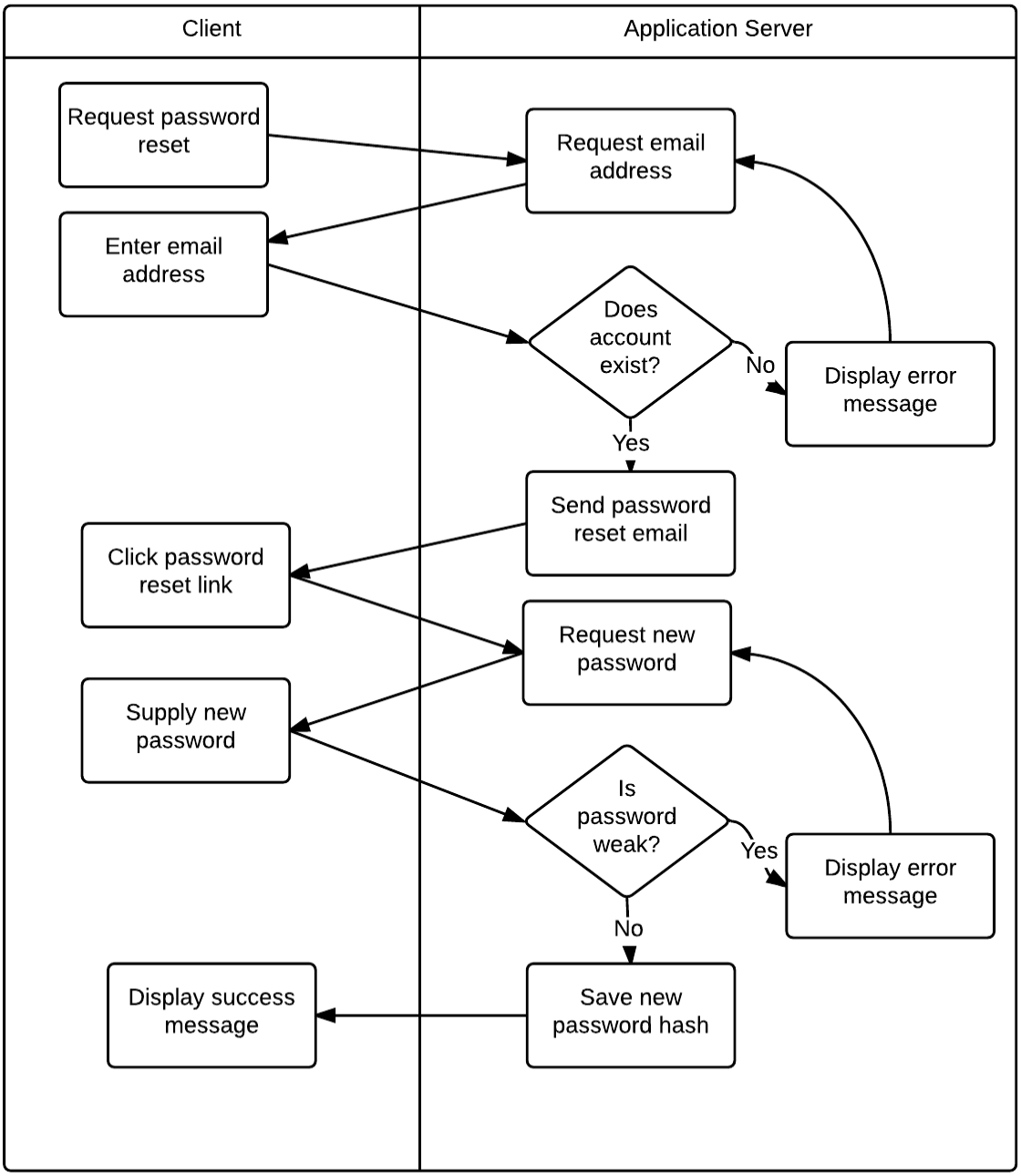
Creating a new user

The first step every user must perform with the application is to create a new account. The basic workflow to create a new user account is given below.



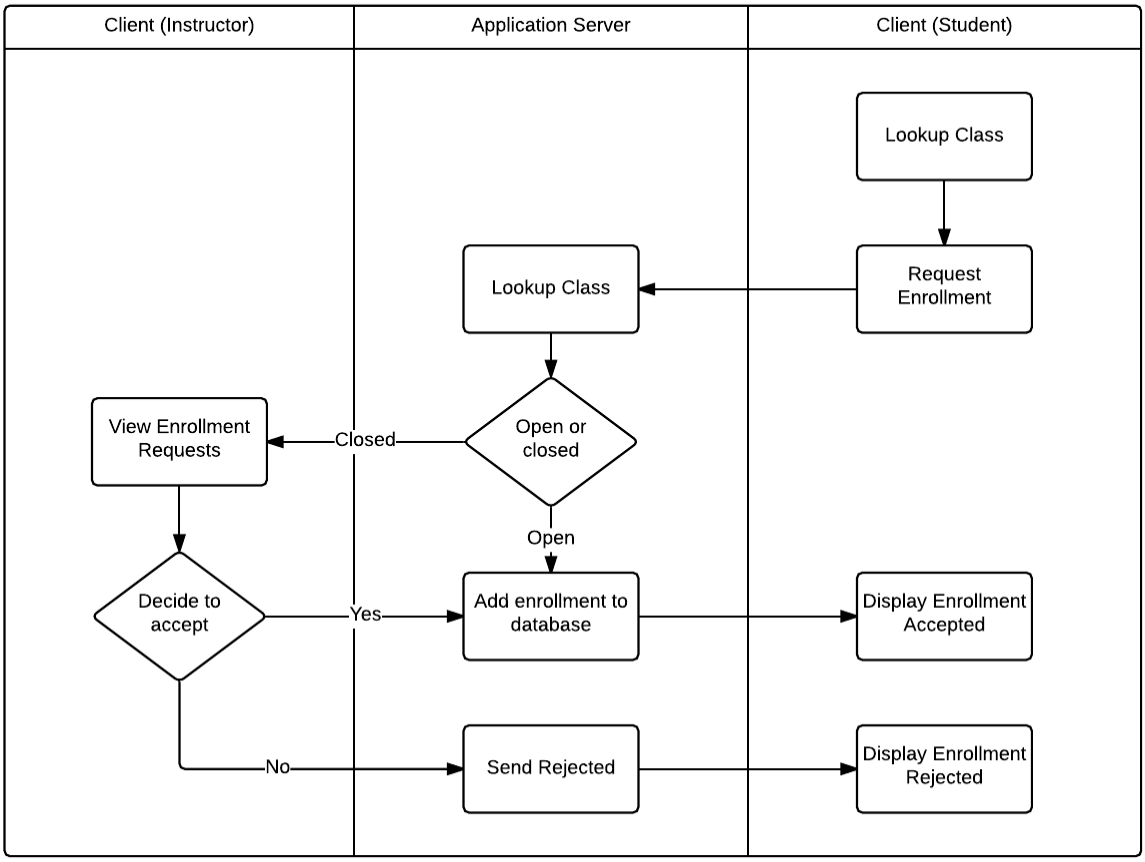
Resetting a user's password

In the unfortunate event that a user forgets their password, a mechanism will be provided to allow a user to reset their password through the email address they provided when they created their account.



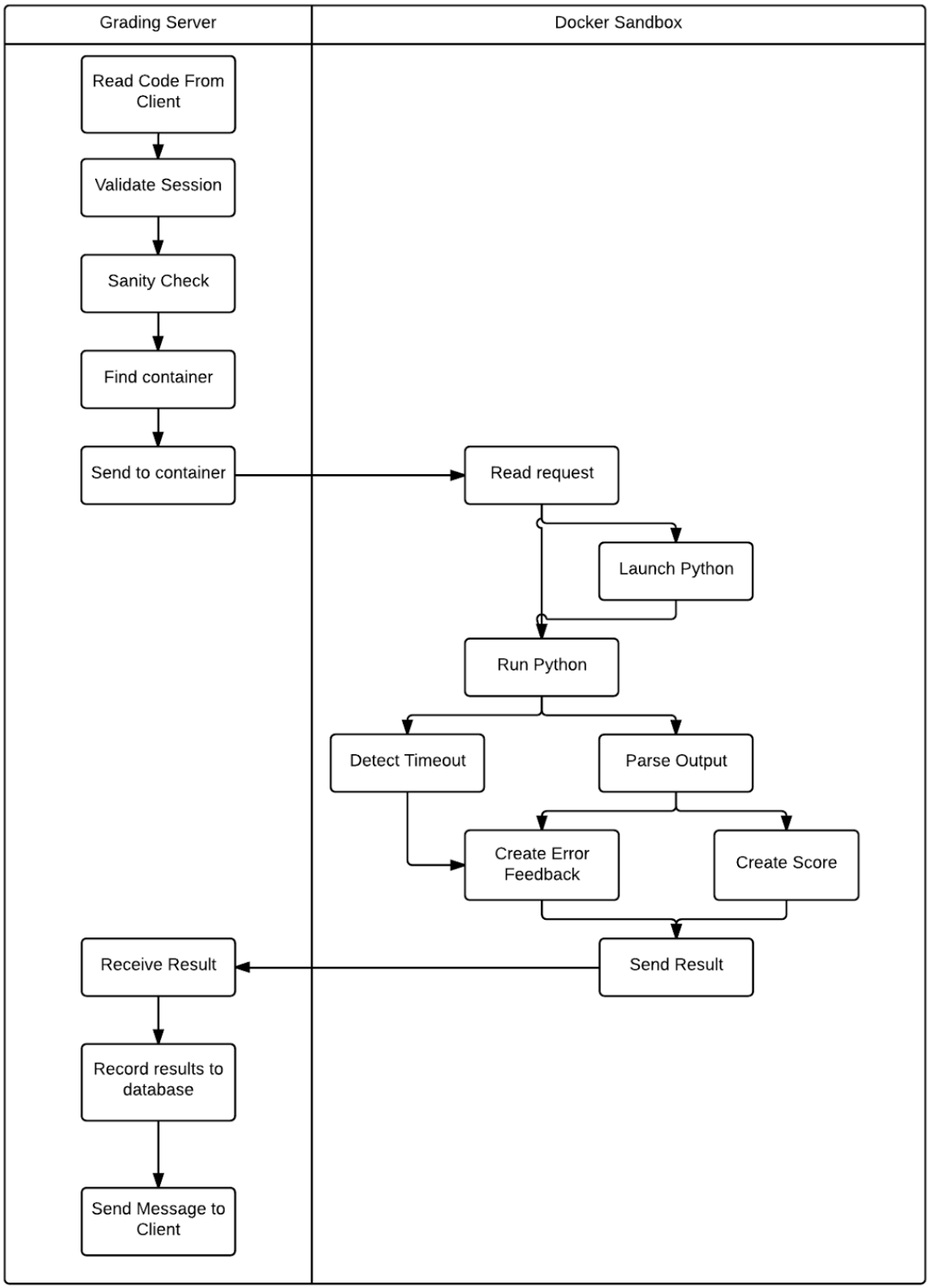
Enrolling in a class

Enrollment in a class varies based on whether the class is open or closed enrollment. In an open enrollment class the user can directly enroll without instructor intervention. For closed enrollment classes, a user can request enrollment, but an instructor must accept the enrollment attempt.



Making a code submission

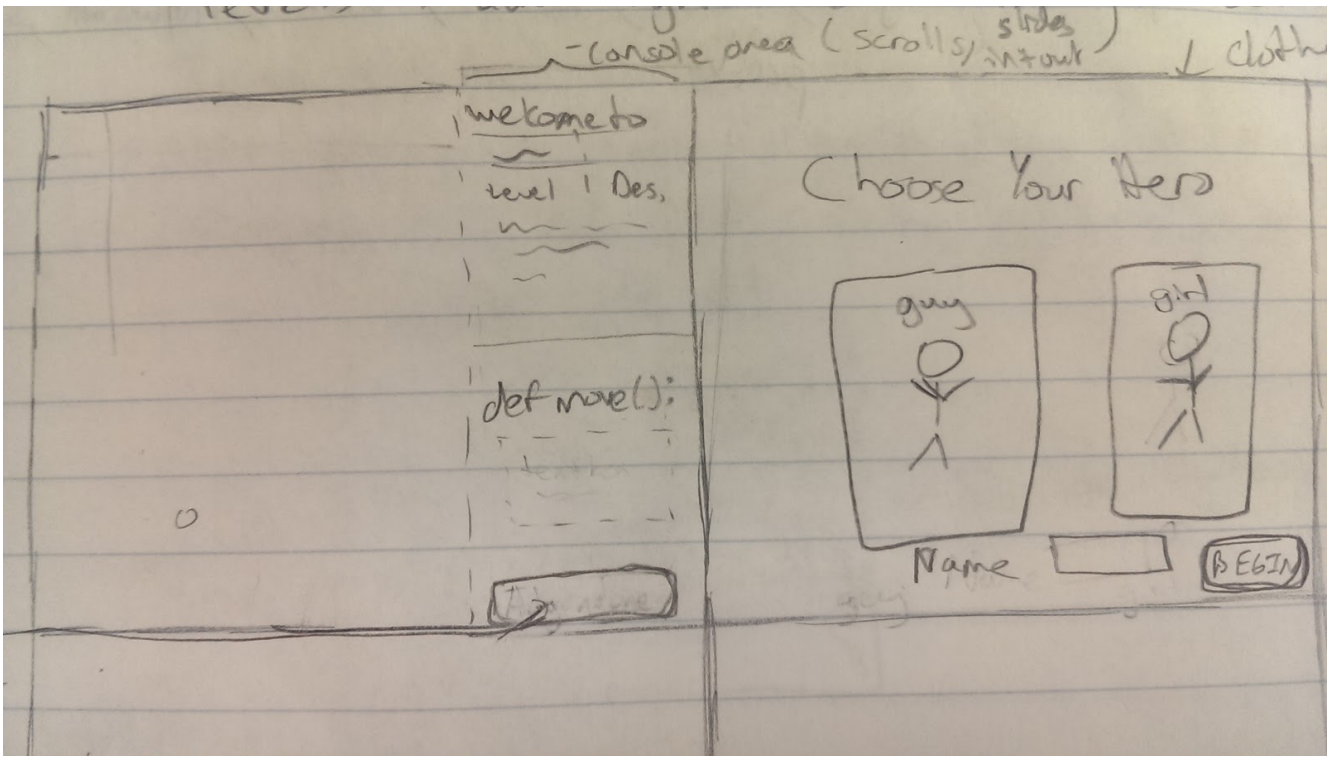
After a user submits a piece of code to the Grading Server, the code will be run within the sandbox. Both the submitted code and the level or challenge will be sent, and results will be returned from the container and subsequently be sent back to the requesting client.



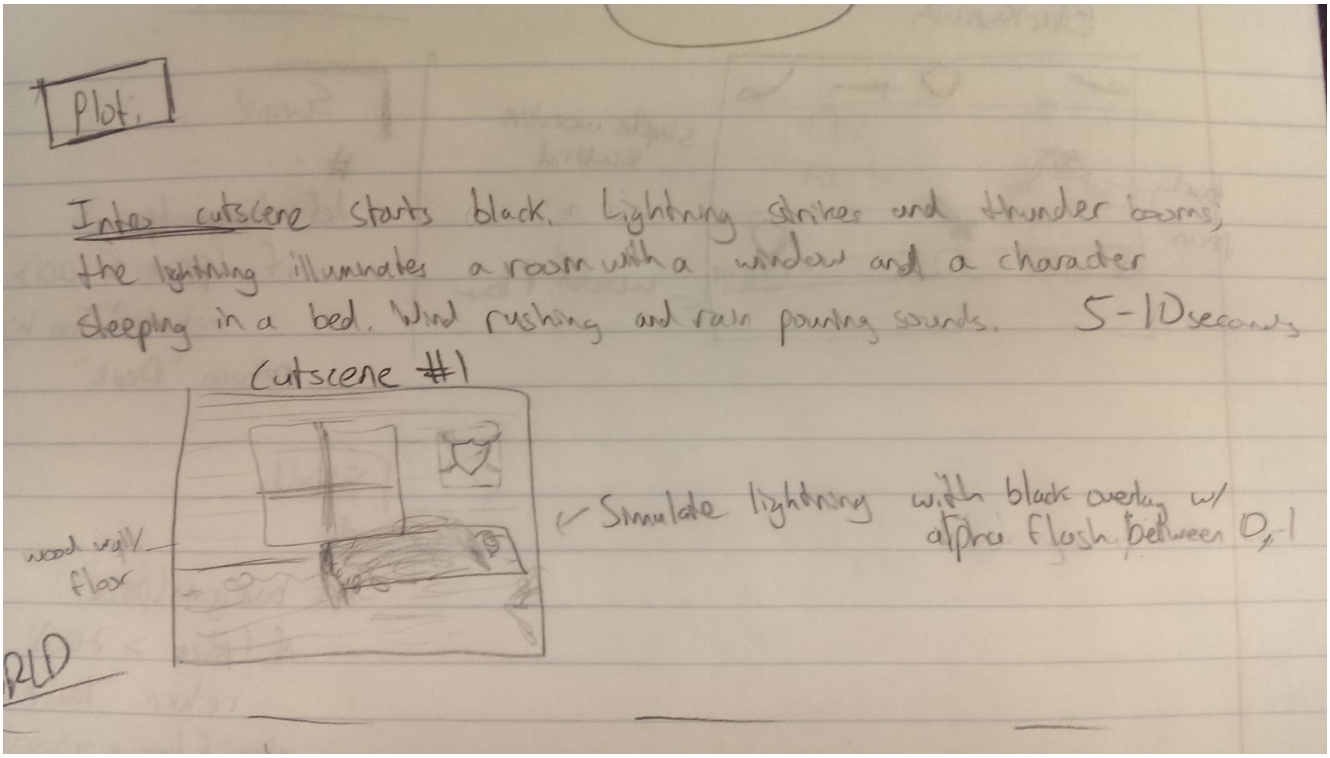
UI Mockups

The following are notebook sketches conceptualizing the client interface to the application.

Console & Character Selection



Console & Character Selection



Console & Character Selection