TEST PLAN FOR
FULL STACK FLUSH

ChangeLog

Version	Change Date	By	Description
1.0.0	Feb 28, 2025	All members	Initial Tests
2.0.0	Mar 21, 2025	All members	Integration + Acceptance Tests

1 Introduction

1.1 Scope

This is our testing scope for Sprint 3:

1. Account Management
    - Create an account
    - Login to an account
    - Ensure the page renders correctly
2. Balance Management
    - Retrieving the user's balance
    - Updating the user's balance
        - Withdrawing
        - Depositing
3. Leaderboard Management
    - Retrieving user stats with filters
    - Applying filters for leaderboard
    - Additional calculation for leaderboard
    - Ensure the page renders correctly
4. Profile Management
    - Retrieve user information
    - Update user's time spent and money spent
5. Tutorial Management
    - Retrieving tutorials with ID
6. Win and Lose Management
    - Retrieve user stats
    - Update user stats
7. Blackjack
    - Play multiple hands
    - Win / Lose / Neutral Scenario
    - Safe gambling notifications
    - Balance and betting
8. Poker
    - Discarding and Playing hands
    - Re-ordering of cards
    - Safe gambling notifications
    - Balance and betting

1.2 Roles and Responsibilities

Name	Net ID	GitHub username	Role
Mateo DeSousa	desousa2	@mateod812	Full Stack Developer, QA Analyst
Kaye Ruwe Mendoza	mendozkr	@kayerm	Full Stack Developer, Test Manager
Darian Hamel	hameld1	@DarianHamel	Full Stack Developer, QA Analyst
Prashant Nigam	nigamp	@real-prash	Full Stack Developer, QA Analyst
Scott Barrett	barret17	@ScottLBarrett	Full Stack Developer, QA Analyst
Chineze Obi	OBIC3	@Chineze-prog	Full Stack Developer, Test Manager

Role Details:

1. Full Stack Developer
    - A developer who worked on both the back-end and the front-end. They integrated their own database, database controllers (linking the back-end and front-end) and designed their own UI.
2. Test Manager
    - Planned and made sure tests were functional and present.
3. QA Analyst
    - Responsible for going through merge requests for dev/main branches, and that there are no conflicts with merges. If there are, they will troubleshoot and piece it with the existing branch.

2 Test Methodology

2.1 Test Levels

Core Feature: Account Management (Signup.test.js)

Unit Testing

- Successfully creates a new user with default values. POST request to make a new user returns a 201 response status and the message response says “User signed in successfully”.
- Prevent user creation if the username already exists. POST request to make a new user with an existing username returns a 400 response status and a message response that says “User already exists”.
- Passwords are transferred in the body in hash, not plain-text. Applying the bcrypt.compare function to the plaintext password and hashed password returns TRUE.
- Returns a token after a successful signup. POST request to create an account returns a 201 response status and a “token” exists in the response message.
- Fails when required field(s) like username and/or password is not provided. POST request with missing fields returns a 404 response status and a response message that says parameters are missing.
- Handles server errors gracefully when the database fails. Returns a 500 response status and server error message but does not crash.

Integration Testing

- Checks if a new user can be created successfully when valid details are provided. A POST request is made to the /signup endpoint with a username of "testuser" and a password of "testpassword". Expects a 201 status code, a success message, a token, and the default balance to be set to 100. Additionally, it expects the returned user object to include the correct username.
- Ensures that the server responds with an error if the provided username already exists. A user with the username "testuser" is created in the database. A POST request is made to the /signup endpoint with the same username and password. Expects a 400 status code and a message indicating that the user already exists.
- Verifies that the server returns an error if either the username or password is missing. A POST request is made to /signup with only the username provided and no password. Expects a 404 status code and an appropriate message indicating that required parameters are missing.
- Ensures that the password is stored as a hash in the database. A POST request is made to /signup with valid username and password. After the user is created, the password in the database is compared against the plain text password using bcrypt.compare to verify that it is hashed correctly.
- Simulates a server error during user creation and ensures that the server handles it appropriately. “User.findOne” is mocked to simulate a database error. A POST request is made to /signup. Expects a 500 status code and a message indicating a server error. After the test, the mock is restored to its original behavior.

Acceptance Tests

1) As a new user, I want to be able to create a new account and access the whole web application.
    - I open the Full Stack Flush webpage.
    - Given I'm in the role of guest user, I click on the person icon on the top right of the screen.
    - The icon presents me with two options, "Login" or "Register".
    - Since I do not have an account, I click on the "Register" option.
    - The system shows me a Sign Up form containing a "Username" and "Password" fields, which are required.
    - I fill in the "Username" field with a unique username, and the "Password" field with a password I picked.
    - Then I click the "Submit" button.
    - The system shows a notification with the message "User signed in successfully" and directs me to the homepage, indicating a successful sign up.
    - Here I can now access any tab I want without the redirection back to the homepage because I am not registered in an app.

Core Feature: Account Management (Login.test.js)

Unit Testing

- Successfully logs in with valid credentials. POST request to login returns a 201 response status and a response message that says "User logged in successfully".
- Prevent login if one or both credentials (username and/or password) are wrong. POST request with an incorrect username returns a 404 response status and 401 for an incorrect password.
- Fail to log in and produce a message if one or both fields (username and/or password) are missing. POST request with missing fields returns a 400 response status and a response message that says "All fields are required".
- Returns a token after a successful login. POST request after successful login returns a 201 response status and a "token" exists in the response message.
- Handles server errors gracefully when the database fails. Returns a 500 response status and server error message but does not crash.

Integration Testing

- Ensures that a user can successfully log in with valid credentials. A test user is created in the database with a predefined username and password. A POST request is made to the /login endpoint with the correct username and password. Expects a 201 status code, indicating successful login. The response should include a success message, a success flag set to true, and a token.
- Checks if the server returns a 401 status when an incorrect password is provided. A test user is created with a valid password. A POST request is made with the correct username but an incorrect password. Expects a 401 status code and a message indicating that the password or username is incorrect.
- Ensures that the server returns a 404 status if the username provided does not exist. A test user is created with a known username. A POST request is made with an incorrect

username and a valid password. Expects a 404 status code and a message indicating that the password or username is incorrect.
- Checks if the server returns a 400 status code when either the username or password is missing in the request. A POST request is made with only the username and no password. Expects a 400 status code with a message indicating that all fields are required.
- Simulates a server error when the User.findOne function fails, such as when there's a database issue. The "User.findOne" method is mocked to throw an error. A POST request is made with a valid username and password. Expects a 500 status code and a message indicating a server error.

Acceptance Tests

1) As a logged-out user, I want to be able to log in to my account so that I can access my user profile and find fun gambling games to play.
    - I open the Full Stack Flush webpage.
    - Given I'm in the role of guest user, I click on the person icon on the top right of the screen.
    - The icon presents me with two options: "Login" and "Register".
    - Since I do not have an account, I click on the "Login" option.
    - The system shows me a Login form containing a "Username" and "Password" fields, which are required.
    - I fill in the "Username" field with the username I used to register, and the "Password" field with the password I picked with when I registered.
    - Then I click the "Submit" button.
    - The system directs me to the homepage, indicating a successful login.

2) As a logged-in user, I want to be able to log out of my account whenever I want to.
    - Given I am in the role of a registered user.
    - Wherever I am on the page, the Navbar with the person icon on the top right of the screen is present.
    - I click on this person icon, and it presents me with two options: "Profile" and "Logout".
    - I click on the "Logout" option. When I click on the person icon again, the two options present, "Login" and "Register", indicating a successful logout.

Core Feature: Balance Management (Balance.test.js)

Unit Testing

- Successfully returns the correct balance of an existing user. GET request returns a 200 response status and the correct balance variable of a specific user.
- Testing getting a balance of a nonexistent user. GET request returns a 404 response status if the user does not exist.
- There is a successful deposit when the password is correct (verification for deposit). Balance is updated to the correct value when the user sends their correct password with the request.
- There is an unsuccessful deposit when the password is incorrect. Returns a 400 response status when the password is incorrect, and verification for deposit fails.

- There is a successful withdrawal from a user using the correct password. Balance is updated to the correct value when the user sends their correct password with the request.
- There is an unsuccessful withdrawal from a user using an incorrect password. Returns a 400 response status when the password is incorrect, and verification for withdrawal fails.
- There is an unsuccessful withdrawal if the withdrawal is greater than the user's balance. Returns a 400 response status and withdrawal fails.
- Testing whether withdrawal or deposit works in a nonexistent user. Returns a 404 response status as the user does not exist and cannot be found so nothing to update.
- Handles server errors gracefully when the database fails. Returns a 500 response status and does not crash.
- Testing whether withdrawal or deposit works with an invalid request. Returns a 400 response status and a message that the request was invalid and missing parameters.

Integration Testing

- Seeds the database with a test user. Sends a GET request to /balance with the test user's username. Expects the response to be a 200 status code. Validates that the correct balance is returned.
- Sends a GET request to /balance with an unknown username. Expects the response to be a 404 status code. Confirms that the error message indicates the user was not found.
- Sends a POST request to /update-balance with the correct username, password, and amount. Expects the response to be a 200 status code. Confirms that the balance is updated correctly.
- Sends a POST request to /update-balance with the correct username but incorrect password. Expects the response to be a 400 status code. Checks that the error message states "Incorrect password".
- Sends a POST request to /update-balance with a withdrawal amount greater than the balance. Expects the response to be a 400 status code. Confirms that the error message states "Insufficient balance".

Acceptance Tests

1) As a logged-in user, I want to be able to put money into my account.
    - Given I am in the role of a registered user.
    - I click on the person icon on the top right corner of the screen. I click on the person icon, and it presents me with two options: "Profile" and "Logout".
    - I click on the "Profile" option and it directs me to a page with my user information.
    - Underneath my username, there are options buttons which include "Balance".
    - When I click on "Balance", it shows me information like my Current Balance. It also shows me a form of the amount I want to deposit, my card number, expiration date, confirmation number, and my password.
    - After I put all my information about that into the form, I click on the button "Confirm Deposit".
    - After I click on "Confirm Deposit", I notice the change on my "Current Balance" which indicates that I have successfully put money in my account.

Core Feature: Leaderboard Management (Leaderboard.test.js)

Unit Testing

- Ensure returned leaderboard correctly sorts users based on query parameters. Users are in the correct order in the array based on filters and GET request returns a 200 response status.
- Tests whether filters will correctly filter the leaderboard returned, and ensures users are in the correct order in their sorted array.
- Ensures the calculation of ratio accurately sets win/loss ratio. The expected win-loss ratio is returned based on a user's wins and losses variables.
- Sorting by *wins* both in ascending and descending order has the right order. Users are in the correct order in the array based on filters.
- Sorting by *moneySpent* both in ascending and descending order has the right order. Users are in the correct order in the array based on filters.
- Sorting by *timeSpent* both in ascending and descending order has the right order. Users are in the correct order in the array based on filters.
- Sorting by *username* both in ascending and descending order has the right order. Users are in the correct order in the array based on filters.
- Attempting to sort by a parameter that does not exist returns a 400 response status.
- Handles server errors gracefully when the database fails to find() something. Returns a 500 response status and does not crash.

Integration Testing

- Seeds the database with test users who have different numbers of wins. Sends a GET request to /leaderboard with sortBy=wins and order=desc. Expects the response to be a 200 status code. Validates that the users are returned in the correct order (highest wins first).
- Seeds the database with test users who have varying amounts of money spent. Sends a GET request to /leaderboard with filter=highSpenders. Expects the response to be a 200 status code. Verifies that only users who qualify as high spenders are included. Ensures that users who do not meet the criteria are not present in the response.
- Sends a GET request to /leaderboard with an invalid sortBy parameter. Expects the response to be a 400 status code. Checks that the error message indicates an invalid sorting option.
- Mocks the “User.find” function to simulate a database failure. Sends a GET request to /leaderboard with sortBy=wins and order=desc. Expects the response to be a 500 status code. Confirms that the error message states a database error occurred. Restores the original User.find function after the test.
- Seeds the database with test users who have different win/loss ratios. Sends a GET request to /leaderboard with sortBy=winLossRatio and order=desc. Expects the response to be a 200 status code. Checks that the win/loss ratio is correctly calculated and sorted in descending order. Ensures that the ratio formatting is accurate (e.g., 2.00 for an 8:4 win/loss ratio).

Acceptance Tests

1) As a logged-in user, I want to be able to view a leaderboard that compares my win/loss ratio against everyone else's so that I can see how well I am doing compared to others.

- Given I am in the role of a registered user.
- Wherever I am on a page, the Navbar with the trophy icon on the top very left of the screen is present.
- I click on the trophy icon, and it directs me to the “Leadership” page.
- On the page, I can see my username, my current ranking, my wins and losses count, my win/loss ratio, how much money I’ve spent, and how much time I’ve spent.
- On the same page, I can also see others’ information to which I can compare my own scores with other players.

2) As a logged-in user, I want to be able to filter the leaderboard based on different criteria such as win/loss ratio, time spent, money spent, etc. so that I can compare my gambling habit progress against other users and stay motivated to improve my habits.
    - Given I am in the role of a registered user.
    - Wherever I am on a page, the Navbar with the trophy icon on the top very left of the screen is present.
    - I click on the trophy icon, and it directs me to the “Leadership” page.
    - Underneath the “Leadership” title, there are drop downs for which I can select a field to filter, in what order and with which users.
    - I select a field to filter between username, wins, losses, win/loss ratio, money spent, and time spent.
    - I select an order between ascending or descending.
    - I select a group of users between all users, high spenders, and longest playtime.
    - As I select a field, order, and users, the leaderboard changes in real time and can compare my progress against others instantaneously.

Core Feature: Profile Management (Profile.test.js)

Unit Testing

- Returns the correct user information of a valid user in the response body and also returns a 200 response status from the GET request.
- Returns a 404 response status from the GET request if the user requested for stats does not exist.
- Handles server errors gracefully when the database fails to find a user. Returns a 500 response status and does not crash.
- The returned user information should not expose the password and should be stored in a hash not in plain-text in the response body of the GET request.

Integration Testing

- Checks if the correct user balance is returned. A user is created with a balance of 500. A GET request is made to the /getBalance endpoint with the username testuser. Expects a 200 status code and a response body containing the correct balance (500).
- Checks if the user's time spent is updated correctly. A POST request is made to /setTimeSpent with a new time spent value of 60. Expects a 200 status code and a success response. The user’s timeSpent value in the database is updated to 60.

- Checks if the daily time and money spent limits are reset correctly. A POST request is made to /resetDailyLimits. Expects a 200 status code and a success response. The user's daily time spent and daily money spent are reset to 0.
- Checks if the user's money limit is updated correctly. A POST request is made to /setMoneyLimit with a new money limit value of 2000. Expects a 200 status code and a success response. The user's moneyLimit is updated to 2000 in the database.
- Ensures that the server rejects invalid money limits (e.g., negative values). A POST request is made to /setMoneyLimit with a negative money limit value of -10. Expects a 400 status code and a failure response indicating an invalid value.
- Checks if the user's limits (money and time) are returned correctly. A GET request is made to /getLimits with the username testuser. Expects a 200 status code and the correct money limit (1000) and time limit (120).
- Checks if the user's statistics (wins and losses) are returned correctly. A GET request is made to /getStats with the username testuser. Expects a 200 status code and the correct statistics (10 wins and 5 losses).
- Ensures that the last login date is returned correctly. A GET request is made to /getLastLogin with the username testuser. Expects a 200 status code and the correct last login date to be returned.
- Ensures that a 404 error is returned for a non-existing user. A GET request is made to /userInfo with a non-existing username unknown. Expects a 404 status code and a message indicating that the user was not found.
- Checks if the server rejects negative time values when updating the time spent. A POST request is made to /setTimeSpent with a negative time spent value of -10. Expects a 400 status code and a failure response indicating that negative values are not allowed.

Acceptance Tests

1) As a logged-in user, I want to be able to change my basic information like my limits.
    - Given I am in the role of a registered user.
    - I click on the person icon on the top right corner of the screen. I click on the person icon, and it presents me with two options: "Profile" and "Logout".
    - I click on the "Profile" option and it directs me to a page with my user information.
    - Underneath my username, there are options buttons which include "My Profile".
    - When I click on "My Profile", it shows me information like my username, password as hashed, time limit and money limit.
    - When I direct to the "Edit" button on the time limit section, it presents me with a number box where I can change the number. After I modify the number, and click the "Save" button.
    - After I click the "Cancel" button, it directs me back to the "Edit" section where I see that my "Time Limit" is now changed, indicating that I can change my information.

2) As a logged-in user, I want to be able to view my win-to-loss ratio so that I can see how well my performance is.
    - Given I am in the role of a registered user.
    - I click on the person icon on the top right corner of the screen. I click on the person icon, and it presents me with two options: "Profile" and "Logout".

- I click on the “Profile” option and it directs me to a page with my user information.
- Underneath my username, there are options buttons which include “Stats”.
- When I click on “Stats”, it shows me information like my money spent, time spent, wins, losses, and win/loss ratio.
- The table about my stats indicates how good my performance is in my past few games and habits.

3) As a logged-in user, I want to be able to view my transaction history so that I can keep track of how much money I am gambling and which days I visited the platform.
    - Given I am in the role of a registered user.
    - I click on the person icon on the top right corner of the screen. I click on the person icon, and it presents me with two options: “Profile” and “Logout”.
    - I click on the “Profile” option and it directs me to a page with my user information.
    - Underneath my username, there are options buttons which include “History”.
    - When I click on “History”, it shows me a history from the most recent activity about deposits and cash back I receive from games.
    - I confirm that the activities listed are from the most recent to oldest.

Core Feature: Win/Lose Management (WinLose.test.js)

Unit Testing

- GET request for a user’s wins value returns a 200 response status and the correct amount of wins associated with their account.
- Returns a 404 response status if the user is not found when trying to GET their wins.
- Handles server errors gracefully when the database fails to find a user. Returns a 500 response status and does not crash.
- GET request for a user’s losses values returns a 200 response status and the correct amount of losses associated with their account.
- POST request for updating wins and losses returns a 200 response status and the expected updated values for wins and losses are correct.
- GET request returns 400 response status if the username is not found and returns a message stating “User not found”.
- Updating wins/losses with negative values returns a 400 response status and a response message that states “Invalid values”.
- Only increments or updates the wins/losses values if only one is provided. If only wins are provided in the body, add that to the wins total, if only losses are provided in the body, add that to losses total. Returns a 200 response status.

Integration Testing

- Checks if the server correctly returns the number of wins for a valid user. A user is created with wins set to 10. A GET request is made to /getWins with the query parameter username=testuser. Expects a 200 status code and the response body to contain the correct number of wins (wins: 10).

- Ensures the server returns a 404 status code if the user does not exist. A GET request is made to /getWins for a non-existent user (username=unknownuser). Expects a 404 status code and a message indicating that the user is not found.
- Simulates a database error while retrieving the wins and ensures the server handles it appropriately. “User.findOne” is mocked to simulate a database error. A GET request is made to /getWins for a valid user. Expects a 500 status code and a message indicating a server error.
- Checks if the server correctly returns the number of losses for a valid user. A user is created with losses set to 5. A GET request is made to /getLosses with the query parameter username=testuser. Expects a 200 status code and the response body to contain the correct number of losses (losses: 5).
- Ensures the server returns a 404 status code if the user does not exist when querying losses. A GET request is made to /getLosses for a non-existent user (username=unknownuser). The test expects a 404 status code and a message indicating that the user is not found.
- Simulates a database error while retrieving the losses and ensures the server handles it appropriately. “User.findOne” is mocked to simulate a database error. A GET request is made to /getLosses for a valid user. Expects a 500 status code and a message indicating a server error.
- Checks if the server correctly updates a user's wins and losses. A user is created with wins set to 10 and losses set to 5. A POST request is made to /updateStats to update wins to 12 and losses to 8. Expects a 200 status code and the updated stats in the response body.
- Ensures the server returns a 404 status code if the user is not found during stats update. A POST request is made to /updateStats with a non-existent user (username=unknownuser). Expects a 404 status code and a message indicating that the user is not found.
- Ensures the server returns a 400 status code if the wins or losses values are negative. A user is created with wins set to 10 and losses set to 5. A POST request is made to /updateStats to update wins to -2 and losses to 3. Expects a 400 status code and a message indicating invalid values.
- Ensures the server returns a 400 status code if the username is missing in the request body. A POST request is made to /updateStats with missing username but valid wins and losses values. Expects a 400 status code and a message indicating that the username is required.
- Simulates a database error while retrieving the user during stats update and ensures the server handles it. “User.findOne” is mocked to simulate a database error. A POST request is made to /updateStats to update the stats of a valid user. Expects a 500 status code and a message indicating a server error.
- Simulates a database error during the save operation and ensures the server handles it appropriately. A user is created with wins set to 10 and losses set to 5. “User.save” is mocked to simulate a database error. A POST request is made to /updateStats to update the stats of a valid user. Expects a 500 status code and a message indicating a server error.

Core Feature: Tutorial Management (Tutorial.test.js)

Unit Testing

- Retrieving the tutorial by its ID returns a 200 response status from the GET request and returns the expected tutorial.
- Returns a 404 response status when an invalid tutorial ID is provided in the GET request.
- GET request for tutorials without ID returns all the tutorials that exist in the database and returns a 200 response status.
- Handles server errors gracefully when the database fails to find a tutorial. Returns a 500 response status and does not crash.
- Returns a message to say the id parameter is missing if it is missing in the GET request.

Integration Testing

- Checks if the server correctly returns all tutorials. Two tutorials are created with different titles, content, video URLs, and difficulty levels. A GET request is made to /api/tutorials to retrieve all tutorials. Expects a 200 status code, a success: true flag, and the response body to contain exactly two tutorials with their correct _id values.
- Ensures the server correctly returns a single tutorial when given a valid ID. A tutorial is created with a specific title, content, video URL, and difficulty. A GET request is made to /api/tutorials?id=<tutorial_id> to fetch that specific tutorial. Expects a 200 status code, a success: true flag, and the correct tutorial _id in the response.
- Ensures the server returns a 404 status code when requesting a non-existent tutorial by ID. A new, randomly generated ObjectId is used that does not correspond to any tutorial in the database. A GET request is made to /api/tutorials?id=<non_existent_id>. Expects a 404 status code, a success: false flag, and a message "Tutorial not found".
- Ensures the server handles a database error when fetching tutorials. The find method of the Tutorials model is mocked to simulate a database failure. A GET request is made to /api/tutorials. Expects a 500 status code, a success: false flag, and a message "Server error".

Acceptance Tests

1) As a user that is new to gambling, I want to view a list of available games with tutorials so that I can choose the one I want to learn.
    - Given I am in the role of a registered user.
    - Wherever I am on the page, the Navbar with the book icon on the top left of the screen is present.
    - I click on the book icon, and it redirects me to a “Game Tutorial” page.
    - On the page, I am presented with two buttons which are the two game tutorial options: “Blackjack” and “Poker Minigame”.
    - When I click on the “Blackjack” button, there is a paragraph that explains the concept of the game Blackjack to me.
        - There is a button that says “Watch Video”.
        - When I click on the “Watch Video” button, it opens a new tab explaining Blackjack.
    - I press the back button to return to the “Game Tutorial” page.
    - When I click on the “Poker Minigame” button, there is a paragraph that explains the concept of the game Poker to me.
        - There is a button that says “Watch Video”.

    - When I click on the “Watch Video” button, it opens a new tab explaining the type of Poker hands.

Core Feature: Blackjack (Blackjack.test.js)

Unit Testing

- Expecting Card variables to contain what they were assigned during construction.
- Expecting the deck to contain 52 unique cards when it is created. Each card should be one of the four suits within the 13 ranks.
- Expecting the calculation for the player’s hand to work correctly. Taking soft aces into account.
- Expecting a game to correctly add players to the game and queue. When a player first joins they are put into the game. When the game is started, new players are placed into the queue.
- Expecting a game to correctly remove players from the game or queue.
- Expecting the game to tell players when the game has started.
- Expecting the game to correctly deal out cards. Each player gets two and the dealer only tells the players about one of their cards. Each player also receives a message about the other player’s dealt cards.
- Expecting the game to handle the case where the dealer has a natural 21 and end early.
- Expecting the game to calculate each player’s result correctly and tell them.
- Expecting the game to tell a player when they get 21 after a HIT call.
- Expecting the game to tell a player when they bust after a HIT call.
- Expecting the game to tell all players a new card when dealt after a HIT call.
- Expecting the game to correctly go through the dealer’s turn and tell players each new card the dealer receives.
- Expecting the game to move onto the next player when a player calls STAND.
- Expecting the game to restart when the players tell the server they want to play again.
- Expecting the socket connection to close when the server kicks a player for any reason.
- Expecting the server to keep track of and start new games correctly.
- Expecting the server to create new games when all current games are full.
- Expecting the server to tell each game which players to remove correctly.
- Expecting the server to tell each game what messages they receive correctly.
- Expecting the server to handle bad messages gracefully

Acceptance Tests

1) As a logged-in user, I want to receive notifications if I need to input more money into my account before I place my bet.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Blackjack” button, and it redirects me to the Blackjack page.
    - On the Blackjack page, I see my current balance, a customizable bet amount, a Start Game button, and a Start Free Game option.
    - I see my current balance is $313.
    - I decide to pick a bet amount $400 which is over my current balance.

- After inputting a $400 bet, I press the Start Game button since I want my game to actually play around with money.
- When I click Start Game, the game does not start and a notification pops up that says “Invalid bet amount” indicating I cannot start the game if I bet more than the balance I currently have.

2) As a logged-in user, I want to have the ability to stand, double down, hit, or split my blackjack hand so that I can make the best decision for my particular hand.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Blackjack” button, and it redirects me to the Blackjack page.
    - On the Blackjack page, I see my current balance, a customizable bet amount, a Start Game button, and a Start Free Game option.
    - I click on the Start Free Game option.
    - After I click on the Start Free Game option, I am immediately presented with the dealer’s hand and my hand.
    - On the side there are two options: “Hit” and “Stand”.

3) As a logged-in user, I want to be able to see the amount of money or tokens that I bet for my current hand so that I can understand the risk of my current bet.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Blackjack” button, and it redirects me to the Blackjack page.
    - On the Blackjack page, I see my current balance, a customizable bet amount, a Start Game button, and a Start Free Game option.
    - I see my current balance is $313.
    - I decide to pick a bet amount of $10 which is less than my current balance.
    - After inputting a $10 bet, I press the Start Game button.
    - When I click Start Game, the game starts where I am presented with a Blackjack game.
    - On the top of the game screen, there is a label called “Bet Amount” and beside it is the bet input I filled before the game started indicating the risk of my current bet of the game.

Future Core Feature: Poker (Poker.test.js)

Unit Testing

- Expecting to see “Coming Soon”, “♠Poker Game ♥”, “We’re working hard to bring you an immersive poker experience” and “Back to Home” on the Poker page.

Integration Testing

- Checks if the server correctly initializes a new poker game with the specified difficulty. A POST request is made to /poker/start with the body { difficulty: "easy" }. Expects a 200 status code and the response body to contain the gameID, playerHand, handsRemaining (set to 4), discardsRemaining (set to 3), and gameOver (set to false).

- Ensures the server correctly draws a specified number of cards from the deck. A GET request is made to /poker/draw with the query parameters gameID (from the previous test) and count=2. Expects a 200 status code and the response body to contain the newCards property with an array of 2 cards.
- Verifies that the server correctly scores a poker hand. A POST request is made to /poker/score with the body { gameID, selectedCards: [{ rank: "Ace", suit: "Spades" }] }. Expects a 200 status code and the response body to contain the score and currentScore properties.
- Checks if the server correctly sorts a poker hand by rank. A POST request is made to /poker/sort-hand with the body containing a hand of cards and the criteria "rank". Expects a 200 status code and the response body to contain the sortedHand property. The sorted hand should have the cards in ascending order of rank (e.g., "Ace" first and "King" last).

Acceptance Tests

1) As a logged-in user, I want to be able to sort my poker hand by suit or rank so that I can better visualize which poker hand will be best for me to play.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Poker” button, and it redirects me to the Poker Minigame page.
    - When I get the Poker Minigame, I am presented with a dropdown of difficulty, an input for the bet amount, and “Start Game” and “Back to Home” buttons.
    - I select a difficult option between Easy, Medium and Hard.
    - I select a bet amount that is lower than my current balance.
    - I press the “Start Game” button to play the game with my chosen options.
    - It takes me to a Poker Minigame page where I am presented with the cards, and four buttons at the bottom: “Play Hand”, “Sort by Rank”, “Sort by Suit” and “Discard”.
    - When I press the “Sort by Rank” button, I notice the cards order change from ranking by suit, to sorted the cards by rank.
    - When I press the “Sort by Suit” button, I notice the cards order change back from sorted by rank, to sorted by suit.

2) As a logged-in user, I want to be able to select 1-5 cards to play as my poker hand so that I have the freedom to play more or less cards for the current hand.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Poker” button, and it redirects me to the Poker Minigame page.
    - When I get the Poker Minigame, I am presented with a dropdown of difficulty, an input for the bet amount, and “Start Game” and “Back to Home” buttons.
    - I select a difficult option between Easy, Medium and Hard.
    - I select a bet amount that is lower than my current balance.
    - I press the “Start Game” button to play the game with my chosen options.
    - It takes me to a Poker Minigame page where I am presented with the cards, and four buttons at the bottom: “Play Hand”, “Sort by Rank”, “Sort by Suit” and “Discard”.

- I click on four different cards on the card lineup on the screen.
- When I press the “Play Hand” button at the bottom, there is a notification “Two Pair” and my “Score: 10”.

3) As a logged-in user, I want to be able to discard 1-5 cards so that I can search for more cards in the deck and play a better poker hand.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Poker” button, and it redirects me to the Poker Minigame page.
    - When I get the Poker Minigame, I am presented with a dropdown of difficulty, an input for the bet amount, and “Start Game” and “Back to Home” buttons.
    - I select a difficult option between Easy, Medium and Hard.
    - I select a bet amount that is lower than my current balance.
    - I press the “Start Game” button to play the game with my chosen options.
    - It takes me to a Poker Minigame page where I am presented with the cards, and four buttons at the bottom: “Play Hand”, “Sort by Rank”, “Sort by Suit” and “Discard”.
    - I click on four different cards on the card lineup on the screen.
    - When I press the “Discard” button at the bottom, the four cards I selected are replaced by four different cards I have not played before.

2.2 Frontend Tests

Core Feature: Account Management (Signup.test.js)

Unit Testing

- Expecting to see “Username:”, “Register Now!”, “Enter your username”, “Enter your password”, “Submit” and “Close” if the boolean *show* is true.
- Expecting to see “Register Now!” if the boolean *show* is false.

Integration Testing

- Checking if the signup model displays when the “show” prop is True. Expecting to see “Username:”, “Register Now!”, “Enter your username”, “Enter your password”, “Submit” and “Close” if the boolean *show* is true.
- Ensure the “Register Now!” text appears only if the “show” is trueand that the modal renders nothing or a relevant message if the “show” prop is false. Expecting the "Register Now!" text to not be visible when show is false.
- Verify the username and password input fields update correctly when text is entered. Expecting the input fields to reflect the changes made during user interaction.
- Simulate a successful signup by submitting the form with valid data. Expecting the component to display a success message indicating the signup was successful.
- Simulate a failed signup attempt (.e.g. if the username is already taken). Expecting the component to show an error message indicating the signup failed.
- Verify that the modal closes when the close button is clicked. Expecting the close button to trigger the “onClose” function, and the modal should be closed.

- When the login button is clicked, the signup modal should switch to the login form. Expecting the “setShowLogin” function to be called with “True”, and the modal should close.

Core Feature: Account Management (Login.test.js)

Unit Testing

- Expecting not to see the component when the boolean *show* is false.

Integration Testing

- Ensure the login modal renders correctly when the show is true. Expecting the modal to display "Login to your account" and that the username and password input fields are present.
- Ensure typing in the input fields updates their values. Expecting the username and password fields to reflect the entered text.
- Ensure a successful login navigates the user to / and reloads the page. Expecting that the toast notification confirms login success, the modal closes after login, navigation to / occurs, and the page reloads.
- Ensure an incorrect login attempt triggers an error. Expecting that the toast notification shows "Login failed" and no navigation or page reload occurs
- Ensure clicking "Close" hides the login modal. Expecting the mockOnClose function to be called.
- Ensure clicking "Register" switches to the signup form. Expecting the mockSetShowSignup(true) to be called and that mockOnClose() is called to close the login modal.

Core Feature: Leaderboard Management (Leaderboard.test.js)

Unit Testing

- Expecting to see “Username”, “Wins”, “Losses”, “Win/Loss Ratio”, “Money Spent” and “Time Spent” on the dropdown menu.
- Expecting to see “No data available” when the leaderboard is rendered as empty.

Integration Testing

- Ensure the sorting dropdown and its options (Username, Wins, Losses, Win/Loss Ratio, Money Spent, Time Spent) are correctly displayed. Expecting that all these options are present in the dropdown when the component is rendered.
- Verify that the Leaderboard component successfully makes an API call to fetch leaderboard data.
- Ensure that user data (e.g., username, wins, losses, win/loss ratio, money spent, and time spent) is correctly rendered when the API returns data. Expecting that if the API call succeeds and returns data, the leaderboard should display user information.
- Simulate a scenario where the API returns an empty array. Ensures that the component displays "No data available" when there is no leaderboard data.
- Simulate an API failure (e.g., server error with status 500). Expecting an appropriate error message is displayed to the user and the UI does not break and is still be navigable.

Future Core Feature: Safe Gambling (GamblingReminders.test.js)

Unit Testing

- Expecting to see a toastify notification with the gambling reminder.
- Expecting to see two different reminders at the different intervals.

Integration Testing

- Not Applicable

Acceptance Testing

1) As a logged-in user, I want to receive warnings, reminders, and useful facts about the dangers of online gambling so that I can be aware and educated about the issues surrounding gambling addictions, and also so that I can be sure I am gambling responsibly.
    - Given I am in the role of a registered user.
    - While I am idle on any page for 10 seconds, a notification pops up about tips on how to gamble responsibly.

2) As a logged-in user, I want to receive notifications when I have met my time or money limit for the day so that I can stick to my limits.
    - Given I am in the role of a registered user.
    - When I am on the homepage, I am presented with buttons like “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us”.
    - I click on the “Blackjack” button, and it redirects me to the Blackjack page.
    - On the Blackjack page, I see my current balance, a customizable bet amount, my daily limits and how far I am, a Start Game button, and a Start Free Game option.
    - The daily limits contain a bar of how long I have been playing and how much I have spent.
    - I have reached my money-spent daily limit therefore the bar is red. When I attempt to “Start Game” which uses my real balance, it redirects me to the homepage.
    - While I am on the homepage, I get a notification with the message “You have reached your daily limit and are locked out from playing” indicating I can not go back to my game since it will just redirect me again and again.

Future Core Feature: Safe Gambling (Resources.test.js)

Unit Testing

- Expecting to see “Gambling Resources” on the Resources page.

Integration Testing

- Ensure all gambling resource sections appear on the page. Expecting to see "Gambling Resources", "Before You Play", "Responsible Gambling Tips", and "Help and Support"
- Ensure the external resource links are displayed with the correct URLs. Expectant the component to display and correctly link to “https://www.responsiblegambling.org”, “https://afm.mb.ca/programs-and-services/gambling/” and “https://www.camh.ca/en/health-info/mental-illness-and-addiction-index/problem-gambling”.

- Verify clicking the "Back to Home" button navigates to the homepage. Expecting the "Back to Home" button to have an href attribute pointing to /, ensuring navigation.

Non-Core Feature: About Us! (AboutUs.test.js)

Unit Testing

- Expecting to see “About Full Stack Flush” and “Meet Our Team” on the About Us page.

Integration Testing

- Not Applicable

Non-Core Feature: Home (Home.test.js)

Unit Testing

- Expecting to see “Blackjack”, “Poker”, “Tutorials”, “Leaderboard”, “Resources” and “About Us” on the Home page.

Integration Testing

- Verifies that the signup modal is displayed when the show prop is set to true. Expecting to see the "Username:" label, "Register Now!" text, "Enter your username" input field, "Enter your password" input field, "Submit" button, and the "Close" button.
- Ensures that "Register Now!" is not visible when show is false. Verifies that the modal renders either nothing or an appropriate message when show is false.
- Verifies that the input fields update correctly when text is entered. Expecting the input fields to reflect the changes made during user interaction.
- Simulates form submission with valid data. Expecting the component to display a success message indicating that the signup was successful.
- Simulates an attempt to sign up with an already taken username or invalid data. Expecting the component to display an error message indicating that the signup failed.
- Verifies that clicking the close button triggers the onClose function. Ensures that the modal is closed after the button is clicked.
- Ensures the Home component renders with the navigation links: "Blackjack", "About Us", "Resources", "Tutorials", "Poker", and "Leaderboard".
- Verifies that clicking on navigation links correctly routes to the expected pages: "Blackjack" navigates to the Blackjack page, "About Us" navigates to the About Us page and displays "About Full Stack Flush", "Resources" navigates to the Resources page and displays "Gambling Resources", "Tutorials" navigates to the Tutorials page and displays "Game Tutorials", "Poker" navigates to the Poker page and displays "♠ Poker Minigame ♥" and "Leaderboard" navigates to the Leaderboard page and displays "Leaderboard".

2.3 Test Completeness

Testing will be considered complete when:

- We have 80% back-end code coverage for our test cases.
- All test cases execute and pass successfully.

3 Resource & Environment Needs

3.1 Testing Tools

General Tools/Methods:

- GitHub Actions for CI/CD

Front-end Tools:

- Chrome Developer Tools
- React Testing Library
- Jest

Back-end Tools:

- React Testing Library
- Jest
- Supertest
- MongoDB Memory Server

3.2 Test Environment

The minimum **system** requirements that will be used to test the Application.

- *Operating System*: Windows 10 and above, MacOS, and Linux
- *Web browsers*: Google Chrome, Firefox

The minimum **software** requirements that will be used to test the Application.

- NodeJS v22.12.1 or above (https://nodejs.org/en)
- npm v10.9.2 or above (terminal)
- Latest install of Virtual Studio (https://code.visualstudio.com/)

4 Terms/Acronyms

TERM/ACRONYM	DEFINITION
QA	Quality Assurance
UI	User Interface
CI/CD	Continuous Integration/Continuous Deployment