Northeastern

Project 1: Shambomon

Matt Dang, Kimberly Nguyen

March 27, 2018

Table of Contents

I. Introduction and Game Description

Shambomon is a two-player game that is a combination of Roshambo, commonly known as "Rock, Paper, Scissors," and Pokémon[1]. The gameplay involves players battling against one another via multiple rounds of Roshambo, with the winner of each round being the player who chooses the attack that trumps that of the opposing player, as described in Roshambo's game rules[2]. Each player starts the game with 100 health points (HP), and after each round, the losing player's HP is deducted by a certain value. The deducted value is dependent on the presence of a multiplier, which is applied when a certain player achieves two or three consecutive wins. Otherwise, the base deducted value is 10.

Along with the three regular rock, paper, and scissors attacks, there is a special attack that each player can activate once per game. This attack deals 40 damage if successful. The success of the attack is determined by a "die roll" simulation on the server. The attack is successful if the die lands on a six; no damage is dealt if it lands on any other number. The special attack trumps all other attacks, except if the other player also successfully activates her special attack in the same round.

When first visiting the homepage of the game, users must log in to an existing account or create a new one. Once logged in, users will be prompted for a "game name." This name determines the game room in which they will be placed (i.e., to join the same game, both users must enter the same game name). Depending on whether or not they are one of the first two users to join the room, they will either play or spectate the game. If they are playing the game, they will be brought to a page where they can select a Pokémon to represent themselves. There are twelve Pokémon to choose from, and each Pokémon has its own character selection icon and battlefield avatar. Meanwhile, spectators will be able to watch the game in real time but cannot interact with it in any way. Once the game is complete, the stats for each user will be recorded in

---

[1] "Pokémon." *Wikipedia*, Wikimedia Foundation, 24 Mar. 2018, en.wikipedia.org/wiki/Pok%C3%A9mon.
[2] "Rock–Paper–Scissors." Wikipedia, Wikimedia Foundation, 20 Mar. 2018, en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors.

a leaderboard, where users can view their past matches and compare their win ratios to that of other users.

II. UI Design

We aimed for a user-centered design, focusing on minimalism to achieve a straightforward, easy-to-use interface. The minimalist design also reflects the simpleness of the two classic games that inspired Shambomon, and the fonts that we chose are reminiscent of the old 8-bit Pokémon games.

On the homepage, users are immediately prompted with a log-in form. If they do not have an account yet, they can register via the link in the top-right of the form. Once logged in, users will see their username in the top-right, followed by a "`LOG OUT`" link. If users try to visit the homepage again after logging in, they will conveniently be redirected back to the game name page. As such, the "`HELP`" and "`LEADERBOARD`" links from the homepage are also present on the game name page in case users need a refresher on how to play or if they want to check their stats. The game name page is another form, and it redirects users to the character selection page upon submission.

The Pokémon icons that we chose are from DeviantArt, designed by artist JedFlah. We created two rows of icons, with the name of the corresponding Pokémon above each icon. We thought this looked nicer than having, say, a dropdown of the Pokémon names. In addition, as a visual aid, the opacity of the icons change when users hover over them.

If the game does not have two players yet, users will see a "`Waiting for another player...`" message. Once a second player joins, the battlefield is rendered. The design of the battlefield is similar to that of the Pokémon games in that the user's character is rendered at the bottom, the names and avatars of each Pokémon are displayed, and there is a health bar that displays the HP of each player. We made the health bars change from green to yellow to red as the HP drops so that players are aware of their statuses. The attack buttons are images of their corresponding attack--rock, paper, or scissors--and are displayed in this familiar order. A star image is used for the special attack. On the left-hand side is a message box displaying the past attacks and damage taken for each round in descending order, with the most recent round information being a different color for greater clarity. Spectators do not see the attack buttons

since the attacks do not apply to them. In the middle of the screen, there is a message that displays "`YOUR TURN`," "`OPPONENT'S TURN`," or "`YOU ARE SPECTATING`" so that players always know the current state of the game. Lastly, there is a "`HELP`" link in the top-right that users can use to review the game rules. Once the game is over, players will see a message indicating whether or not they have won, and there will be links to either start a new game or view the leaderboard.

We used the auto-generated `User` index page as our leaderboard page since we wanted to list the stats for each user, and a table made the most sense. Beside each row in the table is a "`Match History`" link, where users can view the past matches for that user. For the match history page, we used the auto-generated `User` show page. Victories are highlighted in blue, while defeats are highlighted in pink. Using the auto-generated pages allowed us to not have to create unnecessary new pages, and we styled the pages to match our other pages.

III. UI to Server Protocol

When users first visit the homepage, they are required to either log in to an existing account or register a new one. When users submit the registration form, a `POST` request is sent to the `User` controller, which calls `create_user()` on the inputted values. The server creates a hash for the password if it is long enough and if the password confirmation matches and verifies that the username is unique (case insensitive). If there are no errors, then a `User` record is created. When users submit the log-in form, a `POST` request is sent to the `Session` controller. The server validates the username and password combination and saves the user in "`assigns`" upon success.

We used one global agent with a map and put/get operations to back up our game. When a user joins the channel by clicking on an icon from the character selection page, the agent either loads a saved game or creates a new one. The name of the game is saved in "`socket.assigns`" so that it can be referenced later whenever we need to load or save the game state. The server sets the joined user as either a player or a spectator in the game state. Then, it sends a message with the key "`:after_join`" and the updated game state as the value to itself and everyone else on the channel. When this message is received, the server sends a broadcast to the channel to refresh the new game state.

On the battlefield, players take turns choosing one of four attack buttons. Once an attack button is clicked, an "`attack`" request gets pushed to the channel. The message payload contains a key "`attack`" with the value being "`Rock`," "`Paper`," or "`Scissor`" and a key "`special`" with the value being a boolean indicating whether or not the special attack has been activated. The server handles the attack information, sends a broadcast to the channel to refresh the new game state, and updates the client view. The client side has a listener on the channel for the updated game state broadcasts and sets the game state upon receiving one.

When the game is over, a "`history`" request gets pushed to the channel. The message payload contains information about the match (who won, which Pokémon were chosen, etc.). To prevent duplicate records from being created for a single game, only the client side of the winner triggers this message. The server calls `create_match()` from the `Gameplay` module to create

a `Match` record with the values from the message payload. A "`stats`" request is also pushed to the channel. The message payload contains a key "`id`," where the value is the user's ID. The server calls `update_stats()` from the `Accounts` module on the user ID. The server does not send anything to the client side in response since these changes appear on the leaderboard page, not the battlefield page.

The end-game page has links for players to either start a new game or view the leaderboard. When either link is clicked, a "`reset`" message gets pushed to the channel. The message payload contains a key "`id`," where the value is the user's ID. The server removes the ID from the list of players in the game state, sends a broadcast to the channel to refresh the new game state, and updates the client view. The client side sets the new game state when it receives an "`:ok`" response from the server.

IV. Data Structures on Server

We created three resources: `User`, `Match`, and `Session`. A `User` record gets created when the user submits the registration form. If the username is unique, the password is more than seven characters long, and the password confirmation matches the password, then the server adds the user to the database after hashing the password. A `User` also has a "`wins`" count and a "`losses`" count, which get updated after every game and is displayed on the leaderboard page.

A `Match` record also gets created after every game. A `Match` has a "`player_id`" corresponding to the winning player and an "`opponent_id`," both of which belong to `User`, and string fields to keep track of the Pokémon chosen for that match. The `Match` records for each user is displayed on the leaderboard page as well. If a user is deleted, all of the matches for that user get deleted as well.

A `Session` get created when the user logs in. The server validates the username and password combination upon submission of the log-in form and saves the user in "`assigns`" upon success. This allows us to access the current user on every page. A `Session` gets deleted when the "`LOG OUT`" link is clicked.

This game state is maintained as a map that contains the keys: `turn`, `attacks`, `players`, `lastLosses`, `spectators`, `messages`, and `gameOver`. The "`turn`" value is used to track the current player's turn and is initialized to 0. The "`attacks`" value is used to count the number of attacks registered for the current round and is initialized to 0. The "`players`" value is a list of two maps that contain player information (`id`, `char`, `health`, and `attack`). The two maps are initialized to have the "`id`" be `nil`, the "`char`" be an empty string, the "`health`" be 100, and the "`attack`" be an empty string. The "`lastLosses`" value is a map used to track the losing player of the previous two rounds in order to calculate the attack multiplier. It is initialized to have the two values be `nil`. The "`spectators`" value is a list of the IDs of all of the spectators of the game and is initialized to be an empty list. The "`messages`" value is a list of the messages (strings) detailing the attacks chosen and damage taken for each round and is

initialized to be empty list. Finally, the "`gameOver`" value is a flag that is set when the current game has concluded and is initialized to be `false`.

We use one global agent to keep the game state persistent. The agent has a `start_link()`, a `save()`, and a `load()` function. `start_link()` is used by the supervisor to start the agent, which gets initialized with an empty map. `save()` backs up the state of the game by adding it to the agent's map. `load()` searches the agent's map for the game state with the given name and returns it if it exists.

V. Implementation of Game Rules

During the gameplay, each player takes turns choosing an attack. The first player to join the game channel is the first to be able to choose her attack. After she chooses her attack, the second player chooses hers, and the turns switch back and forth from there. During a player's turn, she can click one of four attack buttons. Once chosen, the attack buttons are disabled, and the turn is switched to the other player. When the attack button is clicked, a message is pushed to the channel with the attack value ("Rock," "Paper," or "Scissor") and a flag indicating whether or not the special attack was activated.

Once both players have clicked their attacks, the server determines the winner for the round by comparing the attacks to see which trumps the other, according to the "Rock-paper-scissors[3]" game rules. If the special attack was activated, the server generates a random number between one and six (inclusive) to determine whether or not the attack is successful. Once a winner for the round is determined, the server calls another function to calculate the damage dealt to the loser. The base damage is 10, and if there are any applicable multipliers, they will be factored accordingly. If the special attack is successful, and the other user did not land a successful special attack in the same round, then the damage is set to 40.

The game keeps track of the previous two rounds' losing player and uses that to calculate the multiplier. If a player has won two consecutive rounds, a multiplier of 1.5x is applied to the base damage. If a player has won three consecutive rounds, the multiplier is 2x the base damage. Once a player achieves the 2x multiplier, the object tracking the previous two rounds is reset to clear any multipliers. A tie can also break the multiplier streak.

Turns continue switching off between the two players, and the server keeps calculating damage values until one player's HP drops to 0 or less. When a player's HP reaches 0, the server responds by ending the game and rendering an end-game page. Since damages are dealt after both players have chosen an attack value for the round, only one player can reach 0 HP and lose the game.

---

[3] "Rock–Paper–Scissors." Wikipedia, Wikimedia Foundation, 20 Mar. 2018, en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors.

VI. Challenges and Solutions

One challenge we faced involved the game state not being synchronized for each player. When the first player joined a game, she had to wait for a second player to join in order to begin the gameplay. However, when the second player joined, the first player's client side was not able to detect this, and the players had to manually refresh the page in order to receive the updated game state containing the other player's information. This issue also occurred during the gameplay. After a player chose an attack, the other player had to refresh the page in order to know that it was her turn. We resolved this by attaching a listener to the channel variable in the JavaScript on the client side. This listener receives a new game state and triggers a "set state" call to render that state for all clients. On the server, a new game state is sent as a broadcast to everyone on the channel after a player joins the game and after each turn, triggering their render functions on the client side.

Another challenge we faced was deploying our application. We modeled our nginx configuration after the ones we used for our memory games since we were using websockets again and followed the steps we did to deploy our task trackers since we had to set up a production database. However, we ran into a new error, which resulted in a "502 Bad Gateway." The new error was a result of using the Argon2 password hashing algorithm, which we had not used in previous applications. Eventually, after a lot of research, we learned that there was an additional command we needed to run to properly build the application with the Argon2 dependency and were able to successfully deploy Shambomon.