V&V

Group 12

MCMASTER CAPSTONE

MONICA BAZINA-GROLINGER BAZINAGM@MCMASTER.CA
FARZAN YAZDANJOU YAZDANJF@MCMASTER.CA
ANANT PRAKASH PRAKAA1@MCMASTER.CA
NIDA NASIR NASIRN4@MCMASTER.CA
PHILIP LEE LEEP18@MCMASTER.CA

Contents

1. Controls / Versions

Date	Version	Developer(s)	Change
February 9, 2024	1.0	Entire Team	Initial V&V Doc
March 16, 2024	1.1	Farzan	Final Updates: New Route, Testing, Endpoint and Socket Mocking

2. Project Description

The purpose of our chess variant project is to create an innovative, engaging, and challenging experience by introducing numerous gameplay elements that provide a new twist on the traditional game of chess. Development of RhombiChess consists of a JavaScript front-end and a Python back-end which communicate through our defined API endpoints. The front-end consists of various react components that provide unique functionalities and come together to make the entire UI. The backend consists of various functions and classes that allow manipulation of the board state and game information.

3. Component Test Plan

3.1 Unit Tests

3.1.1 Front-End

We will be using `Jest`, an industry-standard JavaScript Testing Framework, to conduct our front-end testing. Using Jest unit tests, we will ensure that key elements on the front-end render and behave correctly in various common scenarios as well as uncommon edge cases. By using Jest, we aim to simplify the testing process, improve the robustness of our application, and implement a Test-Driven Development approach to RhombiChess. As our application communicates continuously with the backend, we must mock various endpoint calls (GET and POST) as well as socket connections (along with emitted updates).

The Board Component

Testing the board component involves various considerations. We must implement unit tests to cover the following functionalities and ensure that unexpected errors do not arise. We can do the following testing by mocking the API endpoints with custom board states and results.

- Ensuring that the board correctly displays all of the individual pieces with the default starting state, a random game state, and a game state with missing pieces.
- Ensuring that the board's orientation matches the current player's assigned colour.
- Ensuring that piece movements behave logically – this means allowing only the moves that the backend deems legal, preventing the user from stacking pieces on a tile, preventing the user from moving the opponent's pieces, allowing the user to cancel a move, and preventing the user from making moves outside of their turn.
- Ensuring that the user is always presented with a confirmation modal for each piece move and that the confirmation or cancellation works as expected (piece is moved or returned).

The Chat Component

The chat component is straightforward but needs robustness testing to ensure it behaves correctly when many messages are sent between users. The following tests will be implemented.

- Ensure that the component can handle a message history of 1000 messages and display all of them in chronological order.
- Ensure that the user can send multiple messages, of varying lengths, and see them displayed.
- Ensure that the user cannot send messages that consist only of whitespace.
- Ensure that the user cannot spam more than 2 messages per second (warning is displayed).

The Create Game Component

The Create Game component allows users to create a new game. The user has the option to select a password for the game and the option to choose their colour. We will need to ensure that the user can start a game if they provide (A) No password, random colour (default settings) and (B) Password (no white spaces), black or white colour. Furthermore, we will need to unit test

that the user is automatically redirected to the game URL once the `create_game` endpoint returns the game ID (we will mock this response).

The Join Game Component

The Join Game component allows users to join an existing game by inputting an existing game code. If the game is password protected a password field will appear. Once the user inputs the correct information they get redirected to the game. The following test will be implemented.

- Check the game code field is visible initially.
- Check that upon entering a correct game code for a game without a password, the user is directed to the game.
- For a game with a password, ensure the password field becomes visible after inputting a valid game code.
- Check that the correct game code and game password navigate the user to the game.
- Check that an incorrect game code and/or game password displays an error message.

The Sign-In Card Component

The Sign-In Card component is where the user enters their credentials to log into their account. They need to provide an email and password. The following tests will be implemented:

- Check that an existing email and password can sign in.
- Verify that an incorrect email and password display an error message.
- Check that an incorrect email or password displays an error message.

The Sign-Up Card Component

The Sign-Up Card component is where the user can sign up for an account. This involves them providing a full name, email, password, and password confirmation. The following test will be implemented.

- Check that with a valid potential email and password a user can register.
- Ensure that an invalid email or password displays an error message.
- Verify the user cannot register with missing fields (such as missing full name).

3.1.2 Back-End

For our backend, validation and verification are being done using the Python package `pytest`. The expectation for the backend is to be resilient as it will need to cater to a large user base. Since most of the actual game logic and computation is happening at the server level, the backend needs to also be able to handle large workloads. Another consequence of having most of the game mechanics in the backend is that we need to ensure that all these tasks happen in an acceptable time frame. This metric will vary for each aspect of the backend and will be discussed in more detail for each component later. The goal for these tests is to ensure that the server can handle simultaneous requests from many users and update the game state in an acceptable time frame. In addition to this, we need to verify that the game logic is correct and the game itself is behaving as expected. Below is the expected validation required for each component.

ChessBoard (ChessPiece and ChessTile):

- We need to ensure that the board is represented correctly at the beginning of a game.
- Ensure that moving a piece correctly updates the new tile, as well as frees the old tile.
- Ensure that the player cannot move the opponent's pieces.
- Ensure that the player cannot use captured pieces.
- Ensure that the board has the correct colour and orientation assigned to tiles.

Routes (Expected Cases):

- For each route, we need to ensure:
  - The expected response is correct after repeated calls.
  - If incorrect data is provided by the client, an appropriate error is returned.
- `/api/new_game`
  - Ensure that players are assigned the correct colour.
  - ensure that game_id returned is correct, and a new game is created for that id.
- `/api/game/<game_id>`
  - Ensure correct response based on request type (GET/POST)
  - [POST] Ensure that the move requested gets applied to correct game_id
  - [POST] Ensure that the board returned is correct based on requested move.
  - [POST] Ensure that the opponent cannot move any player's pieces using this request.
- `/api/game/promotion/<game_id>`
  - Ensure correct response based on request type (POST)
  - [POST] Ensure that the requested promotion is valid.
  - [POST] Ensure that the requested promotion gets applied.
- Websocket (SocketIO):
  - Ensure that the created chat room corresponds to the correct game.
  - Ensure that messages stored on the server update correctly.
  - Ensure that message records are maintained in case the user disconnects from server.

Verification of Rules:

- Individual pieces: Ensure legal moves generated for each piece are correct.
- Game-end Scenarios:
  - Ensure that player is restricted to appropriate action in the event of a check.
  - Ensure that check/checkmate scenarios are correctly identified.

Timer:

- Ensure the timer is frequently synced with the server – to ensure that the client has the correct time and to prevent cheating.
- Ensure that the client gets the correct timer in the event of a connection loss.

3.2 Performance Tests and Metrics

In addition to unit tests, we will also conduct performance tests to ensure that our application responds to user actions with minimal latency. The tests and metrics that we have decided upon are as follows.

3.2.1 Front-End

Maximum Latency between Board Updates after Moves (sent or received)

This will evaluate the latency in updating the board after moves are made or received in order to create a more responsive and consistent user experience. This will involve exploratory testing, which means that the tester will go through gameplay and ensure there are no spikes in latency or delay. This testing is done manually as we do not have the resources or see a strong need for automating such testing.

Maximum Latency when Chatting with Opponent

One of the primary requirements of our application is that it reliably supports chatting between players. To ensure that the users experience accurate communication, we will test that our front-end chat component can handle displaying 1000 messages on load within 500ms, and that any additional messages sent through the web socket are received and displayed within a maximum of 250ms. In reality, it is very unlikely that two users will send 1000 messages throughout the course of a single game - therefore, this performance test ensures a comfortable maximum.

Maximum Latency when Logging In or Signing Up

Single user login/sign-ups: simulate one user signing up and logging in under normal conditions, then measure the latency where 300 ms is the best-case scenario, a range between 300 ms and 1 ms is the latency we are aiming for and 1s is the worst-case scenario.

Multiple user logins: simulate multiple users logging in at the same time and then calculate the latency where 500 ms is the best-case scenario, a range between 500 ms and 1.5 is the latency we are aiming for and 1.5s is the worst-case scenario.

Multiple user sign-ups: simulate multiple users signing up at the same time and measure the latency after each user registers successfully where 700 ms is the best-case scenario, a range between 700 ms and 2s is the latency we are aiming for and 2s is the worst-case scenario.

- The users will be gradually increased from a smaller user group of about 10 users to a larger user group of 200 users and the effect on the latency will be evaluated.

Maximum Latency when Creating Games

Single game creation: simulate a single user creating a game at a time and then calculate the latency from when the user sends the create game request to the point when the game is successfully created where 200 ms is the best-case scenario, a range between 200 ms and 800 ms is the latency we are aiming for and 800ms is the worst-case scenario.

Multiple game creation: simulate multiple users creating new games at the same time and then calculate the average latency for creating games under this load where 300 ms is the best-case scenario, a range between 300 ms and 1.5 s is the latency we are aiming for and 1.5s is the worst-case scenario.

- The users will be gradually increased from a small user group of 5 users to a larger user group of 100 users and the effect on the latency will be evaluated.

Maximum Latency when Joining Games (password required)

Single user joins game: simulate a single user joining a game that requires a password and then calculate the latency from when the user sends the join game request (after the password is entered) to the point when the user successfully joins where 200 ms is the best-case scenario, a range between 200 ms and 800 ms is the latency we are aiming for and 800ms is the worst-case scenario.

Multiple users join multiple games: simulate multiple users joining games at the same time and then calculate the average latency for joining games under this load where 300 ms is the best-case scenario, a range between 300 ms and 1.5 s is the latency we are aiming for and 1.5s is the worst-case scenario.

- The users will be gradually increased from a small user group of 5 users to a larger user group of 100 users and the effect on the latency will be evaluated.

Maximum Latency when Confirming a Move

When a user selects a destination tile for a piece they are moving, this action triggers a pop-up that prompts them to cancel or confirm the move via buttons on the modal or the enter key (to confirm the move). This action should have minimal latency, to ensure that the following tests will be performed:

- Appearing: This test measures the duration from the trigger action (selecting a destination tile for a piece) to the appearance of the confirmation modal. This is repeated 30 times, and the average time of this action should be 200ms in its best case, the worst accepted 2s, and a target of 500ms.
- Disappearing: This test measures the duration from the trigger action (selecting the confirm button on the modal) to the disappearance of the confirmation modal. This is repeated 30 times, and the average time of this action should be 1s in its best case, the worst accepted 3s, and a target of 2ms.

3.2.2 Back-End

Handling Numerous Simultaneous Games

This metric is aimed primarily at evaluating the system's ability to handle multiple chess games happening simultaneously at some time. To be more specific, we expect the system to retain low latency across all games so that any delays after a move is made are not noticeable. The main metrics we will use to test this are load tests, where we purposefully stress out the system of up to 50 games and measure the effects of latency as it increases. Another metric we would like to look at during these tests is the total throughput of the system as well as the error rate. The unit of transaction of our throughput to be more specific will be any new move requests which need to be relayed to the opponent player and the moves they are allowed to make. The error rate will be measured by objectively counting the number of failures that occur which are not limited to disconnects, request time-outs, fatal errors etc.

Handling Various Simultaneous API Requests

This metric will be used to test how well the system will be able to process simultaneous requests at all endpoints which includes and is not only limited to the chat requests, new game requests, move requests etc. Specifically, we will measure the system's response time to ensure that all requests are processed within an acceptable timeframe, aiming for minimal delay to maintain a smooth user experience for up to 50 requests. Additionally, bottlenecks in the system will be identified, which are crucial for understanding the system's capacity under peak load conditions. Finally, resource utilization metrics, such as CPU and memory usage, will be assessed to ensure that the system remains efficient and scalable, even as the number of simultaneous requests increases.

Performance of Various Algorithms (Check, Checkmate, Possible Moves)

To assess the algorithmic performance of the checkmate/check moves and the possible moves algorithm in our online chess game, we have established a comprehensive set of performance metrics and tests. This metric will be used to evaluate the efficiency, accuracy, and speed of these algorithms under various game scenarios. Firstly, we will measure the response time of these algorithms to ensure that the time it takes for calculating matches the theoretically calculated time using big O notation. Secondly, simulated accuracy tests will be conducted to verify that every possible move, check, or checkmate condition is correctly identified without error and that any calculated moves are consistent with the rules of the game. Lastly, we will use tests of different board states to ensure scalability of the algorithms as the number of calculated pieces increases and we will set expected time metrics for different numbers of pieces. These evaluations will help us refine and optimize our algorithms, guaranteeing a smooth experience for all users.