Team Name: Game Fame
UNIs: gc2767, sa3522, arz2116, sl3948
GitHub Repo: https://github.com/alekzieba/game-fame

Game Fame Second Iteration

Test Plan

Since we are managing application state with Redux, the unit testing for our application is performed by testing that the actions dispatched by the action creators are as they should be. The testing for this application was done with Jest and the files that our test suite runs are located within the "test/actions" directory of our project. The equivalence partitions and boundary conditions for each group of action creators are as follows:

Authentication:

- Signing In:
  - Equivalence Partitions: Emails that are currently registered as users within our database; emails that are not currently registered as users within our database.
  - Boundary Conditions: None since emails are strings and this operation is simply a lookup.
  - NOTE: this set of tests is the only one that is particularly difficult to test since it is dependent upon the user interacting with a separate application (their web browser) to sign into our application. (Users sign in with Google.) We have made progress towards writing complete tests for this (by simulating a user), but since they do not fully work at the moment, we have not committed these changed.
- Signing Out:
  - Equivalence Partitions: Application states where the users are signed in (valid); Application states where the users are not signed in (also valid). Note that if the users are not signed in, performing the sign out action is equivalent to doing nothing and should succeed.
  - Boundary Conditions: None since the equivalence partitions span all possible application states.

Connect Four

- Retrieve Board from Database
  - Equivalence Partitions: Keys that have a game associated with them (valid); Keys that do not have games associated with them (invalid).
  - Boundary Conditions: None since game keys are strings and this operation is simply a lookup.
- Making a Move
  - Equivalence Partitions: Board states where a move for the chosen column is permitted (valid); Board states where a move for the chosen column is not permitted (invalid).

    - Boundary Conditions: Board states that are empty; Board states where the column clicked is one move away from being full; Board states where the column clicked is full.
- Creating a Board
    - Equivalence Partitions: The set of all application states (should make a new board no matter what).
    - Boundary Conditions: None since this should always succeed.

Game Info

- Retrieving Game Info
    - Equivalence Partitions: Emails that are currently registered as users within our database; emails that are not currently registered as users within our database.
    - Boundary Conditions: None since emails are strings and this operation is simply a lookup.

Tic Tac Toe

- Retrieve Board from Database
    - Equivalence Partitions: Keys that have a game associated with them (valid); Keys that do not have games associated with them (invalid).
    - Boundary Conditions: None since game keys are strings and this operation is simply a lookup.
- Creating a Board
    - Equivalence Partitions: The set of all game keys (valid).
    - Boundary Conditions: None since this should always succeed.
- Resetting a Board
    - Equivalence Partitions: The set of all game keys (valid).
    - Boundary Conditions: None since this should always succeed.

Branch Coverage

While researching coverage tools, we learned that a coverage tool called Istanbul is built into Jest, so we activated it on our test suite. The results are listed below:

```
----------------------|----------|----------|----------|----------|-------------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------------|----------|----------|----------|----------|-------------------|
All files             |    68.75 |    49.15 |    62.16 |    69.23 |                   |
 app/actions          |    68.15 |    49.09 |    61.11 |    68.66 |                   |
  ConnectFour.js      |       84 |    53.06 |    70.59 |    85.14 |... 93,212,216,217 |
  auth.js             |    19.23 |        0 |       25 |    19.23 |... 48,52,53,55,59 |
  games.js            |       75 |      100 |       50 |       75 |             14,15 |
  tictactoe.js        |    69.23 |       50 |    85.71 |    69.23 |... 17,18,22,26,27 |
 app/constants        |      100 |      100 |      100 |      100 |                   |
  firebase.js         |      100 |      100 |      100 |      100 |                   |
 internals/scripts    |    71.43 |       50 |      100 |    71.43 |                   |
  CheckBuiltsExist.js |    71.43 |       50 |      100 |    71.43 |             19,27 |
```

Team Name: Game Fame
UNIs: gc2767, sa3522, arz2116, sl3948
GitHub Repo: https://github.com/alekzieba/game-fame

```
----------------------|----------|----------|----------|----------|-------------------|
```

From these results, we can observe that the unit tests ("app/actions" section) have some work to do in terms of branch coverage, but this is most likely due to the technical difficulty in testing asynhronous Firebase code (e.g. making sure that updates are detected within the database, which raises timing issues) – this is something that we will be working on within the next iteration. We did not write any test cases for loops because our tested actions do not rely on loops (this is due to an asynchronous design pattern recommended by Firebase). Additionally, it is worth noting that the abysmally low coverage rates for "auth.js" are due to the issue mentioned above about the action being dependent on the results from an external application (due to OAuth).

Functionality Changes

We will be adding three key features to our product for the second iteration demo.

(1) Messages within games: We have a chat between the players of the game which stores all logs (even conversations from prior game) and brings them into the latest one. We accomplish this by adding a new category to our Firebase backend which stores user1&user2, where user1 and user2 are the respective players, under the "conversations" child node. For the next iteration, we will be adding functionality for users to be able to delete their messages from the chat (which will delete them from the other user's end as well).

(2) Locking players after move: In order to have a fair game, we need to ensure that the user who just made a move cannot play for the other user. We accomplish this by having an extra parameter in our backend game logic which keeps track of the userID of the last player who made a move. Once that is stored, we check the backend before letting the next player make a move, ensuring that only the alternate player is allowed to play. For the next iteration, we plan on providing functionality for an "undo move" if both players agree to it.

(3) Improving appearance of boards: The tic-tac-toe board was too small and the connect-four board colors were a bit too bright. Changing the appearance to make the games look better is very important if we want people to keep using our product!