Sistemes i Tecnologies Web (Web Systems and Technologies)

Stardew Fishing: Part 1 - Backend

Contents

1 Introduction

In this exercise, we will develop a backend application that will be used by a Vue frontend, implemented on the second part of the project.

The frontend will display a modified version of the fishing functionality from the 2016 game *Stardew Valley*. The player will be able to cast the a fishing line into the water, wait for a fish to take the bait, and play a mini-game to attempt to catch it. The player will have various attempts to catch fish. Once all attempts are exhausted, they will be able to press a *RETRY* button to restart the game.

Figure 1 illustrates the four main states of the player: prepared to cast the line, waiting for a fish to take the bait, playing the mini-game, and, after all attempts have been used, pressing the *RETRY* button to start over.

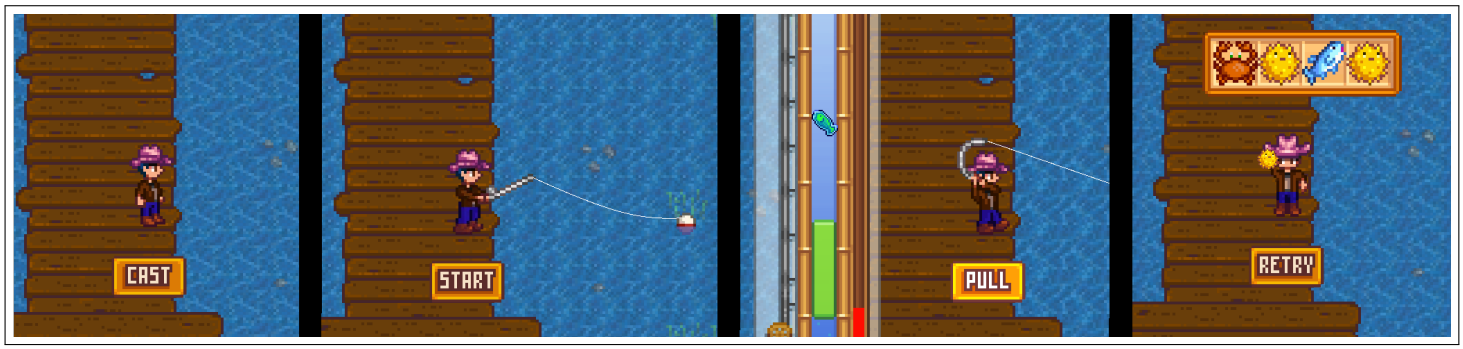

Figure 1: The four main states of the player

The backend must track the player's state at all times (standing, waiting for a fish to take the bait, playing the mini-game, etc.), monitor the number of capture attempts, and manage fish movement, the catch bar movement, and the progress bar movement during the mini-game.

To function properly, the backend must implement the closures described in Section 3 and provide the endpoints outlined in Section 4.

2 Project structure

The project will include the following files and folders:

- **package.json**: This file lists the npm [1] dependencies required by the backend server. We will need the *express* [2] package to start an *Express.js* server, the *cors* package to handle requests from the Vue frontend, the *ws* package to start a WebSocket server, and the *jest* [3] package for running unit tests. Before starting the server or running tests, install these dependencies using **`npm install`** (or **`npm i`** for short).
- **app.js**: This file initializes the *Express* server on port `8081`. It must implement the endpoints described in Section 4. To start the server, execute this file with Node.js using **`node app.js`** [4] (or **`nodemon app.js`** for automatic refresh).
- **game.js**: This file contains the **`Game`** function, which manages the game's state. It is initialized by **`app.js`**, and some of its exposed functions are called by the endpoints. More details in Section 3.5.
- **catching_minigame.js**: This file contains the **`CatchingMinigame`** function, which manages the lifecycle of the fish capture mini-game. More details available in Section 3.4.
- **fish.js**: This file contains the **`Fish`** function, which manages the movement of the fish in the mini-game. More details available in Section 3.2.
- **catch_bar.js**: This file contains the **`CatchBar`** function, which manages the movement of the catch bar in the mini-game. More details in Section 3.1.
- **progress_bar.js**: This file contains the function **`ProgressBar`**, which manages the movement of the progress bar in the mini-game. More details in Section 3.3.

- **public/globals.js**: This file contains helper variables and functions used throughout the code. You must **not** edit or move this file. It will be used by both the backend and the frontend.
- **tests**: This folder contains unit test files. Each file corresponds to a test case. More details in Section 6.
- **public**: This folder contains a compiled version of the Vue frontend application,which will be implemented in the second part of the project. Once the server is running, you can access the front-end at `http://localhost:8081`.

3 Closures

Before implementing the endpoints, the following functions must be implemented: **`Game`**, **`CatchingMinigame`**, **`Fish`**, **`CatchBar`** and **`ProgressBar`**. Figure 2 illustrates which parts of the mini-game these functions manage.

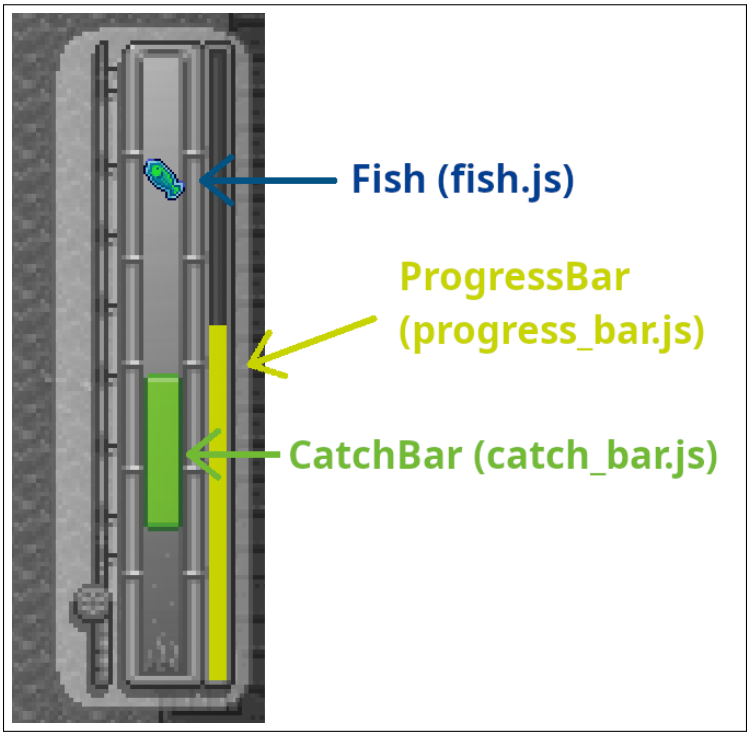

Figure 2: Mini-game parts

All of these functions use closures. Each function contains local variables modified by the exposed functions in the returned dictionary. Even after execution, these local variables persist due to JavaScript's lexical scoping.

Here is a basic example of a function using closures:

```
export default function FunctionWithClosure()
{
    let localVariable = 0;

    const exposedFunctionThatUpdatesLocalVariable = () => {
        localVariable += 1;
    }

    // If we do not return the function here, it'll not be exposed
    return {
        exposedFunctionThatUpdatesLocalVariable
    };
}
```

Using **`export default`** makes the function accessible to other JavaScript files. It can be imported as follows:

```
import FunctionWithClosure from "./function_with_closure.js"
```

Below, we describe the variables and methods to implement for each function.

3.1 CatchBar

In file **catch_bar.js**, you must implement function **CatchBar**. It will manage the movement of the catch bar inside the mini-game. We present below its parameters, local variables, and methods to expose. All local variables are declared as **undefined** (without assigning it anything), and later set its initial values on method **start**.

3.1.1 Parameters

- **swappedDirectionCallback**: Callback function that must be called every time the catch bar changes direction.

3.1.2 Local variables

- **lastSwapAt** (**number**): Timestamp of the last direction change.
  - Initial value: The current timestamp (**Date.now()**).
- **lastSwapPosition** (**number**): Position where the catch bar last changed direction.
  - Initial value: **CATCH_BAR_INITIAL_POSITION** from file **globals.js**
- **direction** (**string**): Current movement direction.
  - Domain: **"up"** or **"down"**
  - Initial value: **"down"**

The real-time position of the catch bar is not stored—only its direction, last swap timestamp, and last swap position are recorded.

3.1.3 Methods to expose

- **start**: Initializes local variables and calls **swappedDirectionCallback** (given we have updated the direction of the catch bar).
- **updateDirection**: Updates **direction**, **lastSwapAt**, and **lastSwapPosition** based on the received parameter **newDirection**. To update **lastSwapPosition**, we must use function **computeCatchBarCurrentPosition**, present in **globals.js** file.
- **getInfo**: Return a dictionary containing **direction**, **lastSwapAt** and **lastSwapPosition**.

3.2 Fish

In file **fish.js**, you must implement the function **Fish**. It will manage the movement of the fish inside the mini-game. We present below its parameters, local variables, and methods to expose. All local variables are declared as **undefined** (without assigning it anything), and later set its initial values on method **start**.

3.2.1 Parameters

- **speed**: Movement speed of the fish. It will be used by **computeFishCurrentPosition** to calculate the current position of the fish.
- **swappedDirectionCallback**: A callback function that must be invoked each time the fish changes direction.

3.2.2 Local variables

- **`lastSwapPosition`** (**`number`**): The position of the fish when it last changed direction.
  - Initial value: Random number between `0.0` and **`FISH_MAX_POS`** (defined in **`globals.js`**)
- **`lastSwapAt`** (**`number`**): The timestamp of the last direction change.
  - Initial value: The current timestamp.
- **`direction`** (**`string`**): The current movement direction of the fish.
  - Domain: `"up"` or `"down"`
  - Initial value: `"down"`

Similar to **`CatchBar`**, the real-time position of the fish is not stored—only its direction, last swap timestamp, and last swap position are tracked.

3.2.3 Methods to expose

- **`start`**: Called when the mini-game begins. Steps it must perform:
  - Set the initial values to the local variables.
  - Call **`swappedDirectionCallback`** (given we have updated the direction of the fish).
  - Start a timer to swap the direction of the fish after a certain period, determined by the function **`computeFishTimeToNextSwap`** from **`globals.js`**. Once the timer completes, it executes:
    * Updates **`lastSwapPosition`** with its current position. Given that time has passed since the last time the direction was swapped, we must calculate the position where the fish is right now. It is not necessary the calculate it yourself, you only need to use the function **`computeFishCurrentPosition`**, present in **`globals.js`** file.
    * Updates **`lastSwapAt`** with the current timestamp.
    * Swaps the fish's direction.
    * Start a timer to swap the direction of the fish again after a random duration (Using the time provided by **`computeFishTimeToNextSwap`**).

    This process involves recursion to ensure that the fish continuously and randomly changes direction.
- **`getInfo`**: Called whenever information about the fish's position is needed. It returns a dictionary containing: **`direction`**, **`lastSwapAt`** and **`lastSwapPosition`**.
- **`finish`**: Called when the mini-game ends, it stops the recursive behavior that continuously swaps the fish's direction.

3.3 ProgressBar

In file **`progress_bar.js`**, you must implement the function **`ProgressBar`**. It will manage the movement of the progress bar inside the mini-game. We present below its parameters, local variables, and methods to expose. All local variables are declared as **`undefined`** (without assigning it anything), and later set its initial values on method **`start`**.

3.3.1 Parameters

- **`fishSpeed`**. Movement speed of the fish in the mini-game. It will be used by **`computeFishCurrentPosition`** to determine the fish's current position.
- **`finishCallback`**: A callback function that is triggered when the progress bar reaches either its upper or lower limit.

3.3.2 Local variables

- **`lastSwapPosition`** (**`number`**): The position of the progress bar when it last changed direction.
  - Initial value: **`PROGRESS_BAR_INITIAL_POSITION`** from file **`globals.js`**
- **`lastSwapAt`** (**`number`**): The timestamp of the last direction change.
  - Initial value: The current timestamp.
- **`direction`** (**`string`**): The current movement direction of the progress bar.
  - Domain: **`"up"`** or **`"down"`**
  - Initial value: **`"down"`**
- **`state`**: epresents the current state of the progress bar, indicating whether it is still running or has reached a final outcome.
  - Domain: **`"in_progress"`**, **`"successful"`** or **`"failed"`**
  - Initial value: **`"in_progress"`**

Figure 3: Two possible outcomes: a successfully captured fish on the left and a failed capture attempt on the right.

3.3.3 Methods to expose

- **`start`**: Called when the mini-game begins. Steps it must perform:
  - Set the initial values to the local variables.
  - Start a timer **`t1`** to determine when the progress bar will reach either its upper or lower limit. The function **`timeForProgressBarToReachLimit`** from **`globals.js`** provides the remaining time (in milliseconds). When the timer finishes, we must:
    - Update the **`state`** variable:
      - If the direction was **`"up"`**, it means the progress bar has reached its upper limit, and hence you have correctly captured the fish. In that case, we set **`state`** to **`"successful"`**.
      - If the direction was **`"down"`**, it means the progress bar has reached its lower limit, and hence the fish has escaped. In that case, we set **`state`** to **`"failed"`**.

      Figure 3 illustrates how this two possible outcomes will be shown in the frontend.
    - Stop the timer **`t2`** (described below), as there is no need to check for interactions between the catch bar and the fish anymore.
    - Call **`finishCallback`**.

- Start a timer **t2** that executes every **PROGRESS_BAR_TICK_FREQUENCY** milliconds (from **globals.js**). This timer checks whether the catch bar and the fish are touching to determine if the progress bar should change direction. The function:
  - Computes the current position of the catch bar using function **computeCatchBarCurrentPosition** from **globals.js**.
  - Computes the current position of the fish using function **computeFishCurrentPosition** from **globals.js**.
  - Uses **catchBarAndFishTouch** from **globals.js** to check if the fish and the catch bar are touching.
  - If the progress bar is moving down and the fish and catch bar are touching—or if the progress bar is moving up and they are *not* touching—the progress bar swaps direction. When this happens:
    - Update local variable **lastSwapPosition** with the current position of the progress bar (using function **computeProgressBarCurrentPosition** from **globals.js**).
    - **direction** is swapped.
    - **lastSwapAt** is updated with the current timestamp.
    - Reset **t1**, given **lastSwapPosition** has changed, and **timeForProgressBarToReachLimit** will now return a new time value indicating when the progress bar will reach a limit.
  - Timer **t2** restarts, meaning this function will execute every **PROGRESS_BAR_TICK_FREQUENCY** milliseconds. This requires a recursive approach or the use of **setInterval**.

- **fishSwappedDirection**: Called from function Fish every time the fish swaps direction. This function has three parameters: **fishDirection**, **fishLastSwapAt** and **fishLastSwapPosition**. We must store this values in local variables, given the function that checks if the fish and the catch bar are touching, will need to store these three values.

- **catchBarSwappedDirection**: Called from function CatchBar every time the catch bar swaps direction. This function has three parameters: **catchBarDirection**, **catchBarLastSwapAt** and **catchBarLastSwapPosition**. We must store this values in local variables, given the function that checks if the fish and the catch bar are touching, will need to store these three values.

- **getInfo**: Called every time we want information about the position of the progress bar. It must simply return a dictionary containing local variables **direction**, **lastSwapAt**, **lastSwapPosition** and **state**.

3.4 CatchingMinigame

In file **catching_minigame.js**, you must implement the function **CatchingMinigame**. It will act as a middleware between **Game** and the functions related to the mini-game, which are: **ProgressBar**, **CatchBar** and **Fish**. We present below its parameters, local variables, and methods to expose. All local variables are declared as **undefined** (without assigning it anything), and later set its initial values on method **start**.

3.4.1 Parameters

- **finishCallback**: A callback function that must be executed when the catching mini-game ends.

3.4.2 Local variables

- **progressBar** (**Object**): Dictionary of functions to interact with the progress bar.
  - Initial value: The dictionary returned by calling **ProgressBar**. The callback function passed as a parameter to **ProgressBar** will be executed when the progress bar has reached one of its limits, signaling the end of the mini-game. When this happens, the function must:
    - Call the **finish** method of local variable **fish** presented below (to stop its internal timer that randomly swaps direction).
    - Call **finishCallback**.

- **catchBar** (**Object**): Dictionary of functions to interact with the catch bar.

  - Initial value: The dictionary returned by calling **`CatchBar`**. The callback function passed as a parameter to **`CatchBar`** is triggered whenever the catch bar swaps direction. If this occurs, we must call the **`catchBarSwappedDirection`** function from **`progressBar`**.

- **`fish`** (**`Object`**): Dictionary of functions to interact with the fish.
  - Initial value: The dictionary returned by calling **`Fish`**. The callback function passed as a parameter to **`Fish`** is triggered whenever the fish swaps direction. If this occurs, we must call the **`fishSwappedDirection`** function from **`progressBar`**.

3.4.3 Methods to expose

- **`start`**: Called when the mini-game begins. It receives parameter **`difficulty`**, corresponding to the difficulty of the mini-game. This value is used to determine the fish's movement speed. Steps that **`start`** function must perform:
  - Set the initial values to the local variables. Using **`DIFFICULTY_TO_FISH_SPEED`** from **`globals.js`**, we must obtain the fish's speed, an pass it as an argument to both **`progressBar`** and **`fish`**.
  - Calls the **`start`** method on the **`progressBar`**, **`catchBar`**, and **`fish`** objects to initialize them.
- **`getInfo`**: Called every time we want information about the position of the progress bar, the catch bar, and the fish. It must return a dictionary containing:
  - **`progressBarInfo`**: The result of calling the **`getInfo`** function on **`progressBar`**.
  - **`catchBarInfo`**: The result of calling the **`getInfo`** function on **`catchBar`**.
  - **`fishInfo`**: The result of calling the **`getInfo`** function on **`fish`**.
- **`updateCatchBarDirection`**: Called when the catch bar's direction needs to be updated. It invokes the **`updateDirection`** function on **`catchBar`**.

3.5 Game

In file **`game.js`**, you must implement the function **`Game`**. It will act as a middleware between the endpoints in **`app.js`** and the game, managing the state of the player and the mini-game. We present below its local variables and methods to expose.

3.5.1 Local variables

- **`playerState`** (**`String`**): States that the player can be on.
  - Domain: **`"standing"`**, **`"line_cast"`**, **`"fish_bit"`** or **`"playing_minigame"`**
  - Initial value: **`"standing"`**
  - Explanation of the states:
    - **`"standing"`**: It means the player has not cast the line to the water yet.
    - **`"line_cast"`**: It means that the player has cast the line to the water, and is waiting for a fish to take the bait. The fish will bite after exactly **`FISH_BIT_TIMEOUT_MS`** milliseconds (defined in **`globals.js`**). If we click the *START* button before the fish takes the bait, the player will simply reel in the line and return to its initial state.
    - **`"fish_bit"`**: It means that the fish has taken the bait. In this state, we have **`PULL_ROD_TIMEOUT_MS`** (**`globals.js`**) milliconds to press the *START* button in the frontend. If we don't, the fish will escape and we will not be able to start the mini-game (and the player will reel in the line, going back to its initial state).
    - **`"playing_minigame`**: It means we have clicked the *START* button on time and that the mini-game has started. After the mini-game ends (and either the fish escapes, or we are able to capture it), the player will reel in the line and return to its initial state.
- **`attemptNumber`** (**`Number`**): Integer indicating the current catching attempt. It starts as `0`, and every attempt will increase its value, until the maximum number of attempts is reached (The number of attempts will be dictated by the length of the array **`ATTEMPTS_DIFFICULTY`** in **`globals.js`**). Once the maximum number of attempts is reached, it resets to `0`, restarting the attempt cycle.

- **`catchingMinigame`** (**`Object`**): Dictionary of functions to interact with the mini-game.
  - Initial value: The dictionary returned by executing the **`CatchingMinigame`** function. The callback function passed as a parameter to **`CatchingMinigame`** will be executed when the mini-game ends. The function must return the player's state to its initial value.

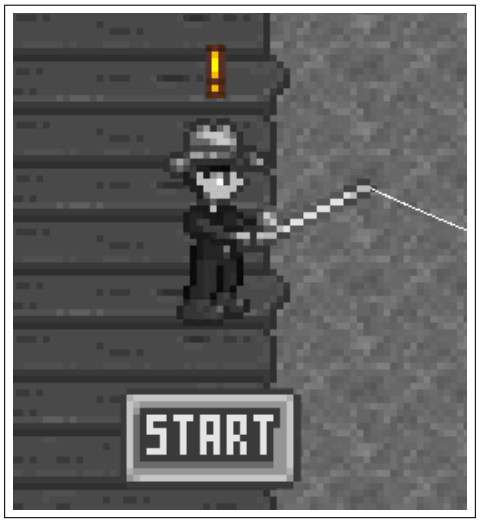

Figure 4: When the fish takes the bait, a yellow exclamation will appear.

3.5.2 Methods to expose

- **`castLine`**: This function will be called by endpoint **`/cast_line`**. The frontend calls this endpoint when the *CAST* button is clicked (only visible if the player is standing). This function must:
  - Return the **`playerState`** as an error code if the player is not standing.
  - Cast the line to the water.
  - Set a timer for the fish to take the bait. After the fish takes the bait, we must:
    - Inform the frontend by resolving the promise created by function **`waitForBite`** presented below. When the frontend receives the response of endpoint **`/wait_for_bite`**, it will display a yellow exclamation as shown in Figure 4 to indicate us that the fish has taken the bait, and that we must click the *START* button to start the mini-game.
    - Set a timer for the fish to escape in case we don't click the *START* button on time. If it's the case, we must return the player to its initial state.
  - Return **`null`**, indicating that we have correctly cast the line to the water.
- **`waitForBite`**: This function will be called by endpoint **`/wait_for_bite`**. The frontend calls this endpoint just after casting the line to the water. This function returns a promise. Its purpose is to **resolve** just when the fish takes the bait (as seen in the explanation of **`castLine`** function). In case that the player is not waiting for the fish to take the bait when this function is called, we must **reject**, passing the **`playerState`** as an error code as its argument. Take into account that this function creates a promise, but it does not resolve it.
  Hint: Remember that in JavaScript, functions can be stored in variables, passed as arguments, and returned from other functions.
- **`reelIn`**: This function will be called by endpoint **`/reel_in`**. The frontend calls this endpoint when the *START* button is clicked (once the line has been cast to the water). There are two possible situations:
  - The fish has not bitten yet:
    - Return the player to its initial state.
    - **Reject** the promise started by **`waitForBite`** function, indicating the frontend to reel in the line.
    - Prevent the timer that makes the fish take the bait (setted in function **`castLine`**) to execute.
  - The fish has bitten. In that case, we must start the catching mini-game by performing the following steps:

    - Update **playerState** accordingly.
    - Starting the mini-game using local variable **catchingMinigame**. Take into account that the exposed function **start** needs the **difficulty** of the mini-game as a parameter. This value can be retrieved using array **ATTEMPTS_DIFFICULTY** (which indicates the difficulty of each attempt) and the local variable **attemptNumber**.
    - Increment **attemptNumber**, cycling back to 0 when the maximum number of attempts is reached.
    - Cancel the timer that makes the fish escape if the **START** button is not pressed in time.

  If the mini-game fails to start, return a dictionary with **{errorCode: playerState}**.
  If the mini-game starts successfully, return **{difficulty: selectedDifficulty}**.

- **updateCatchBarDirection**: This function will be called by endpoints **/move_catch_bar_up** and **stop_moving_catch_bar_up** when the frontend presses or releases the *PULL* button during the mini-game. It must call the **updateCatchBarDirection** function from **catchingMinigame**.
- **getCatchingMiniGameInfo**: This function will be called by endpoint **/get_mini_game_info**. The frontend repeatedly calls this endpoint to retrieve real-time information about the progress bar, the catch bar, and the fish's position in the mini-game. It must call function **getInfo** from **catchingMinigame**.

4 Endpoints

In file **app.js**, you must implement a series of endpoints in order to allow to frontend to communicate with the backend. To do that, we will use the routing functionality of *Express.js* [5]. We list below the endpoints to implement, specifying for each one the task it must perform. All endpoints have method type GET. Just above the implementation of the endpoints, you can see a commented bit of code similar to this:

```
const game = Game();
```

Once you implement function **Game**, uncomment this bit of code, and use it in your endpoints to call the exposed functions of **Game** (remember to also import the function at the beggining of the file).

4.1 /cast_line

Called by the frontend every time they click the *CAST* button while the player is standing. You must call **game**'s exposed function **castLine**. If the result is an error code, you must return it as a json (using **res** object's function **json** [6]), and respond with an status code [7] of **400**. If the function does not return any error code, you must simply respond with an status code of **200**.

4.2 /wait_for_bite

Called by the frontend just after the **/cast_line** endpoint returns **200**. You must call **game**'s exposed function **waitForBite**, which returns a promise. If it **rejects**, you must return the error code provided in the parameter as a json (Example: **{errorCode: "standing"}**) and respond with an status code of **400**. If it **resolves**, you must simply respond with an status code of **200**.

4.3 /reel_in

Called by the frontend every time they click the *START* button when the player has cast the line to the water. You must call **game**'s exposed function **reelIn**. If the result dictionary contains the key **"errorCode"**, it means that the mini-game has not correctly started, and we must respond with an status code of **400**. On the other hand, if it contains key the **"difficulty"**, it means that the mini-game has started, and we must respond with an status code of **200**. In either case, we must return the result dictionary as a json.

4.4 /get_mini_game_info

Frequently called by the frontend once we are playing the mini-game. We must call **game**'s exposed function **getCatchingMiniGameInfo**, and return its result as a json. We must always respond with an status code of **200**.

4.5 `/move_catch_bar_up`

Called by the frontend every time they press down the *PULL* button while playing the mini-game. It must call **`game`**'s exposed function **`updateCatchBarDirection`**, specifying that we want to move the catch bar in the `"up"` direction. We must always respond with an status code of `200`.

4.6 `/stop_moving_catch_bar_up`

Called by the frontend every time they stop pressing the *PULL* button while playing the mini-game. It must call **`game`**'s exposed function **`updateCatchBarDirection`**, specifying that we want to move the catch bar in the `"down"` direction. We must always respond with an status code of `200`.

5 Web Sockets

You might have noticed that the polling mechanism the frontend uses to repeatedly retrieve information about the mini-game (via the **`/get_mini_game_info`** endpoint) is quite inneficient. Since the frontend does not know when the fish, the catch bar, or the progress bar will change direction, it must constantly request this information from the backend, even when no changes have occurred between calls.

To improve this, a good option in JavaScript is to use **WebSockets**. WebSockets allow the frontend to establish a bidirectional connection with the backend. Once connected, the backend can send updates about changes in the direction of the fish, the catch bar, or the progress bar only when they occur.

The provided frontend attempts to establish a WebSocket connection to **`http`**`://localhost:8080` as soon as it loads. If the connection is successful, it will display the following message in the browser's console:
`"WebSocket connection detected. Prepared to listen to messages."`

Instead of using **`/get_mini_game_info`**, the frontend will then wait for specific messages to arrive via the WebSocket connection. Below are the types of messages it will expect:

- When the fish changes **direction**, it expects the following dictionary:

```
{
    'type': 'fishInfo',
    'data': {
        lastSwapPosition,
        lastSwapAt,
        direction,
        speed
    }
}
```

- When the catch bar changes **direction**, it expects the following dictionary:

```
{
    'type': 'catchBarInfo',
    'data': {
        lastSwapPosition,
        lastSwapAt,
        direction
    }
}
```

- When the progress bar changes **direction** or **state**, it expects the following dictionary:

```
{
    'type': 'progressBarInfo',
    'data': {
        lastSwapPosition,
        lastSwapAt,
        direction,
        state
    }
}
```

Considerations:

- WebSockets sends data using strings. Before sending the dictionary, you must convert it using function **`JSON.stringify`** [8].
- You are allowed to add parameters, local variables and exposed methods to the presented functions in order to achieve this functionality.
- Implementing this mechanism **does not exept you** from also implementing the **`/get_mini_game_info`** endpoint and its underlying functions.

6 Testing

In order to check if you have correctly implemented the **`Game`** function, we provide you are series of tests you can execute using command **`npm run test`**. They are implemented using *jest*. In case the a test fails, *jest* will tell you the expected output, the obtained output, and the part of the test where it failed. For all the tests to run without failing, you need to correctly implement all clousures, but is not necessary to implement the endpoints. To test that the endpoints work correctly, you can use the provided frontend that will be available on **`http://localhost:8081`** once you run the backend using **`node app.js`**.

References

[1] **npm website**: `https://www.npmjs.com`

[2] **Express framework**: `https://expressjs.com`

[3] **Jest testing framework**: `https://jestjs.io`

[4] **NodeJS website**: `https://nodejs.org`

[5] **Express basic routing**: `https://expressjs.com/en/starter/basic-routing.html`

[6] **res.json function**: `https://developer.mozilla.org/en-US/docs/Web/API/Response/json_static`

[7] **res.status function**: `https://developer.mozilla.org/en-US/docs/Web/API/Response/status`

[8] **JSON.stringify function**: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`