TEST PLAN FOR FULL STACK FLUSH

ChangeLog

Version	Change Date	By	Description
1.0.0	Feb 28, 2025	All members	Initial Tests

1 Introduction

1.1 Scope

This is our testing scope for Sprint 2:

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

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.

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.

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.

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.

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.

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.

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.

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.

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

Unit Testing

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

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.

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

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.

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

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

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.

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.

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

Unit Testing

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

2.2 Test Completeness

Testing will be considered complete when:

- We have 100% 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
- 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