SENG 299 “Go” Project

Design Report

June 2016

Version 1.0

Prepared by:
Alex Laing <lainga@uvic.ca>
Alex Rebalkin <alexreba@uvic.ca>
Charlie Friend <cdfriend@uvic.ca>
Jia Xu <xujia@uvic.ca>
Tyler Harnadek <tylerhar@uvic.ca>

Table of Contents

1 Purpose

The purpose of this report is to provide a comprehensive architecture for the development of our Go web application, as well as provide any necessary background knowledge. The information contained within builds upon this project’s requirements specification.

2 Background

Go is an ancient board game in which players take turns placing stones onto a grid. Rulesets vary, but mostly focus on holding the most territory at the end of the game.

3 Design

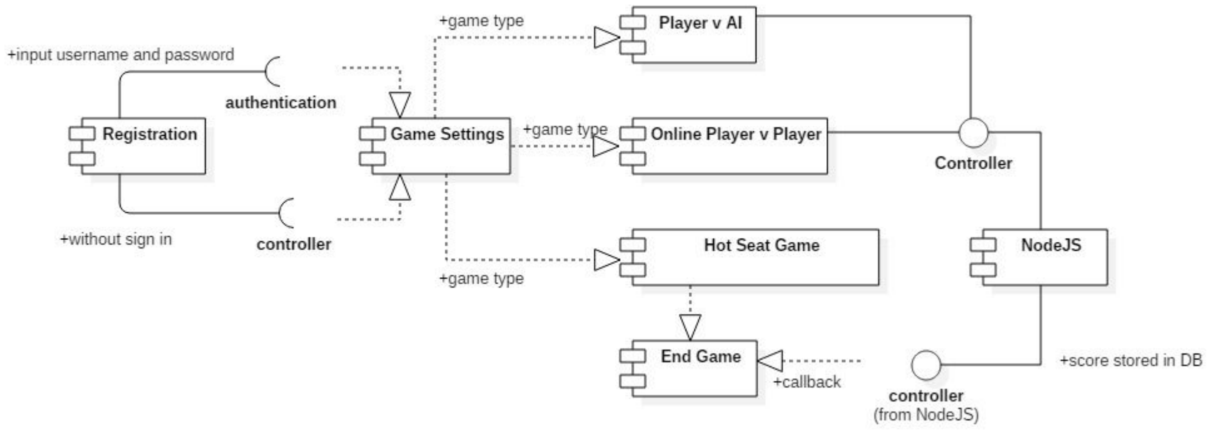
3.1 Design Process

The process used to arrive at the current design was mostly through in-person discussion. All team members were brought together to plan the project out. After defining the problem and reviewing relevant requirements, diagrams and potential ideas were sketched out on a whiteboard which gave the team an opportunity to further discuss the idea and determine its value to the project.

3.2 Software Modules

3.2.1 Front-end Web Application

Front-end as known as the view layer in MVC structure, it handles all the users’ inputs and give feedback to users as web-page form. The front-end component diagram as shown below:

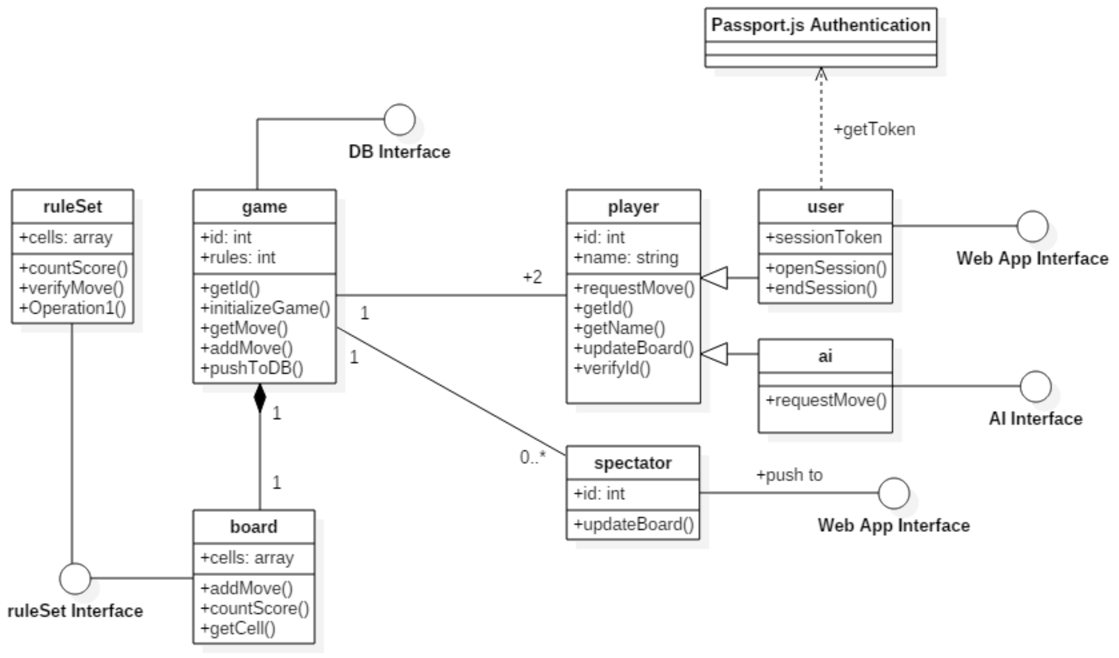


Based on the component diagram, client side portion of the Go web app will be implemented according to these steps:

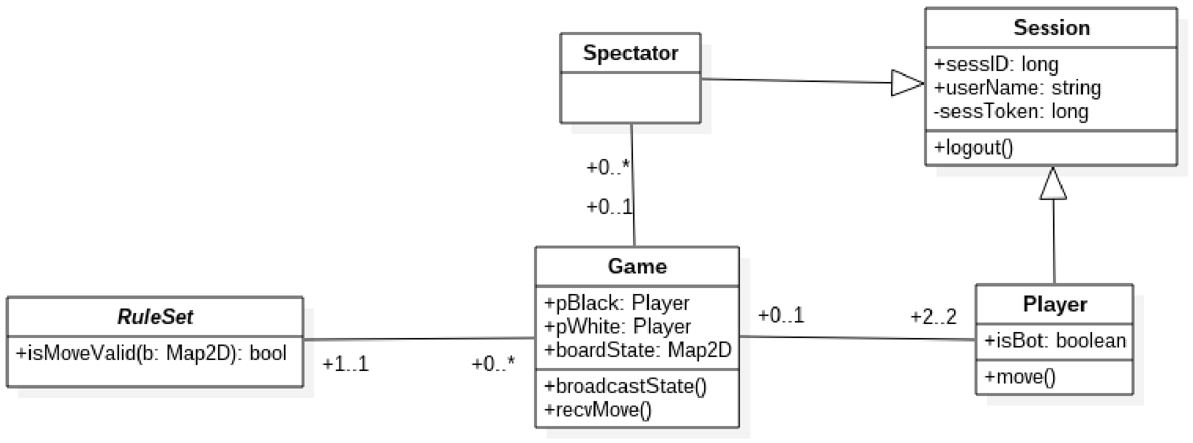
- Registration page: users are shown the registration page when they first open the website. The registration page also serves as the welcome page for the web app, it contains the website information, team description, sign up section, and sign in section.
- Game settings page: after authenticating player identity,, users are able to customize their game. Options include game appearance, game type, board size, etc.
- Gameplay page: this page will vary based upon the choices the user made on the last page. This is where the gameboard will be displayed to users.
- Ending page: This page is shown after a game concludes. It shows the score of the current game and statistics from past games.

3.2.2 Node.JS Server

The back-end logic of the application will be provided by a Node server running Express. The server will handle the routing of games, login management, enforcing of rule sets and recording of history.



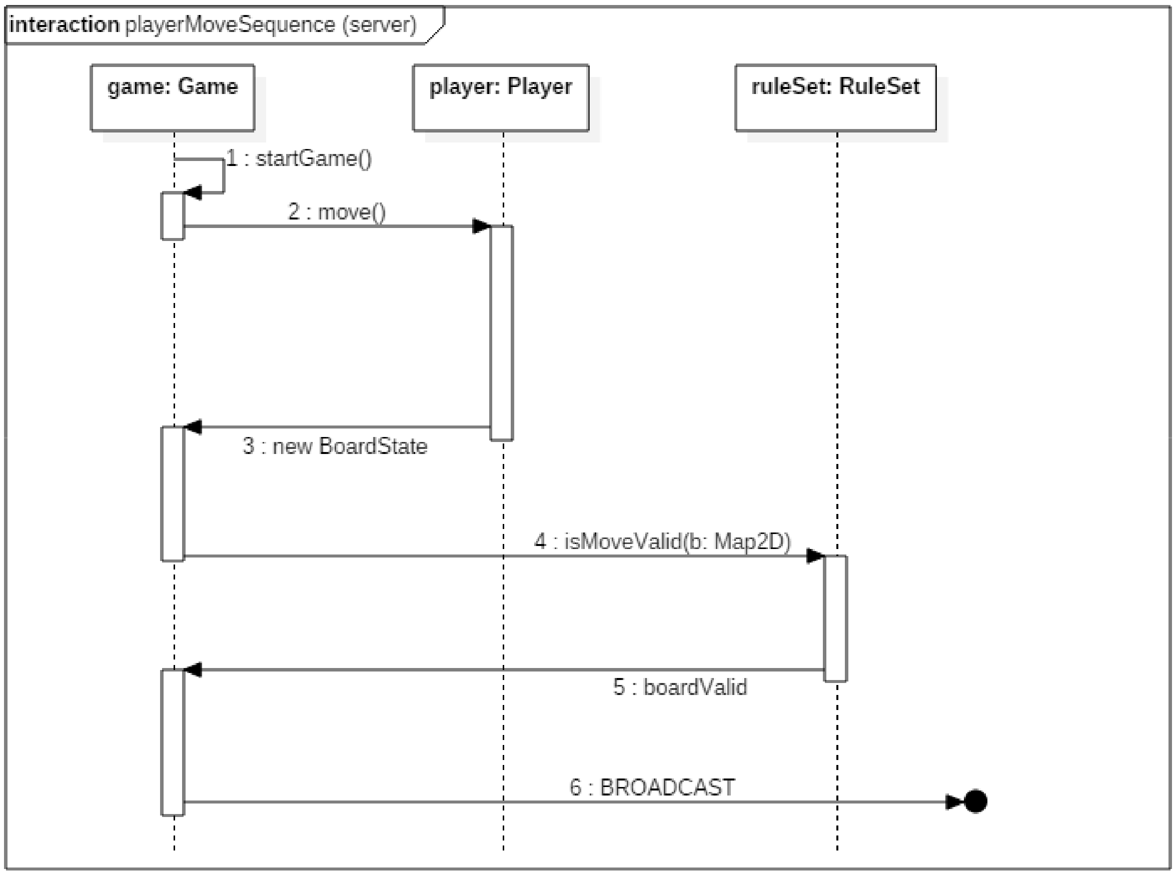
3.2.3 Game Routing



Games will be broadcast by the Node server to interested Sessions using the publish/subscribe design pattern. Active Sessions may be subscribed to at most one game at a time. Upon the game’s creation, two “player” sessions will be assigned to that game. Then, the game will be placed under the route `http://[SERVER URL]/game/[GAME ID].` Any active Session may subscribe to any game. If the Session is not one of the two players for that game, that Session will be classified as a spectator upon connecting.

Players make their moves from the client-side, by posting their move data to the server. As mentioned, a Session can be subscribed to at most one game at any time, so the server will automatically resolve which Game the Player wants to make a move to. Players can be controlled by either humans or some arbitrary AI; in both cases, they receive information about the current state of a Game through its `broadcastState()` method, and inform the server of their desired moves using the same `move()` method. For the sake of spectator and player interest, Players are assigned a flag to identify them as human or AI.

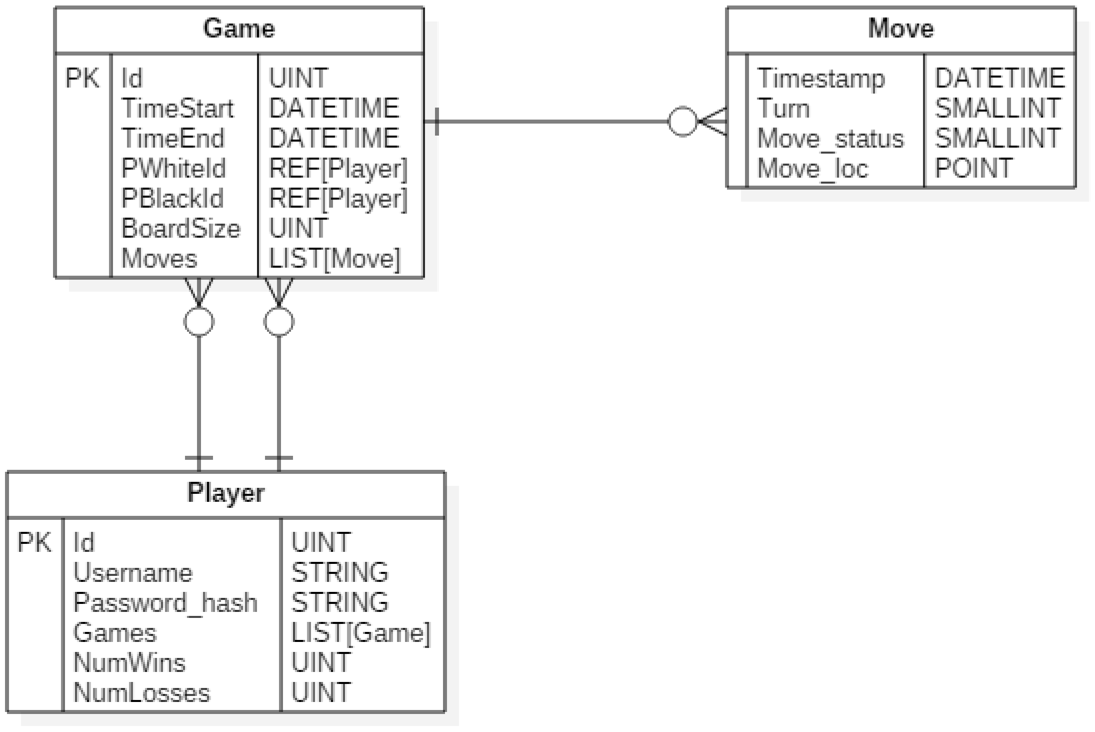
A single move interaction with a Player is illustrated below.



3.2.4 Mongo Database

The NoSQL database MongoDB will be used alongside this application to store user, game, and session data.

The database will store a list of user accounts, comprising the username, hashed password, and win/loss record of each user. The database will further store a list of active client sessions on the server, each of which is associated with a user - for authorization that the session may carry out an action - and a private session token, for authentication that a received request is from the session it claims to be from. Finally, the database will store a list of completed games from the past, allowing users to record, view, and replay them.

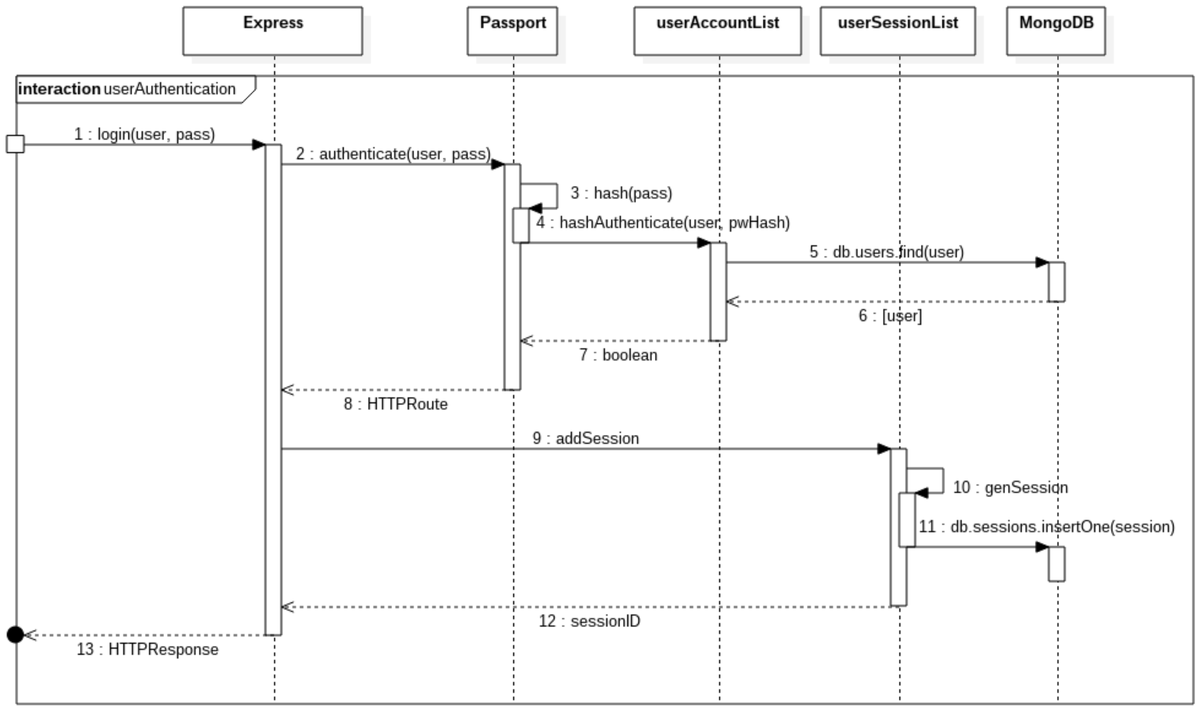


3.2.5 User Authentication

End-User authentication will be provided by the Node module Passport. User account information is stored on the application server. To connect to the server and start playing, the user client passes their credentials to the server’s login API endpoint. If the credentials given are correct, a new active Session is established on the server for that user. The server returns a cookie containing their Session’s sessionToken to the user. This cookie is stored on the user’s client browser, and passed with each subsequent request the user makes, to identify them for authorization purposes.

The possibility is open, in future, for user authentication using external accounts (Facebook, Twitter), using OAuth 2, which Passport is built upon. In future, eavesdropping attacks on passwords and session tokens should be protected by the use of HTTPS; however, HTTPS use and server certificate acquisition are not explored in this iteration of the design.

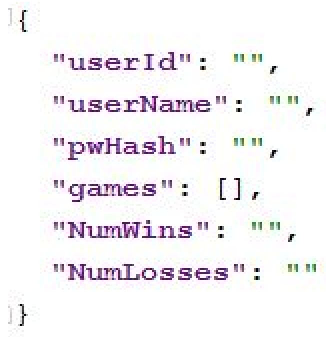
The user login and authentication process is illustrated below.



3.3 Data Structures

3.3.1 User Accounts

User account information is the primary table for this web application. All user account information should be stored as a JSON format. To be more specific, user information stored as an object. Users’ account data will be passed as JSON format to server when every time user sign in to the system, for example, the picture below shows the user account format:

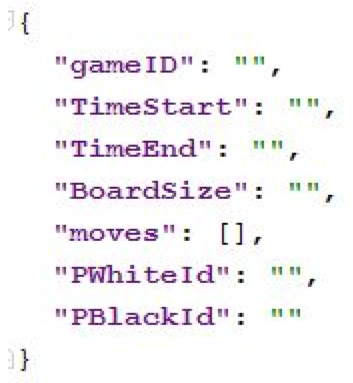

```
{
   "userId": "",
   "userName": "",
   "pwHash": "",
   "games": [],
   "NumWins": "",
   "NumLosses": ""
}
```

3.3.2 Sessions

The application uses the Express framework with Passport, which has a builtin module used for session data. Session information should be automatically stored when users sign in and are authenticated by Passport. Session information is checked for each game duration and security case. Sessions are stored in a JSON format similar to the user account information.

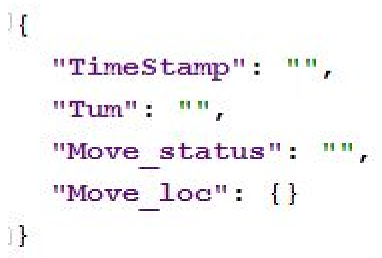
3.3.3 Games

When users start a game, three data diagram will be interacted. As previous ERDdiagram shows that the game will be conducted by three table respectively are game, player, and move. The data stored in game table looks as the picture below:



```
{
   "gameID": "",
   "TimeStart": "",
   "TimeEnd": "",
   "BoardSize": "",
   "moves": [],
   "PWhiteId": "",
   "PBlackId": ""
}
```

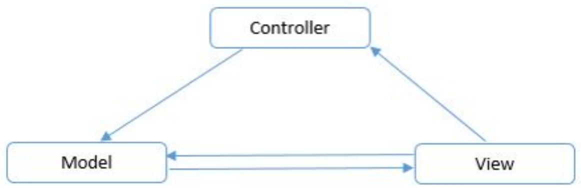
Data structure of Move diagram:



```
{
   "TimeStamp": "",
   "Tum": "",
   "Move_status": "",
   "Move_loc": {}
}
```

3.4 Design Pattern Use

The MVC design pattern will be used in this web application. MVC stands for model, view, and controller. The model layer manages data and logic, then pushes the information to the view layer where it is displayed to the user. The controller accepts input and converts it into commands for the model. The process is represented in the picture below:



Front-end web pages such as sign up/in, settings, and startGame are in the view layer, which can be viewed by users. The controller interprets all client requests. Authentication, filter, session data check will happen in the view layer. The model is used as to deal with data logic based on the requests from the control layer, then dispatches requests to specific file paths through the controller.

Games themselves are managed using a publish/subscribe pattern. In this configuration, the Node.js server represents the publisher, while players and spectators “subscribe” to changes in the board’s state.

4 Analysis

4.1 Risks

User passwords and session tokens are transmitted over HTTP, in plaintext. This leaves them vulnerable to a man-in-the-middle attack, where a third party reads the passwords or tokens. For tokens, the attacker would be able to spoof the user’s identity for that session; for passwords, the attacker would gain full access to the user’s account. This risk can be mitigated in future by use of HTTPS communication between clients and the server. Because of this risk, students and instructors testing the application are **strongly warned** not to reuse passwords from other accounts they have.

MongoDB is not ACID compliant. This means that it is “eventually consistent”, lacking the strict locking mechanisms of traditional SQL databases. For this reason, there are certain cases (power outage, simultaneous write/read) where the state of user and session information becomes undefined. At worst, this will typically affect only a few MongoDB operations, and can be manually rectified by an administrator.

The present design places no checks or limits on the number of sessions active on the server at any time. During high-load periods, users could experience slow and degraded service, or even denial of service. An attacker could create a large number of sessions to force such a denial of service. Future versions of the application should manage the number of active sessions, perhaps with reference to the server’s reported processor load.

4.2 Test Scenarios

In order to test for code-level errors, the system will use the Mocha.js automated unit testing framework. System-level validation testing will be conducted by developers and potential users.

Validation Test Case - Requirement 1

Ensure that users can play a game of “Go” using varying rule sets.

Steps:

1. Sign in to a user account.
2. Attempt to start a game against another player over the network.
3. Play through several turns.
4. Have both players pass their turns. The game should end at this point.
5. Ensure that the calculated score and victor is consistent with the selected rule set.

Variations:

- Play Go against an AI provided by the system.
- Attempt to make an illegal move during the game.
- Play Go using an alternative rule set.
- Disconnect from the server on one client machine and observe the system’s response on the other.

Validation Test Case - Requirement 2

Ensure that the system properly supports “Hot Seat” style play.

Steps:

1. Access the Go server.
2. Attempt to start a hot seat game.
3. Play several turns on the same machine. Ensure that the turns for each side alternate (WHITE, BLACK, WHITE…).
4. Have each player pass their turn.
5. Ensure that the calculated score and victor is consistent with the selected rule set.

Variations:

- Attempt to play an illegal move during the game.
- Play Go using an alternative rule set.

Validation Test Case - Requirement 3

Confirm that the use of 9x9, 13x13 and 19x19 boards are usable within the system.

Steps:

1. Set up a hot seat game, and specify a board size.

2. Confirm that the number of squares on the game board are consistent with the number specified during set-up.

Variations:

- Perform these same steps in network play or against an AI.

Validation Test Case - Requirement 4

Ensure that the system is capable of interfacing with a 3rd party Go AI.

Steps:

1. Set up a network game.
2. Select the opponent to be a 3rd party AI component.
3. Repeat the steps for testing Requirement 1 with the AI player.

Variations:

- Attempt to simulate an error in an internal 3rd party AI’s source code.
- Attempt to simulate a network failure in a 3rd party AI on an external server.

Validation Test Case - Requirement 5

Ensure users can create accounts or play anonymously.

Steps:

1. Log into a user account.
2. Start a game against any opponent.
3. Play a few moves to ensure functionality.
4. Log out of the user account and repeat steps 2 and 3.

Variations:

- Perform the same steps in a hot seat style game.
- Play a networked game against an anonymous opponent.

Validation Test Case - Requirement 6

To confirm users are able to make their own gaming styles in the customize settings section.

Steps:

1. Log into a user account
2. Go to game settings section
3. Browse to a background picture and successfully change the background
4. Select a different colour of stone
5. Play a game with a custom background and stone colour

Variations:

- Try a variety of different colour and background combinations.

Validation Test Case - Requirement 7

Ensure that the look of the Go board and pieces can be customized.

Steps:

1. Set up a “hot seat” style game.

2. Specify a visual style for the board.
3. Play through part of the game and ensure that the game’s visual style is consistent with the selected theme.

Variations:

- Attempt the same steps with varying visual themes.
- Attempt to modify the theme of a networked game or game against an AI player.