SENG 299
SOFTWARE ARCHITECTURE & DESIGN
Go Web Application

Designed and created by:

William Grosset - V00820930
Shreyas Devalapurkar - V00827994
Cailan St. Martin - V00826057
Justin Richard – V00773113
Wyll Brimacombe – V00826298

Table of Contents

1. Purpose

The purpose of this document is to provide a full technical description of the Go web application project and its development during the Summer 2016 semester.

2. Background Information

This collaborative project is a major component of the SENG 299 class at the University of Victoria. The group members are William Grosset, Wyll Brimacombe, Shreyas Devalapurkar, Cailan St. Martin, and Justin Richard. The following document will outline the technical work conducted for Milestone 4 and the overall development of the project. On July 22nd, our group had a successful demonstration of the application to other classmates and the course professor, Simon Diemert.

Some prerequisites to fully understanding the material presented in this document include:

- Milestone 1: *Requirements Document*
- Milestone 2: *Design Document*
- Milestone 3: *Group 15's Feedback Document*

3. High-Level Overview of Design

Our design has four major components: the server "hub", the front-end design, the back-end game logic, and the database. The server hub essentially provides an interface for the other three components to interact; all inter-component communications go through the server via POST and GET requests. The front-end provides a user interface and all interactions between the user and the system. The back-end game code handles all game logic, which then the database is used to store any data that needs to be persistent, such as user account information and game information (for saved replays). Figure 1.0 represents these relationships as a high-level component diagram.

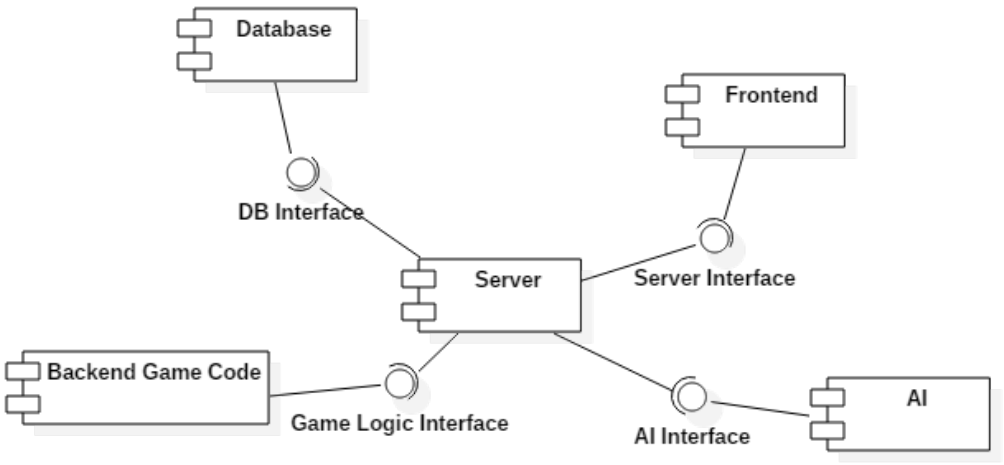

Figure 1.0. High-level component diagram

From a high-level perspective, our current design followed the original design that our team had discussed in the *Design Document* (Milestone 2). As shown in Figure 1.1, our original design shares strong similarities with our current high-level design.

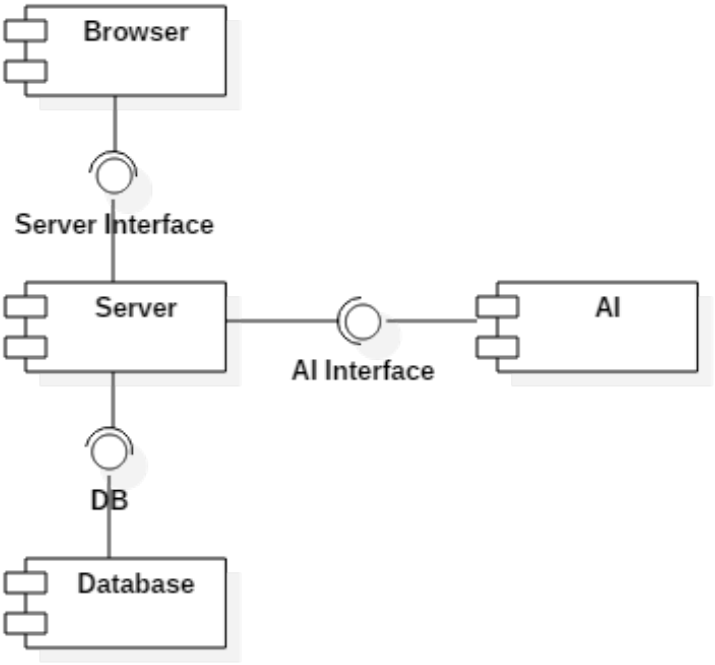

Figure 1.1. Original high-level component diagram

However, our design changed drastically between our initial ideas and our current implementation. Originally, our team constructed an object-oriented approach, influenced

by previous game development experience in languages such as C++. However, our team quickly realized that the interoperability of various components made this object oriented design more difficult. Additionally, experimentation with back-end algorithms using our original object-oriented design was extremely tedious. Additionally, our original design had all game logic hosted on the client side (see Figure 1.2). During implementation, we realized that a trusted client model could lead to manipulation and cheating. Instead, our team decided to host all of our game logic on the server side.

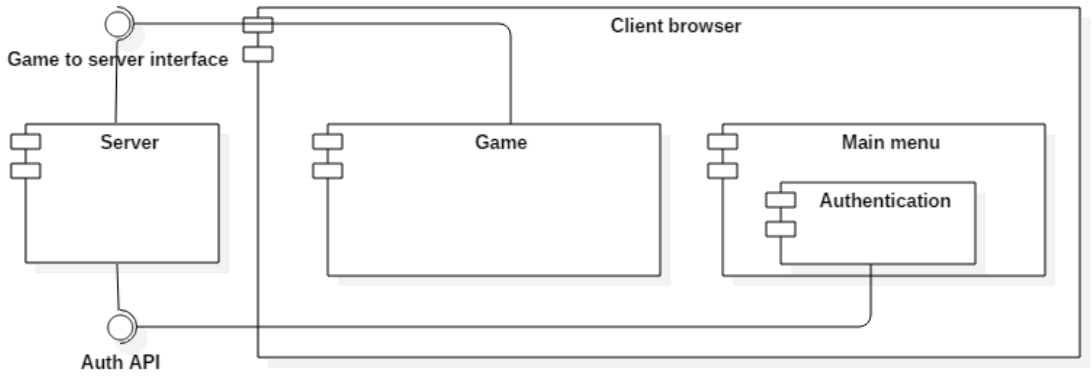

Figure 1.2. Original client-side component diagram

4. Implementation

Our implementation process began on June 1st, which included server skeleton code and the initial HTML mock-up of our front-end. By June 29th, our application had registration and database support with some added bootstrap CSS design. Leading up to our presentation date, July 22nd, our application was fully functional and ready to be demonstrated.

4.1 Server Hub

As the user navigates to different pages, the server is responsible for fetching the page data that the browser requires. As shown in Figure 2.0, the code snippet shows the GET request handler that fetches the “About” page data. Figure 2.1 displays the POST request handler that is used to start a new local game.

```
// About page
app.get('/about', function (req, res) {
    (req.session.username ? user = req.session.username :  user = '');
    var pageData;
    if(user != ""){
        pageData = {loginstatus: '<div><p class="navbar-form navbar-right loginstatus">
                    menu: ''};
    } else {
        pageData = {loginstatus: '<form class="navbar-form navbar-right" action="/actio
                    menu: '<li><a href="/login">Sign in</a></li><li><a href="/register"
    }
    var page = fs.readFileSync("views/about.html", "utf8"); // bring in the HTML file
    var html = mustache.to_html(page, pageData); // replace all of the data
    res.send(html);
})
```

Figure 2.0. Fetch about page data

```
// Create handlers for starting up a new game
app.post('/newLocalGame', function (req, res) {
    // Get game properties
    var userIP = req.connection.remoteAddress;
    console.log("Creating local game for IP: "+userIP);
    var boardSize = req.body.boardSize;
    var player1 = req.body.player1;
    var player2 = req.body.player2;

    // Create the game and redirect to it
    createGame(userIP, player1, player2, boardSize, res);
})
```

Figure 2.1. Creating local game

An important aspect of the server is to ensure that data is passed between each of the components consistently (the system is interoperable). The front-end, back-end, and database all need methods to receive or send game state information, which the server is responsible for. The most critical aspect to keeping consistent data communication throughout the system is the how the game state JSON object is sent. Since HTTP requests can be easily configured to send data as JSON objects, the game state data is passed between the different components (front-end and database). The formatting of this object is defined in our database code (see Figure 2.2.). This object is passed whenever game state data is required.

```
var game = {
      gameId: board.gameId,
      userIP: board.userIP,
      board: board.board,
      boardSize: board.boardSize,
      player1: board.player1,
      player2: board.player2,
      player1score: board.player1score,
      player2score: board.player2score,
      state: board.state,
      move: board.move
  }
```

Figure 2.2. JSON object for game state

`GameId` is a unique identifier that is generated at the beginning of any game, which is used to locate game data within the database. `Board` is a 2D array, where each entry in the array represents an intersection (0 = empty, 1 = player 1 token, 2 = player 2 token). `State` is a variable that represents the current state of the game. The possible values for this variable are shown in Figure 2.3. `Move` indicates which move number the game state is currently on, which helps with replays to be stored and accessed easily. The server also handles interaction with the external AI system. When a game is created against an external AI, the `player2` field of the game state is set to "AI" (see Figure 2.4.).

```
const blackTurn = 0;
const whiteTurn = 1;
const blackPassed = 2;
const whitePassed = 3;
const blackWon = 4;
const whiteWon = 5;
```

Figure 2.3. Different states for state variable

```
app.post('/newAIGame', function (req, res) {
    // Make sure user is logged in
    (req.session.username ? username = req.session.username :  username = '');
    if(username = '') res.redirect('/login?e=3');

    // Create the game and redirect to it
    var boardSize = req.body.boardSize;
    createGame(null, req.session.username, 'AI', boardSize, res);
})
```

Figure 2.4. Creating a new AI game

When a user places a move and the name of `player2` is “AI”, an AI move will be immediately requested from the AI server and played on the board (see Figure 2.5). To avoid conflicts, our application restricts player names to a minimum of 3 characters.

```
// If AI game, call AI move function
if(player2 == "AI" && (moveResult.state == whiteTurn || moveResult.state == blackPassed)){
    console.log("Requesting a move from the AI");
    getAIMove(moveResult, payload.move);
}
```

Figure 2.5. Requesting an AI move during an AI game

4.2 Front-end

A main feature of our front-end design was theme aesthetics. In order to create the various themes, multiple files utilizing Bootstrap CSS and HTML were used. Various CSS files (i.e. theme_summer.css, theme_wooden.css) were created by adjusting values such as “modal content” and “navbar.”

A `views` folder contains all html files for each of the screens that a user can visit during their use of the application. Within these html files, we have a `themePicker` id from which the different values (themes) can be selected (see Figure 2.6). Also, these HTML files create all the buttons, modal dialog boxes, and navbars that are visible on the pages of the application.

```
46    <select id="themePicker" class="form-control">
47      <option value="default">Default</option>
48      <option value="night">Night</option>
49      <option value="wooden">Wood</option>
50      <option value="cottoncandy">Cotton Candy</option>
51      <option value="christmas">Christmas</option>
52      <option value="radical">Radical</option>
53        <option value="nature">Nature</option>
54        <option value="space">Space</option>
55        <option value="medieval">Medieval</option>
56         <option value="winter">Winter</option>
57        <option value="love">Love</option>
58        <option value="summer">Summer</option>
59        <option value="puzzle">Puzzle</option>
60      <option value="work">Work Decoy</option>
61    </select>
```

Figure 2.6. `themePicker` theme options

The themes also affect how the board is presented to the user. The board builds itself based on the user’s choice of dimensions (scalable vector graphics). There are additional

opaque SVG circles that are overlaid on the board (see section 8.2 for hovering issue). Each circle contains a hover event that shows where a token can be placed.

Screenshots for each main user interface can be found in Appendix A of this document. The user interface screenshots highlight each major page a user would likely visit when accessing the application.

4.3 Back-end

The backend game code handles all game logic. This is presented as a module to the server, so that only the functions that the server requires are exposed. In particular, the `processMove()` function is exposed, while the rest of the functions are hidden as implementation details.

The backend game code is stateless and contains many utility functions. The majority of the functions require a game board to be passed as an argument (as well as other relevant information about game state). The `processMove()` function takes a game state object, a "move" (x position, y position, token colour), and the previous board state so the KO rule (see *Requirements Document* (Milestone 1)) can be checked. This function performs a number of tests: an initial check is to see if the move was a pass and updates the game state appropriately. If the move was not a pass, the function checks that the intersection being placed upon is empty, performs any captures that occurred, checks the KO rule for a violation, and then if the move was a suicide. If any of these tests fail, the function will return an unchanged game state. Therefore, the front-end will therefore not update as a result of the move, indicating to the user that the move was rejected.

Many of the algorithms used in the backend game code are variations of a recursive depth first search. The most complex function is to check if empty spaces are disputed or not. This function is used for scoring purposes at the end of the game. The function iterates over each empty space on the board, calling the recursive function, `playerSurroundingEmptiesInternal()`, on each unchecked empty space. Internally, this function uses depth first search to determine whether the "army" of empty spaces is exclusively surrounded by one player or not, and then flags these empty spaces appropriately. Once each empty space has been flagged, these spaces are iterated over once more and assigned to the appropriate player (or to neither, if the territory was disputed). The scores are calculated based off of this result and the board on the front-end will place tokens on the undisputed empty spaces to visually indicate which territory belonged to which player at the end of the game. As shown in Figure 2.7, the code snippet shows the first part of the `playerSurroundingEmptiesInternal()` function.

```
function playerSurroundingEmptiesInternal(board, x, y, surroundingPlayer)
{
    const notSurrounded = 5;
    const stillUnsure = 4;
    const flaggedAsChecked = 100;

    board[x][y] = flaggedAsChecked;

    var adjacentIntersections = getAdjacentIntersections(board, x, y);

    for (var i = 0; i < adjacentIntersections.length; i++)
    {
        var adjX = adjacentIntersections[i][0];
        var adjY = adjacentIntersections[i][1];

        if (board[adjX][adjY] != 0 && board[adjX][adjY] < flaggedAsChecked)
        {
            if (surroundingPlayer == stillUnsure)
            {
                surroundingPlayer = board[adjX][adjY];
            }
            else if (surroundingPlayer != notSurrounded
                    && surroundingPlayer != board[adjX][adjY])
            {
                surroundingPlayer = notSurrounded;
            }
        }
        else if (board[adjX][adjY] == 0)
        {
```

Figure 2.7. Initial if statement of `playerSurroundingEmptiesInternal()`

The back-end algorithm below (Figure 2.8) checks whether a specific move on a board would result in any captures and updates the board accordingly. This function utilizes the recursive algorithms: `isArmySurrounded()` and `removeArmy()` (also implemented in the backend game code). The function, `isArmySurrounded()`, uses depth first search to determine if the army that a particular token belongs to is surrounded by the opposing player. The algorithm determines whether or not that army has any liberties. To check whether a move would capture any opposing pieces, the algorithm tests each of the four intersections adjacent to where the move was played, checks if they contain an enemy piece, and calls `isArmySurrounded()` if they do. If any of these calls return true, a capture has occurred and `removeArmy()` is then called to remove the appropriate armies from the board (and update the scores). Since JavaScript object parameters are passed as copies of references, their contents can be modified within a function call. Both functions make use of these references and modify the board and scores arguments directly.

```
151  function checkForCaptures(board, move, scores)
152  {
153      var x = move.x;
154      var y = move.y;
155      var player = move.color;
156      var enemyPlayer = (player == blackPlayer) ? whitePlayer : blackPlayer;
157
158      var adjacentIntersections = getAdjacentIntersections(board, x, y);
159
160      for (var i = 0; i < adjacentIntersections.length; i++)
161      {
162          var adjX = adjacentIntersections[i][0];
163          var adjY = adjacentIntersections[i][1];
164
165          if (board[adjX][adjY] == enemyPlayer && isArmySurrounded(board, adjX, adjY))
166          {
167              removeArmy(board, adjX, adjY, enemyPlayer, scores);
168          }
169      }
170  }
```

Figure 2.8. `checkForCaptures()` function

4.4 Database

Our database is divided into three main collections: users, games, and history. Usernames and passwords (which are encrypted) are stored in the `users` collection. The process of registering an account is displayed as an activity diagram in Appendix B, Figure B.2. Game state is stored in the games collection after every move in any game (local, networked, or AI). Games are identified by their unique `gameId`, which is generated by the server. A single document is used to track the current state of a given game, while all previous game states are stored as individual documents in the history collection. The history collection is queried to implement the KO rule and the replay visuals.

A custom MongoDB class handles the interactions between the server and the database itself, which is exposed as a module to the server. For example, Figure 2.9 shows the `addHistory()` function, which is used to add a snapshot of a game state to the history collection.

```
// Add a games history state
addHistory(item, callback) {
    var collection = this._db.collection("history");
    item._id = null;
    collection.insertOne(item, function(err, result) {
        callback(err);
    });
}
```

Figure 2.9. Adding game state to history collection

As discussed in section 4.1, documents in the database require a defined structure so that data can be interpreted properly by each part of the system. These structures are defined within the MongoDB functions and have to be properly designed when communicating with other components of the system.

4.5 Design Patterns

From a high-level, our current implementation resembles a **service-oriented architecture**. Our four main components all provide services and interact with each other when requested. For example, the server utilizes the database to store persistent user and game data. Also, the server uses the backend game code to evaluate the validity and resulting game state of any move in a particular game. The server also acts as a provider to the front-end, handling appropriate user inputs that the front-end receives.

The **facade pattern** is used to facilitate the interactions between these components. The majority of the back-end game code involves recursive algorithms and numerous function calls to process a valid move. However, the other components (the server and the frontend) only send a game state and a move, and receive an updated game state, which may or may not involve rejecting the move. The front-end does not need any of the information regarding game logic or move processing. Therefore, our team used the "module" concept in JavaScript. This exposes our server to only the functions that it needs from the back-end game code, hiding the implementation details; in particular, the back-end's `processMove()` function.

4.6 Architecture

All diagrams can be found in Appendix B of this document. The package diagram (Figure B.1) is a representation of the file containers in the Go web application. The diagram also illustrates how all components are organized. The activity diagram (Figure B.2) outlines the actions that occur when a user attempts to register for an account. The user enters their desired username and password into text fields on the browser and the information is sent to the server to be handled. Firstly, the entered username is sent to the database to determine if another user has already taken it. If so, an error message is returned and the browser will update to inform the user of the error. If not, the server proceeds to add a salt to the password and encrypt it (using a third-party SHA-512 hash function). This encrypted password, along with the username, is sent to the database for storage. The user data is saved on the database and the browser is then updated to notify the user that they were successfully registered. The sequence diagram (Figure B.3) shows the steps involved in a successful token placement in a networked play game. There are

numerous places where the move could fail: if the player was not logged in as the correct user, or if the token was placed in an illegal spot. However, for brevity, these cases were omitted from the diagram.

When the user clicks on an intersection, the front-end determines the x and y position within the board that was clicked. This information is then sent to the server for processing. The server first queries the game state info (such as which accounts are playing, the current score, and the current layout of the board) from the database. The server then validates that the move was sent from the correct user account and retrieves the previous board state information from the database that will be required to check the KO rule. All this info is then sent to the back-end game code where the game logic is processed. The move is confirmed to be legal, any captures are made, and scores are updated appropriately. The new, updated game state is then returned to the server, stored in the database, and sent back to the front-end so that the board can be redrawn.

5. Functional Requirements

The following functional requirements were met that were specified in our original design document:

- Basic Gameplay
- Hot Seat Play
- AI Play
- Board Sizes
- User Accounts
- Game Replays
- Board Aesthetics
- Token Aesthetics
- Main Menu
- Running Score
- In-Game Theme Change

The following functional requirements were not met exactly as specified, but were slightly modified during the implementation process:

Functional Requirement	Modifications/Reasoning
End of Game	The requirement that an “exit application” option be shown in the End

Options	of Game Options was omitted, as it was considered redundant for a web application that runs in the browser.
Settings	An explicit “settings” page was omitted. All aesthetic options are accessible via a drop down list at any time while using the application.
Favourite Theme	Rather than associating a favourite theme with a user account, any user can modify their theme, which is then saved in their browser cookies to be loaded when they next open the application.
Save Game Replays	Rather than prompting users to save replays, replays of games played are automatically saved to user accounts.

The following functional requirements were ignored entirely:

Functional Requirement	**Reasoning**
Handicaps	During implementation, our team decided that some of our other medium to low priority functional requirements, such as Network Play and Save Game Replays, would be more beneficial to the user experience than a handicaps option.

6. Design Process

Our group had followed a waterfall development process. This process involved stages of specifying requirements, design, implementation, and testing our application. Our team followed this process since we felt comfortable focusing on the required tasks for each milestone deadline. However, our team had started doing implementation early, as we wanted to get a steady start if we ran into some future issues (see section 4 for implementation details). As you can see from Figure 3.0, most of the programming was completed near the end of the term, as the early stages were planning stages. If we had taken an agile approach, the graph may appear more linear. As shown in Figure 3.1, the majority of our application consists of JavaScript. Our front-end design and back-end logic utilizes ES6. Also, our server utilizes Node.js with the Express framework. The Gantt chart (Figure 3.2) represents the milestone completion of the project development.

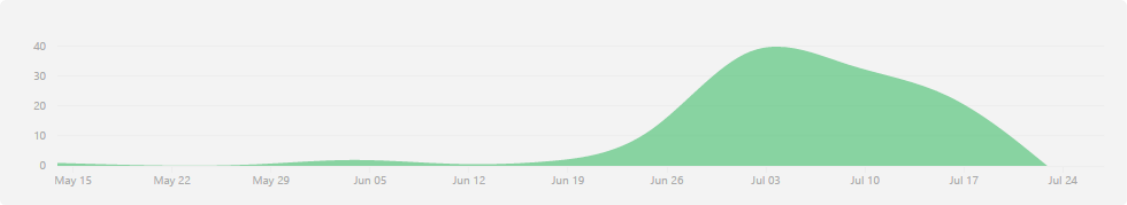

Figure 3.0. Code additions by date

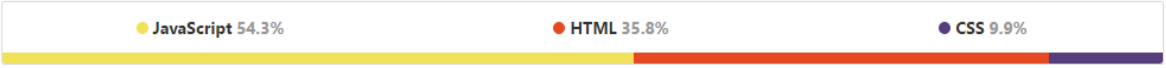

Figure 3.1. Programming languages breakdown

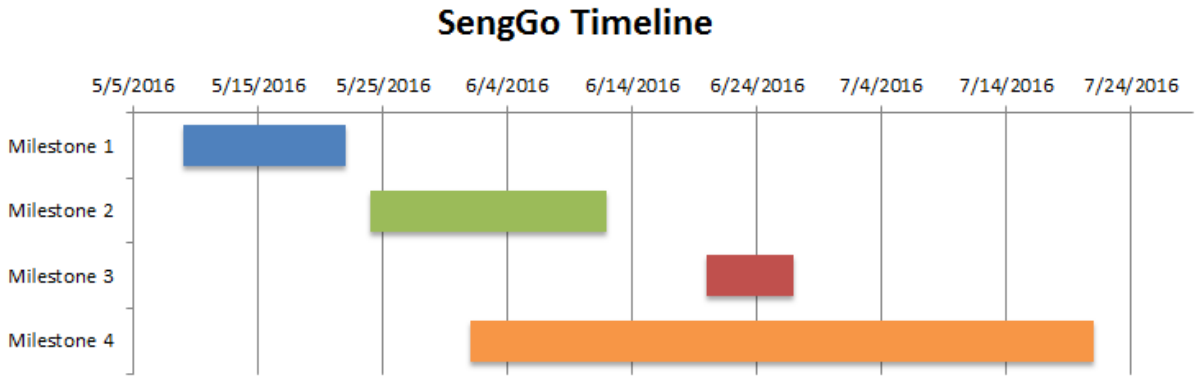

Figure 3.2. Gantt chart

7. Recommendations

Group 15 of the SENG 299 class created a recommendation document for our application based off of our Milestone 2 and 1 documents. Ultimately, our team decided to not use the recommendations given to us into consideration when developing the application. All of the recommendations our team received were in regards to the structure of the report and diagrams that we created, rather than actual design details. If our team decided to go back and re-model our original diagrams, our team would take these recommendations into consideration. However, since they did not provide any feedback about our actual design or future implementation details, our team was unable to utilize the recommendations when implementing our application.

8. Problems Encountered

8.1 SSH Constantly Running

When deploying our web application to the production server, roberts.seng.uvic.ca, our team ran into a reoccurring issue with SSH. The problem with SSH that we ran into is the VPN required to access the network auto disconnects after roughly an hour. This would not end the Node process, but would disconnect the SSH window from the process handler. Ultimately, this resulted in many processes that were causing the server to constantly be live. After a bit of research, our team utilized a few Linux commands (i.e. kill) that could be used to track the processes down and disconnect them.

8.2 Token Hovering

Currently, there is a hovering issue with our game board. While viewing an active game, the game board is refreshed periodically to provide the latest view. Each time we receive a board from the server, even if it has not changed, the front-end continues to redraw it. Our game board tokens use the HTML/JS `onhover()` event to draw the token outline. If the board is redrawn, the front-end loses the shaded token until the mouse moves at least one pixel, which will then fire the event again. The efficient solution is to only redraw the new board if there has been a change in the game state. This would then result the token to continuously cause a hover effect on the intersection and not disappear.

8.3 General Constraints

There were many other problems encountered during the implementation of this project. In general, working with entirely new programming languages introduced a major challenge. Each member had to stay on pace with our project roles, deadlines, and learn anything new that was required. Many of our debugging errors were caused by the Node.js server-side. Often, the bug was a simple capitalization or spelling error when attempting to send a formatted JSON object. With practice, these errors became easier to locate and our efficiency with fixing bugs greatly improved.

9. Project Roles

In order to efficiently work on our project, it was decided that our team should each have assigned roles and focus on development in a specific area. Our team developed the application in a modular style such that we then could each build our components and then connect them all at the end. The table below describes each member's assigned role with a short description:

Name	Role
Cailan St. Martin	*Backend game developer*: Worked on implementing rule algorithms, changing board states, and calculating scores (current and total). The backend developer focused on allowing smooth flow of moves behind the scenes with JavaScript.
Shreyas Devalapurkar	*User interface developer:* The UI developer worked with CSS and HTML to implement a usable, simple, glowing user interface and many themes.
Wyll Brimacombe	*Frontend developer:* The frontend game developer worked with JavaScript to draw game boards and send moves off to the server if they were valid.
William Grosset	*Frontend developer:* No site is great without a little interaction. The frontend developer worked with JavaScript, jQuery, HTML, and CSS to create interactive elements to enhance the user experience.
Justin Richard	*Backend developer:* Worked with NodeJS to connect the other elements of the project together and deliver pages to clients who visit the server.

10. Appendix A

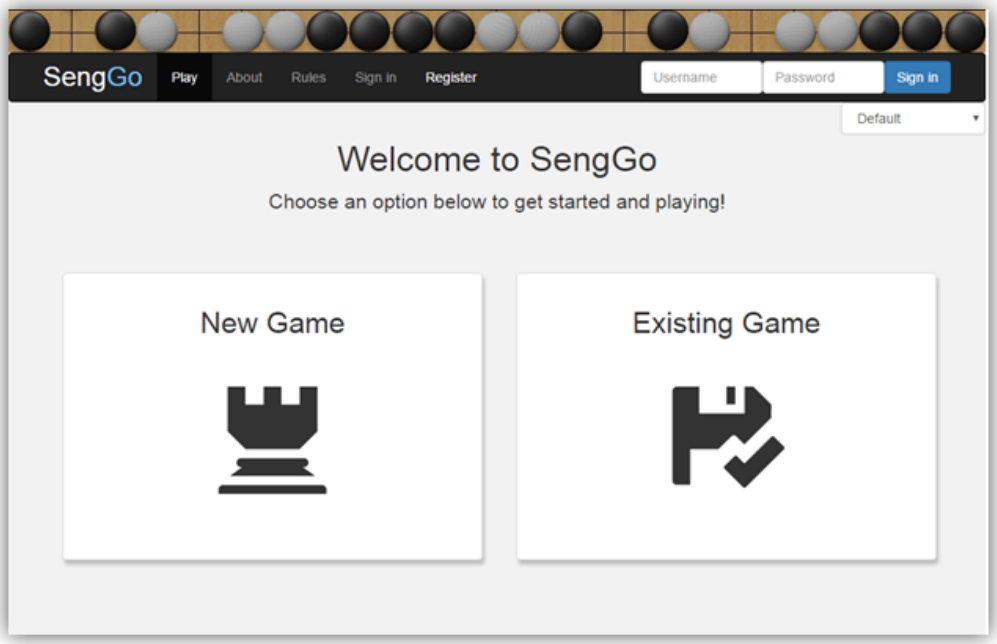

Figure A.1. Home screen of Go application

Figure A.2. New game menu choice

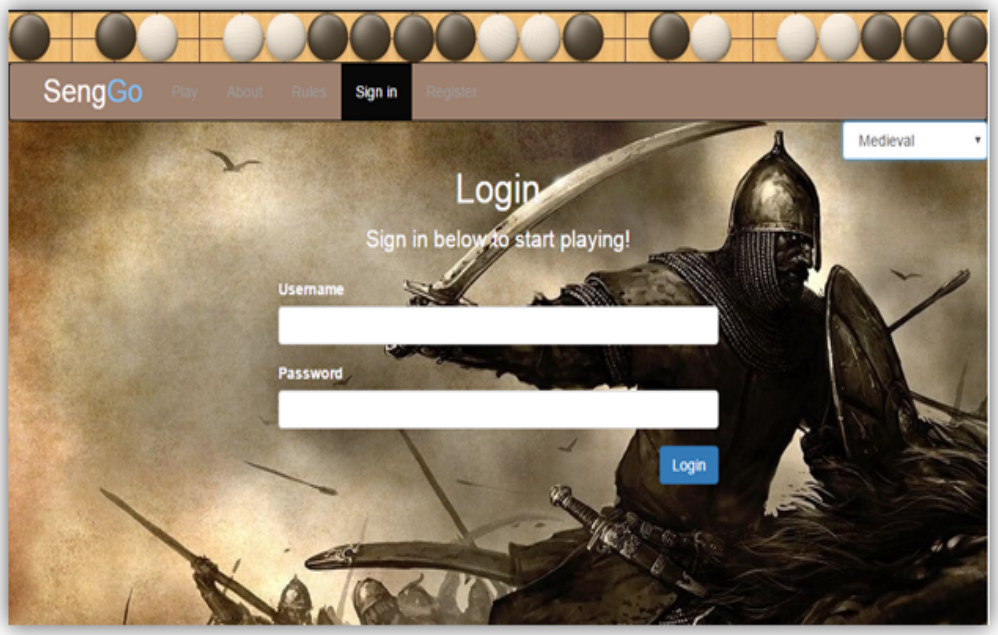

Figure A.3. Login menu

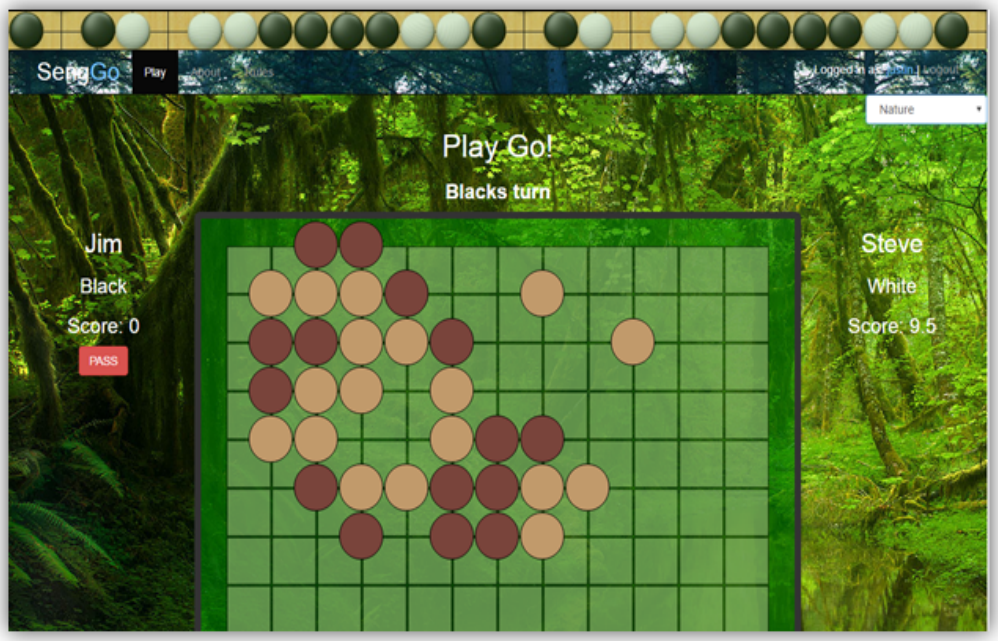

Figure A.4. Board during gameplay

Figure A.5. End of game options

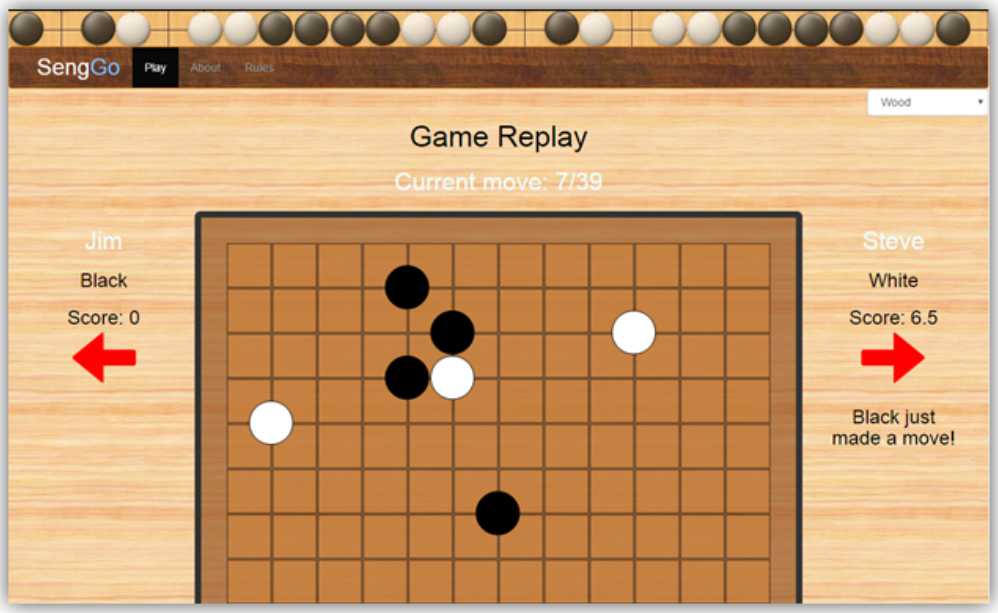

Figure A.6. Game replay

11. Appendix B

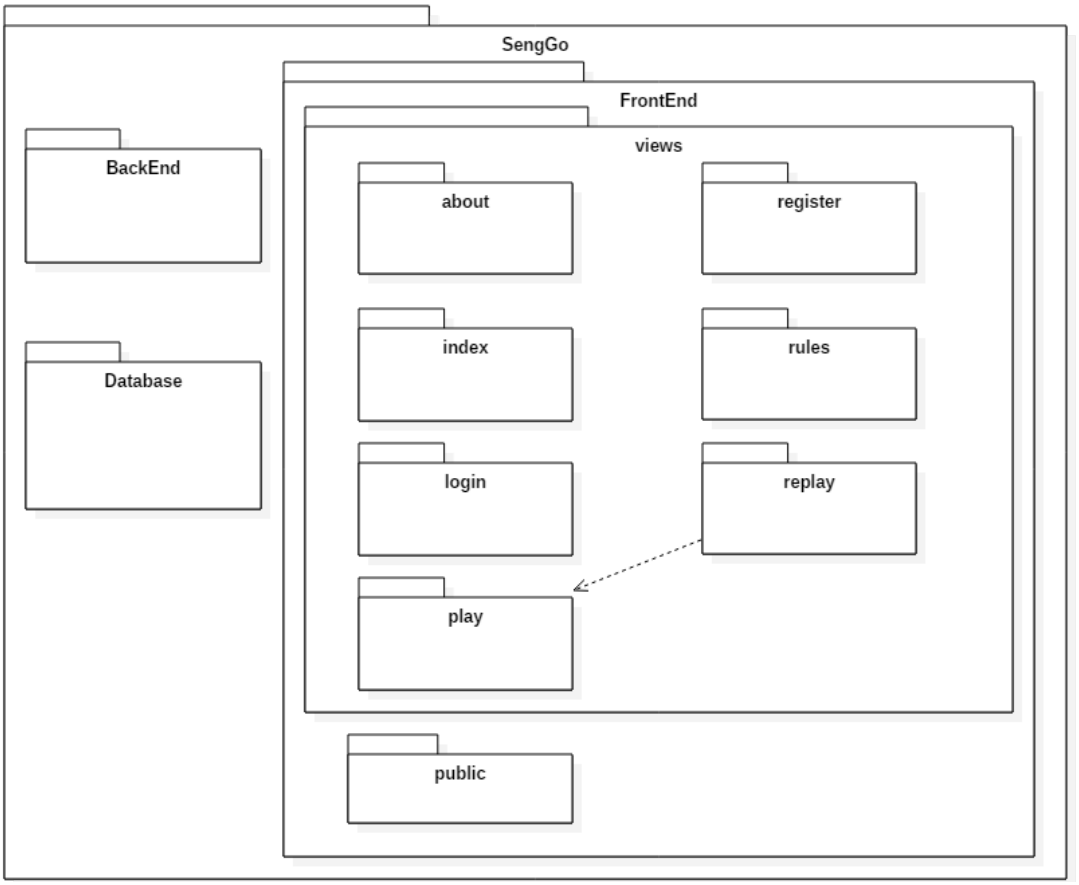

Figure B.1 Package diagram

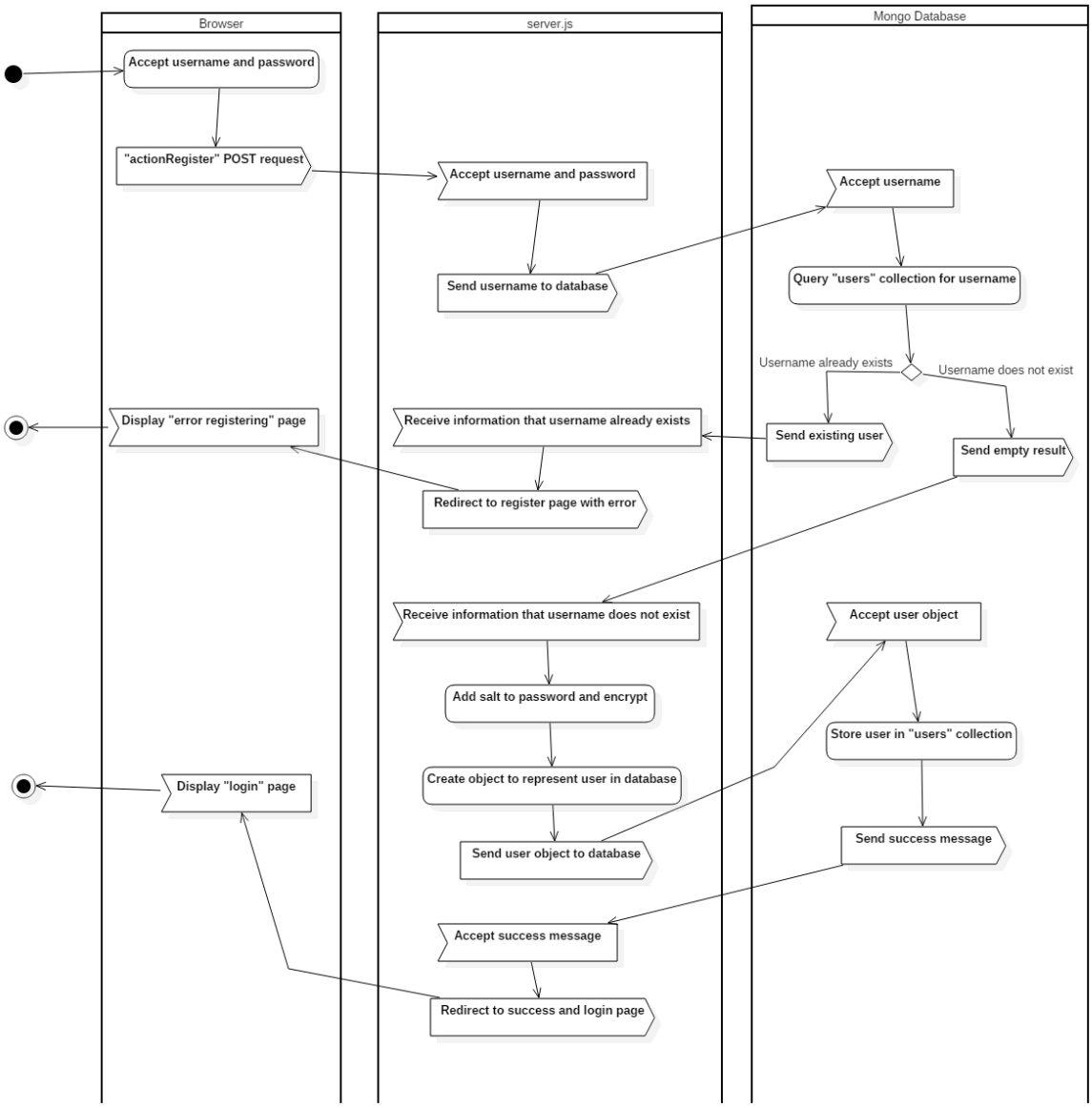

Figure B.2 Activity diagram

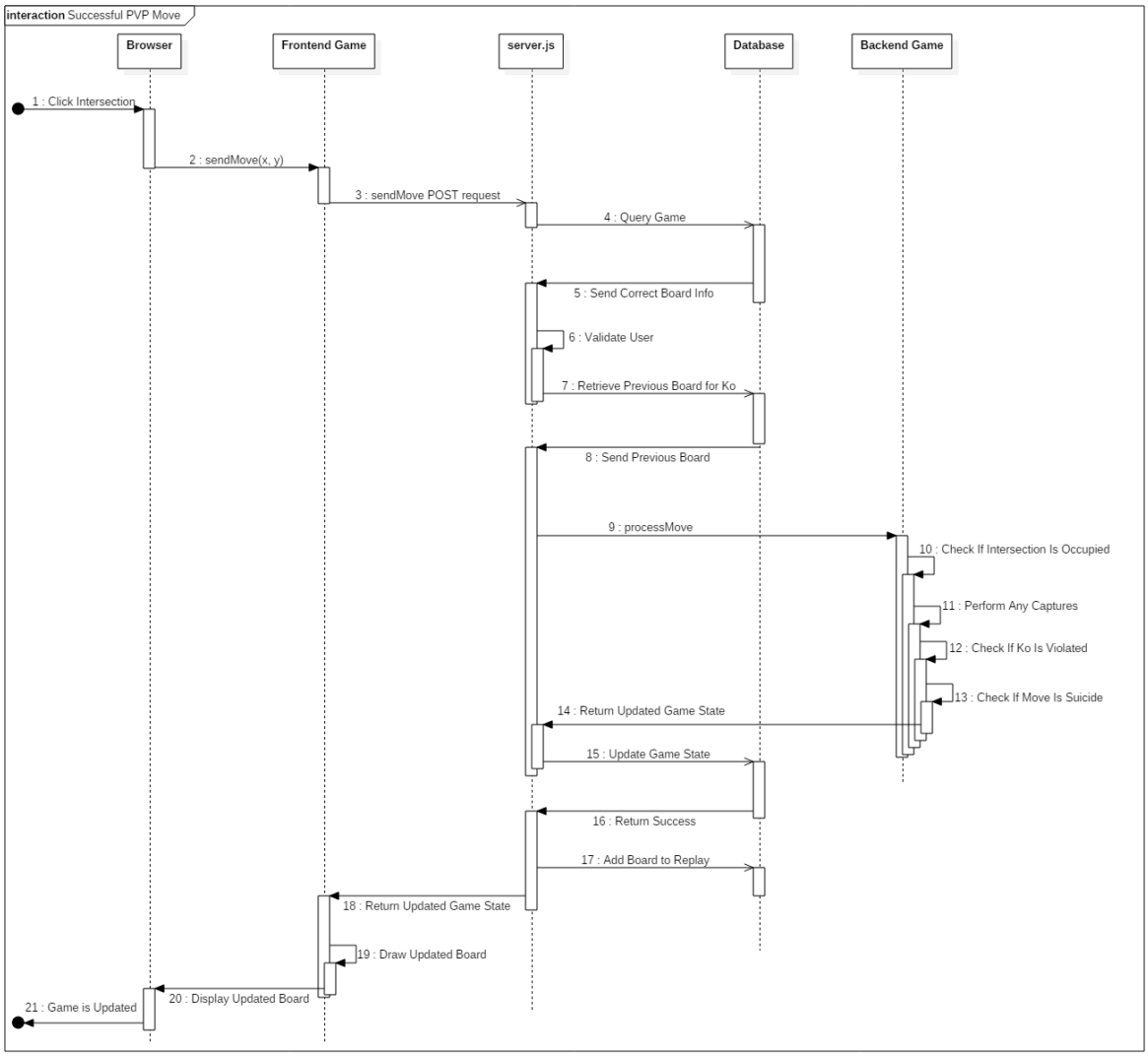

Figure B.3. Sequence diagram