**Documentation for promotionController.js**

This file serves as a controller for managing various Promotional features within the application. It handles tasks such as to retrieve data related to recharge and betting from a database. It includes functions to calculate statistics such as the number of recharges, total recharge amount, first recharge amount, and total betting amount for a given phone number. There are also functions to retrieve the subordinates' data for a given code, and calculate statistics for the direct and indirect subordinates.

**Functions Overview**

The promotionController.js file contains numerous functions, each responsible for specific routes and functionalities within the admin interface of the application.

**Functions**

**getOrdinal(n)**

This function converts a number into its ordinal form. For example, it converts 1 to 1st, 2 to 2nd, 3 to 3rd, and all other numbers to their respective ordinal forms (e.g. 4th, 11th, 21st).

**Parameters**

* n (number): The number to convert into its ordinal form.

**Returns**

The converted ordinal form of the given number.

*function* getOrdinal(*n*) {

*let* s = ["th", "st", "nd", "rd"],

    v = *n* % 100;

  return *n* + (s[(v - 20) % 10] || s[v] || s[0]);

}

**Example**

console.log(getOrdinal(1)); // Output: 1st

console.log(getOrdinal(2)); // Output: 2nd

console.log(getOrdinal(3)); // Output: 3rd

console.log(getOrdinal(4)); // Output: 4th

**getSubordinateDataByPhone(phone)**

This asynchronous function retrieves various data related to recharge and betting for a given phone number.

**Parameters**

* phone (string): The phone number for which to retrieve the data.

**Returns**

An object containing the following properties:

* rechargeQuantity (number): The number of recharges for the given phone number.
* rechargeAmount (number): The total recharge amount for the given phone number.
* firstRechargeAmount (number): The amount of the first recharge for the given phone number.
* bettingAmount (number): The total betting amount for the given phone number. This is calculated by summing the betting amounts from different games.

*const* getSubordinateDataByPhone = *async* (*phone*) *=>* {

*const* [[row\_1]] = await connection.execute(

    "SELECT COUNT(\*) AS `count` FROM `recharge` WHERE `phone` = ? AND `status` = ?",

    [*phone*, 1],

  );

*const* rechargeQuantity = row\_1.count;

*const* [[row\_2]] = await connection.execute(

    "SELECT SUM(money) AS `sum` FROM `recharge` WHERE `phone` = ? AND `status` = ?",

    [*phone*, 1],

  );

*const* rechargeAmount = row\_2.sum;

*const* [[row\_3]] = await connection.execute(

    "SELECT SUM(money) AS `sum` FROM `recharge` WHERE `phone` = ? AND `status` = ? ORDER BY id LIMIT 1 ",

    [*phone*, 1],

  );

*const* firstRechargeAmount = row\_3.sum;

*const* [gameWingo] = await connection.query(

    "SELECT SUM(money) as totalBettingAmount FROM minutes\_1 WHERE phone = ?",

    [*phone*],

  );

*const* gameWingoBettingAmount = gameWingo[0].totalBettingAmount || 0;

*const* [gameK3] = await connection.query(

    "SELECT SUM(money) as totalBettingAmount FROM result\_k3 WHERE phone = ?",

    [*phone*],

  );

*const* gameK3BettingAmount = gameK3[0].totalBettingAmount || 0;

*const* [game5D] = await connection.query(

    "SELECT SUM(money) as totalBettingAmount FROM result\_5d WHERE phone = ?",

    [*phone*],

  );

*const* game5DBettingAmount = game5D[0].totalBettingAmount || 0;

  return {

    rechargeQuantity,

    rechargeAmount,

    firstRechargeAmount,

    bettingAmount:

      parseInt(gameWingoBettingAmount) +

      parseInt(gameK3BettingAmount) +

      parseInt(game5DBettingAmount),

  };

};

**Example**

const data = await getSubordinateDataByPhone("1234567890");

console.log(data.rechargeQuantity);

console.log(data.rechargeAmount);

console.log(data.firstRechargeAmount);

console.log(data.bettingAmount);

**getSubordinatesListDataByCode(code, startDate)**

This asynchronous function retrieves data for the list of subordinates for a given code. The data includes information such as the number of subordinates, their recharge and betting statistics, etc.

**Parameters**

* code (string): The code for which to retrieve the subordinates' data.
* startDate (string, optional): The start date to filter the subordinates' data. If provided, only the subordinates whose time is less than or equal to the start date will be included in the result.

**Returns**

An object containing the following properties:

* subordinatesList (array): An array of objects representing the subordinates. Each object contains information such as the code, phone number, id\_user, level, time, recharge quantity, recharge amount, betting amount, first recharge amount, etc.
* subordinatesCount (number): The number of subordinates.
* subordinatesRechargeQuantity (number): The total recharge quantity of all subordinates.
* subordinatesRechargeAmount (number): The total recharge amount of all subordinates.
* subordinatesWithDepositCount (number): The number of subordinates who have made at least one recharge.
* subordinatesWithBettingCount (number): The number of subordinates who have made at least one bet.
* subordinatesBettingAmount (number): The total betting amount of all subordinates.
* subordinatesFirstDepositAmount (number): The total amount of the first recharge of all subordinates.

*const* getSubordinatesListDataByCode = *async* (*code*, *startDate*) *=>* {

*let* [subordinatesList] = *startDate*

    ? await connection.execute(

        "SELECT `code`, `phone`, `id\_user`, `level`, `time` FROM `users` WHERE `invite` = ? AND time <= ?",

        [*code*, *startDate*],

      )

    : await connection.execute(

        "SELECT `code`, `phone`, `id\_user`, `time` FROM `users` WHERE `invite` = ?",

        [*code*],

      );

*let* subordinatesCount = subordinatesList.length;

*let* subordinatesRechargeQuantity = 0;

*let* subordinatesRechargeAmount = 0;

*let* subordinatesWithDepositCount = 0;

*let* subordinatesFirstDepositAmount = 0;

*let* subordinatesWithBettingCount = 0;

*let* subordinatesBettingAmount = 0;

  for (*let* index = 0; index < subordinatesList.length; index++) {

*const* subordinate = subordinatesList[index];

*const* {

      rechargeQuantity,

      rechargeAmount,

      bettingAmount,

      firstRechargeAmount,

    } = await getSubordinateDataByPhone(subordinate.phone);

    subordinatesRechargeQuantity += parseInt(rechargeQuantity) || 0;

    subordinatesRechargeAmount += parseInt(rechargeAmount) || 0;

    subordinatesList[index]["rechargeQuantity"] =

      parseInt(rechargeQuantity) || 0;

    subordinatesList[index]["rechargeAmount"] = parseInt(rechargeAmount) || 0;

    subordinatesList[index]["bettingAmount"] = parseInt(bettingAmount) || 0;

    subordinatesList[index]["firstRechargeAmount"] =

      parseInt(firstRechargeAmount) || 0;

    subordinatesList[index]["level"] = subordinatesList[index]["level"] || 0;

    subordinatesList[index]["commission"] =

      subordinatesList[index]["commission"] || 0;

    subordinatesWithBettingCount += parseInt(bettingAmount) > 0 ? 1 : 0;

    subordinatesBettingAmount += parseInt(bettingAmount);

    subordinatesFirstDepositAmount += parseInt(firstRechargeAmount) || 0;

    if (rechargeAmount > 0) {

      subordinatesWithDepositCount++;

    }

  }

  return {

    subordinatesList,

    subordinatesCount,

    subordinatesRechargeQuantity,

    subordinatesRechargeAmount,

    subordinatesWithDepositCount,

    subordinatesWithBettingCount,

    subordinatesBettingAmount,

    subordinatesFirstDepositAmount,

  };

};

**Example**

const data = await getSubordinatesListDataByCode("ABC123");

console.log(data.subordinatesList);

console.log(data.subordinatesCount);

console.log(data.subordinatesRechargeQuantity);

console.log(data.subordinatesRechargeAmount);

console.log(data.subordinatesWithDepositCount);

console.log(data.subordinatesWithBettingCount);

console.log(data.subordinatesBettingAmount);

console.log(data.subordinatesFirstDepositAmount);

**getOneLevelTeamSubordinatesData(directSubordinatesList)**

This asynchronous function retrieves data for the subordinates in the direct subordinates list. It includes statistics such as the number of subordinates, their recharge and betting statistics, etc.

**Parameters**

* directSubordinatesList (array): An array of objects representing the direct subordinates. Each object should contain the code of a direct subordinate.

**Returns**

An object containing the following properties:

* oneLevelTeamSubordinatesCount (number): The total number of subordinates in the direct subordinates list and their indirect subordinates.
* oneLevelTeamSubordinatesRechargeQuantity (number): The total recharge quantity of all subordinates in the direct subordinates list and their indirect subordinates.
* oneLevelTeamSubordinatesRechargeAmount (number): The total recharge amount of all subordinates in the direct subordinates list and their indirect subordinates.
* oneLevelTeamSubordinatesWithDepositCount (number): The number of subordinates in the direct subordinates list and their indirect subordinates who have made at least one recharge.
* oneLevelTeamSubordinatesList (array): An array of objects representing the subordinates in the direct subordinates list and their indirect subordinates. Each object contains information such as the code, phone number, id\_user, level, time, recharge quantity, recharge amount, betting amount, first recharge amount, etc.

*const* getOneLevelTeamSubordinatesData = *async* (*directSubordinatesList*) *=>* {

*let* oneLevelTeamSubordinatesCount = 0;

*let* oneLevelTeamSubordinatesRechargeQuantity = 0;

*let* oneLevelTeamSubordinatesRechargeAmount = 0;

*let* oneLevelTeamSubordinatesWithDepositCount = 0;

*let* oneLevelTeamSubordinatesList = [];

  for (*const* directSubordinate of *directSubordinatesList*) {

*const* indirectSubordinatesData = await getSubordinatesListDataByCode(

      directSubordinate.code,

    );

    oneLevelTeamSubordinatesList = [

      ...oneLevelTeamSubordinatesList,

      ...indirectSubordinatesData.subordinatesList,

    ];

    oneLevelTeamSubordinatesCount += indirectSubordinatesData.subordinatesCount;

    oneLevelTeamSubordinatesRechargeQuantity +=

      indirectSubordinatesData.subordinatesRechargeQuantity;

    oneLevelTeamSubordinatesRechargeAmount +=

      indirectSubordinatesData.subordinatesRechargeAmount;

    oneLevelTeamSubordinatesWithDepositCount +=

      indirectSubordinatesData.subordinatesWithDepositCount;

  }

  return {

    oneLevelTeamSubordinatesCount,

    oneLevelTeamSubordinatesRechargeQuantity,

    oneLevelTeamSubordinatesRechargeAmount,

    oneLevelTeamSubordinatesWithDepositCount,

    oneLevelTeamSubordinatesList,

  };

};

**Example**

const data = await getOneLevelTeamSubordinatesData([

{ code: "ABC123" },

{ code: "DEF456" },

]);

console.log(data.oneLevelTeamSubordinatesCount);

console.log(data.oneLevelTeamSubordinatesRechargeQuantity);

console.log(data.oneLevelTeamSubordinatesRechargeAmount);

console.log(data.oneLevelTeamSubordinatesWithDepositCount);

console.log(data.oneLevelTeamSubordinatesList);

### createInviteMap

This function takes an array of user objects (rows) and creates a map of users grouped by their invite codes. It returns the invite map, where each invite code is a key that points to an array of users with that invite code.

*const* createInviteMap = (*rows*) *=>* {

*const* inviteMap = {};

*rows*.forEach((*user*) *=>* {

    if (!inviteMap[*user*.invite]) {

      inviteMap[*user*.invite] = [];

    }

    inviteMap[*user*.invite].push(*user*);

  });

  return inviteMap;

};

The given code defines a function createInviteMap that takes a slice of User structs as input, loops through each user, and organizes them into a map based on their invite field.

Here's a breakdown of the code:

1. func createInviteMap(rows []User) map[string][]User {: This line declares a function named createInviteMap that accepts a slice of User structs as input and returns a map with string keys and slice of User values.
2. inviteMap := make(map[string][]User): Initializes an empty map to store the users based on their invite code.
3. for \_, user := range rows {: Iterates over each user in the input slice rows.
4. if \_, ok := inviteMap[user.Invite]; !ok {: Checks if the invite code of the current user already exists in the map. If not, initializes an empty slice for that invite code.
5. inviteMap[user.Invite] = append(inviteMap[user.Invite], user): Appends the current user to the slice corresponding to their invite code in the map.
6. return inviteMap: Returns the map containing users grouped by their invite codes.

### getLevelUsers

This function calculates the users at each level based on the provided invite map, user code, current level number, and maximum level. It uses recursion to traverse the invite tree and find all the users at each level. It returns an array of user objects with their respective user levels.

/\*\*

\* @param {Object} inviteMap - The invite map.

\* @param {string} userCode - The user code.

\* @param {number} currentLevel - The current level number.

\* @param {number} maxLevel - The maximum level.

\* @returns {Array} - An array of user objects with user levels.

\*/

*const* getLevelUsers = (*inviteMap*, *userCode*, *currentLevel*, *maxLevel*) *=>* {

  if (*currentLevel* > *maxLevel*) return [];

*const* levelUsers = *inviteMap*[*userCode*] || [];

  if (levelUsers.length === 0) return [];

  return levelUsers.flatMap((*user*) *=>* [

    { ...*user*, user\_level: *currentLevel* },

    ...getLevelUsers(*inviteMap*, *user*.code, *currentLevel* + 1, *maxLevel*),

  ]);

};

The provided code defines a recursive function getLevelUsers that takes in parameters inviteMap (an object), userCode (a string), currentLevel (a number), and maxLevel (a number).

1. It first checks if the currentLevel is greater than maxLevel, and if so, returns an empty array.
2. Then it extracts the levelUsers from inviteMap based on the provided userCode. If the levelUsers array is empty, it also returns an empty array.
3. Next, it uses flatMap to iterate over each user in levelUsers. For each user, it creates a new object by spreading the properties of the user object and adding a new property user\_level with the value of currentLevel. It then recursively calls getLevelUsers with updated parameters to retrieve users for the next level and flattens the resulting arrays.
4. Overall, the function recursively fetches users at different levels based on the invite map and current level, stopping when the currentLevel exceeds the maxLevel. The result is an array of user objects with an added user\_level property indicating the level they are associated with.

### getUserLevels

This function combines the createInviteMap and getLevelUsers functions to calculate the user levels for a given array of user objects, user code, and maximum level. It returns an object containing the user levels and the users directly referred by the user code.

/\*\*

\* @param {Array} rows - An array of user objects.

\* @param {string} userCode - The user code.

\* @param {number} maxLevel - The maximum level.

\* @returns {Object} - An object with user levels and level 1 referrals.

\*/

*const* getUserLevels = (*rows*, *userCode*, *maxLevel* = 10) *=>* {

*const* inviteMap = createInviteMap(*rows*);

*const* usersByLevels = getLevelUsers(inviteMap, *userCode*, 1, *maxLevel*);

  return { usersByLevels, level1Referrals: inviteMap[*userCode*] };

};

The provided code defines a function getUserLevels that takes three parameters - rows, userCode, and maxLevel with a default value of 10. Inside the function, it first calls a function createInviteMap(rows) to create a map of user invitations.

Next, it calls another function getLevelUsers with the created inviteMap, userCode, starting level 1, and the maxLevel provided as arguments. This function retrieves the users at different levels based on the invite map.

The function returns an object with two properties:

1. usersByLevels: This contains the users grouped by their levels based on the invite map.
2. level1Referrals: This retrieves the level 1 referrals of the userCode from the invite map.

### userStats

This asynchronous function retrieves user statistics within a given time range and optional phone number. It executes a database query using the connection.query function and returns the resulting rows. The function expects the connection object to be defined and connected to the database beforehand.

/\*\*

\* @param {number} startTime - The start time of the time range.

\* @param {number} endTime - The end time of the time range.

\* @param {string} [phone=""] - The phone number (optional).

\* @returns {Promise} - A promise that resolves to an array of user statistics rows.

\*/

*const* userStats = *async* (*startTime*, *endTime*, *phone* = "") *=>* {

*const* [rows] = await connection.query(

    `

      SELECT

          u.phone,

          u.invite,

          u.code,

          u.time,

          u.id\_user,

          COALESCE(r.total\_deposit\_amount, 0) AS total\_deposit\_amount,

          COALESCE(r.total\_deposit\_number, 0) AS total\_deposit\_number,

          COALESCE(m.total\_bets, 0) AS total\_bets,

          COALESCE(m.total\_bet\_amount, 0) AS total\_bet\_amount,

          COALESCE(c.total\_commission, 0) AS total\_commission

      FROM

          users u

      LEFT JOIN

          (

              SELECT

                  phone,

                  SUM(CASE WHEN status = 1 THEN COALESCE(money, 0) ELSE 0 END) AS total\_deposit\_amount,

                  COUNT(CASE WHEN status = 1 THEN phone ELSE NULL END) AS total\_deposit\_number

              FROM

                  recharge

              WHERE

                  time > ? AND time < ?

              GROUP BY

                  phone

          ) r ON u.phone = r.phone

      LEFT JOIN

          (

              SELECT

                  phone,

                  COALESCE(SUM(total\_bet\_amount), 0) AS total\_bet\_amount,

                  COALESCE(SUM(total\_bets), 0) AS total\_bets

              FROM (

                  SELECT

                      phone,

                      SUM(money + fee) AS total\_bet\_amount,

                      COUNT(\*) AS total\_bets

                  FROM minutes\_1

                  WHERE time >= ? AND time <= ?

                  GROUP BY phone

                  UNION ALL

                  SELECT

                      phone,

                      SUM(money + fee) AS total\_bet\_amount,

                      COUNT(\*) AS total\_bets

                  FROM trx\_wingo\_bets

                  WHERE time >= ? AND time <= ?

                  GROUP BY phone

              ) AS combined

              GROUP BY phone

          ) m ON u.phone = m.phone

      LEFT JOIN

          (

              SELECT

                  from\_user\_phone AS phone,

                  SUM(money) AS total\_commission

              FROM

                  commissions

              WHERE

                  time > ? AND time <= ? AND phone = ?

              GROUP BY

                  from\_user\_phone

          ) c ON u.phone = c.phone

      GROUP BY

          u.phone

      ORDER BY

          u.time DESC;

      `,

    [

*startTime*,

*endTime*,

*startTime*,

*endTime*,

*startTime*,

*endTime*,

*startTime*,

*endTime*,

*phone*,

    ],

  );

  return rows;

};

The provided code is an asynchronous JavaScript function that retrieves user statistics from multiple tables in a database based on the given time range and optional phone number. The function constructs a complex SQL query using JOIN operations to fetch data from different tables.

Here is a breakdown of the SQL query:

1. The query selects various fields from the 'users' table and calculates aggregated values from related tables for deposit amounts, bets, and commissions.
2. Multiple LEFT JOINs are used with subqueries to gather deposit, betting, and commission information based on specific conditions and time ranges.
3. The query groups the results by user phone number and orders them in descending order based on user registration time.
4. The function then executes the constructed query using the 'connection.query' method with the provided parameters and awaits the result, which is stored in the 'rows' variable and returned.

### getCommissionStatsByTime

This asynchronous function retrieves commission statistics for a specific time and phone number. It executes a database query using the connection.execute function and returns the resulting row. The function expects the connection object to be defined and connected to the database beforehand.

/\*\*

\* @param {number} time - The specific time.

\* @param {string} phone - The phone number.

\* @returns {Promise

*const* getCommissionStatsByTime = *async* (*time*, *phone*) *=>* {

*const* { startOfYesterdayTimestamp, endOfYesterdayTimestamp } =

    yesterdayTime();

*const* [commissionRow] = await connection.execute(

    `

      SELECT

          time,

          SUM(COALESCE(c.money, 0)) AS total\_commission,

          SUM(CASE

              WHEN c.time >= ?

              THEN COALESCE(c.money, 0)

              ELSE 0

          END) AS last\_week\_commission,

          SUM(CASE

              WHEN c.time > ? AND c.time <= ?

              THEN COALESCE(c.money, 0)

              ELSE 0

          END) AS yesterday\_commission

      FROM

          commissions c

      WHERE

          c.phone = ?

      `,

    [*time*, startOfYesterdayTimestamp, endOfYesterdayTimestamp, *phone*],

  );

  return commissionRow?.[0] || {};

};

This code defines an asynchronous function getCommissionStatsByTime that takes time and phone as parameters and retrieves commission statistics based on the provided time and phone.

1. It first destructures startOfYesterdayTimestamp and endOfYesterdayTimestamp from the result of calling the function yesterdayTime().
2. Then it executes a SQL query to fetch commission data for the given time and phone, calculating total commission, last week's commission, and yesterday's commission using different conditions. The results are stored in commissionRow.
3. Finally, the function returns the first row of commissionRow if it exists, or an empty object if it doesn't.

## Function: subordinatesDataAPI

This function is an API endpoint that retrieves data related to subordinates. Here's what it does:

*const* subordinatesDataAPI = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* startOfWeek = getStartOfWeekTimestamp();

*const* { startOfYesterdayTimestamp, endOfYesterdayTimestamp } =

      yesterdayTime();

*const* [userRow] = await connection.execute(

      "SELECT \* FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* commissions = await getCommissionStatsByTime(startOfWeek, user.phone);

*const* userStatsData = await userStats(

      startOfYesterdayTimestamp,

      endOfYesterdayTimestamp,

    );

*const* { usersByLevels = [], level1Referrals = [] } = getUserLevels(

      userStatsData,

      user.code,

    );

*const* directSubordinatesCount = level1Referrals.length;

*const* noOfRegisteredSubordinates = level1Referrals.filter(

      (*user*) *=>* *user*.time >= startOfYesterdayTimestamp,

    ).length;

*const* directSubordinatesRechargeQuantity = level1Referrals.reduce(

      (*acc*, *curr*) *=>* *acc* + *curr*.total\_deposit\_number,

      0,

    );

*const* directSubordinatesRechargeAmount = level1Referrals.reduce(

      (*acc*, *curr*) *=>* *acc* + +*curr*.total\_deposit\_amount,

      0,

    );

*const* directSubordinatesWithDepositCount = level1Referrals.filter(

      (*user*) *=>* *user*.total\_deposit\_number === 1,

    ).length;

*const* teamSubordinatesCount = usersByLevels.length;

*const* noOfRegisterAll = usersByLevels.filter(

      (user) *=>* user.time >= startOfYesterdayTimestamp,

    );

*const* noOfRegisterAllSubordinates = noOfRegisterAll.length;

*const* teamSubordinatesRechargeQuantity = usersByLevels.reduce(

      (*acc*, *curr*) *=>* *acc* + *curr*.total\_deposit\_number,

      0,

    );

*const* teamSubordinatesRechargeAmount = usersByLevels.reduce(

      (*acc*, *curr*) *=>* *acc* + +*curr*.total\_deposit\_amount,

      0,

    );

*const* teamSubordinatesWithDepositCount = usersByLevels.filter(

      (*user*) *=>* *user*.total\_deposit\_number === 1,

    ).length;

*const* totalCommissions = commissions?.total\_commission || 0;

*const* totalCommissionsThisWeek = commissions?.last\_week\_commission || 0;

*const* totalCommissionsYesterday = commissions?.yesterday\_commission || 0;

    return *res*.status(200).json({

      data: {

        directSubordinatesCount,

        noOfRegisteredSubordinates,

        directSubordinatesRechargeQuantity,

        directSubordinatesRechargeAmount,

        directSubordinatesWithDepositCount,

        teamSubordinatesCount,

        noOfRegisterAllSubordinates,

        teamSubordinatesRechargeQuantity,

        teamSubordinatesRechargeAmount,

        teamSubordinatesWithDepositCount,

        totalCommissions,

        totalCommissionsThisWeek,

        totalCommissionsYesterday,

      },

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. It extracts an authentication token from the request cookies.
2. It retrieves the start of the current week's timestamp using the getStartOfWeekTimestamp() function.
3. It retrieves the start and end timestamps of yesterday using the yesterdayTime() function.
4. It queries the database to fetch the user information based on the authentication token.
5. If the user is not found, it returns a status code of 401 (Unauthorized) and an error message in JSON format.
6. It fetches the commission stats for the user by calling the getCommissionStatsByTime() function.
7. It fetches the user stats data for the previous day by calling the userStats() function with the appropriate timestamps.
8. It calls the getUserLevels() function with the user stats data and user code to get the user's subordinates at different levels.
9. It performs various calculations to determine statistics related to direct subordinates and team subordinates.
10. Finally, it returns the extracted data in a JSON response with a status code of 200 (OK).

## Function: subordinatesDataByTimeAPI

This function is another API endpoint that retrieves subordinates data based on a given time range. Here's what it does:

*const* subordinatesDataByTimeAPI = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `code`,phone, `invite` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

*const* startDate = +*req*.query.startDate;

*const* endDate = getTimeBasedOnDate(startDate);

*const* searchFromUid = *req*.query.id || "";

*const* levelFilter = *req*.query.level;

*const* page = parseInt(*req*.query.page) || 1;

*const* limit = parseInt(*req*.query.limit) || 10;

*const* offset = (page - 1) \* limit;

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* userStatsData = await userStats(startDate, endDate, user.phone);

*const* { usersByLevels = [] } = getUserLevels(userStatsData, user.code);

*const* filteredUsers = usersByLevels.filter(

      (*user*) *=>*

*user*.id\_user.includes(searchFromUid) &&

        (levelFilter !== "All" ? *user*.user\_level === +levelFilter : true),

    );

    // const usersFilterByPositiveData = filteredUsers.filter(

    //   (user) =>

    //     user.total\_deposit\_number > 0 ||

    //     user.total\_deposit\_amount > 0 ||

    //     user.total\_bets > 0,

    // );

*const* sortedUsersByBet = filteredUsers.sort((*a*, *b*) *=>* *b*.total\_bet\_amount - *a*.total\_bet\_amount);

*const* subordinatesRechargeQuantity = filteredUsers.reduce(

      (*acc*, *curr*) *=>* *acc* + *curr*.total\_deposit\_number,

      0,

    );

*const* subordinatesRechargeAmount = filteredUsers.reduce(

      (*acc*, *curr*) *=>* *acc* + +*curr*.total\_deposit\_amount,

      0,

    );

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*for bets \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

*const* subordinatesWithBetting = filteredUsers.filter(

      (*user*) *=>* *user*.total\_bets > 0,

    );

*const* subordinatesWithBettingCount = subordinatesWithBetting.length;

*const* subordinatesBettingAmount = subordinatesWithBetting

      .reduce((*acc*, *curr*) *=>* *acc* + +*curr*.total\_bet\_amount, 0)

      .toFixed();

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*for first deposit \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

*const* subordinatesWithFirstDeposit = filteredUsers.filter(

      (*user*) *=>* *user*.total\_deposit\_number === 1,

    );

*const* subordinatesWithFirstDepositCount =

      subordinatesWithFirstDeposit.length;

*const* subordinatesWithFirstDepositAmount =

      subordinatesWithFirstDeposit.reduce(

        (*acc*, *curr*) *=>* *acc* + +*curr*.total\_deposit\_amount,

        0,

      );

    //for pagination

*const* paginatedUsers = sortedUsersByBet.slice(

      offset,

      offset + limit,

    );

*const* totalUsers = sortedUsersByBet.length;

*const* totalPages = Math.ceil(totalUsers / limit);

*res*.json({

      status: true,

      meta: {

        totalPages,

        currentPage: page,

      },

      data: {

        usersByLevels: paginatedUsers,

        subordinatesRechargeQuantity,

        subordinatesRechargeAmount,

        subordinatesWithBettingCount,

        subordinatesBettingAmount,

        subordinatesWithFirstDepositCount,

        subordinatesWithFirstDepositAmount,

      },

      message: "Successfully fetched subordinates data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. It extracts an authentication token from the request cookies.
2. It queries the database to fetch the user information based on the authentication token.
3. It checks if the user is not found and returns a status code of 401 (Unauthorized) with an error message in JSON format.
4. It retrieves the start and end timestamps based on the provided start date using the getTimeBasedOnDate() function.
5. It extracts optional query parameters such as id and level from the request.
6. It performs various filtering, sorting, and calculation operations on the retrieved users' data to generate the desired statistics.
7. It provides paginated results based on the requested page and limit.
8. Finally, it returns the formatted results in a JSON response with a status code of 200 (OK).

## Function: subordinatesAPI

This function is also an API endpoint that retrieves subordinates data based on a specified type. Here's what it does:

*const* subordinatesAPI = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `code`,phone, `invite` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* type = *req*.query.type || "today";

*const* { startOfYesterdayTimestamp, endOfYesterdayTimestamp } =

      yesterdayTime();

*const* { startOfMonthTimestamp, endOfMonthTimestamp } = monthTime();

*const* startDate =

      type === "today"

        ? getTodayStartTime()

        : type === "yesterday"

          ? startOfYesterdayTimestamp

          : type === "this month"

            ? startOfMonthTimestamp

            : "";

*const* endDate =

      type === "today"

        ? new *Date*().getTime()

        : type === "yesterday"

          ? endOfYesterdayTimestamp

          : type === "this month"

            ? endOfMonthTimestamp

            : "";

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* userStatsData = await userStats(startDate, endDate, user.phone);

*const* { level1Referrals } = getUserLevels(userStatsData, user.code);

*const* users = level1Referrals

      .map((*user*) *=>* {

*const* { phone, id\_user: uid, time } = *user*;

*const* phoneFormat = phone.slice(0, 3) + "\*\*\*\*" + phone.slice(7);

*const* timeUtc = new *Date*(parseInt(time))

          .toLocaleString("en-GB", {

            year: "numeric",

            month: "2-digit",

            day: "2-digit",

            hour: "2-digit",

            minute: "2-digit",

            hour12: false,

            timeZone: "UTC",

          })

          .replace(",", "");

        if (*user*.time >= startDate)

          return { phone: phoneFormat, uid, time: timeUtc };

        else return null;

      })

      .filter(*Boolean*);

*res*.status(200).json({

      status: true,

      type,

      users,

      message: "Successfully fetched subordinates data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. It extracts an authentication token from the request cookies.
2. It queries the database to fetch the user information based on the authentication token.
3. It retrieves the start and end timestamps based on the specified type (today, yesterday, this month) using appropriate functions (getTodayStartTime(), yesterdayTime(), monthTime() ).
4. It checks if the user is not found and returns a status code of 401 (Unauthorized) with an error message in JSON format.
5. It retrieves the user stats data for the specified time range using the userStats() function.
6. It calls the getUserLevels() function with the user stats data and user code to get the user's subordinates at different levels.
7. It performs additional filtering and formatting to generate the required data structure.
8. Finally, it returns the extracted data in a JSON response with a status code of 200 (OK).

Now that we have analyzed the code file and its functions, we have a better understanding of how it works. These functions can be utilized to fetch and process subordinates' data in a software project.

## Invitation Bonus Functions

### getInvitationBonus

This function retrieves the invitation bonus data for the authenticated user. It requires an authentication token to identify the user in the database. The function performs the following steps:

*const* getInvitationBonus = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `code`, `invite`, `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ status: false, message: "Unauthorized" });

    }

*const* directSubordinatesData = await getSubordinatesListDataByCode(

      user.code,

    );

*let* directSubordinatesCount = directSubordinatesData.subordinatesCount;

*let* directSubordinatesRechargeAmount =

      directSubordinatesData.subordinatesRechargeAmount;

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.INVITATION\_BONUS, user.phone],

    );

*const* invitationBonusData = InvitationBonusList.map((*item*) *=>* {

*const* currentNumberOfDeposits =

        directSubordinatesData.subordinatesList.filter(

          (*subordinate*) *=>*

*subordinate*.rechargeAmount >= *item*.amountOfRechargePerPerson,

        ).length;

      return {

        id: *item*.id,

        isFinished:

          directSubordinatesCount >= *item*.numberOfInvitedMembers &&

          currentNumberOfDeposits >= *item*.numberOfDeposits,

        isClaimed: claimedRewardsRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

        required: {

          numberOfInvitedMembers: *item*.numberOfInvitedMembers,

          numberOfDeposits: *item*.numberOfDeposits,

          amountOfRechargePerPerson: *item*.amountOfRechargePerPerson,

        },

        current: {

          numberOfInvitedMembers: Math.min(

            directSubordinatesCount,

*item*.numberOfInvitedMembers,

          ),

          numberOfDeposits: Math.min(

            currentNumberOfDeposits,

*item*.numberOfDeposits,

          ),

          amountOfRechargePerPerson: Math.min(

            directSubordinatesRechargeAmount,

*item*.amountOfRechargePerPerson,

          ),

        },

        bonusAmount: *item*.bonusAmount,

      };

    });

    return *res*.status(200).json({

      data: invitationBonusData,

      status: true,

      message: "Successfully fetched invitation bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. Verifies the authentication token.
2. Retrieves user data from the database based on the token.
3. Calculates the number of direct subordinates and their total recharge amount.
4. Retrieves claimed rewards data for the user.
5. Calculates the invitation bonus data based on the InvitationBonusList and direct subordinates' data.
6. Returns the invitation bonus data in the response.

Example usage:

GET /invitation-bonus

Request:

{

"headers": {

"Cookie": "auth=AUTH\_TOKEN"

}

}

Response:

{

"data": [

{

"id": 1,

"isFinished": false,

"isClaimed": false,

"required": {

"numberOfInvitedMembers": 3,

"numberOfDeposits": 3,

"amountOfRechargePerPerson": 555

},

"current": {

"numberOfInvitedMembers": 1,

"numberOfDeposits": 2,

"amountOfRechargePerPerson": 400

},

"bonusAmount": 199

},

{

...

}

],

"status": true,

"message": "Successfully fetched invitation bonus data"

}

### claimInvitationBonus

This function allows the authenticated user to claim an invitation bonus. It requires an authentication token and the claim ID of the bonus to be claimed. The function performs the following steps:

*const* claimInvitationBonus = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* invitationBonusId = *req*.body.claim\_id;

*const* [userRow] = await connection.execute(

      "SELECT `code`, `invite`, `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* directSubordinatesData = await getSubordinatesListDataByCode(

      user.code,

    );

*let* directSubordinatesCount = directSubordinatesData.subordinatesCount;

*let* directSubordinatesRechargeAmount =

      directSubordinatesData.subordinatesRechargeAmount;

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.INVITATION\_BONUS, user.phone],

    );

*const* invitationBonusData = InvitationBonusList.map((*item*) *=>* {

*const* currentNumberOfDeposits =

        directSubordinatesData.subordinatesList.filter(

          (*subordinate*) *=>*

*subordinate*.rechargeAmount >= *item*.amountOfRechargePerPerson,

        ).length;

      return {

        id: *item*.id,

        isFinished:

          directSubordinatesCount >= *item*.numberOfInvitedMembers &&

          currentNumberOfDeposits >= *item*.numberOfDeposits,

        isClaimed: claimedRewardsRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

        required: {

          numberOfInvitedMembers: *item*.numberOfInvitedMembers,

          numberOfDeposits: *item*.numberOfDeposits,

          amountOfRechargePerPerson: *item*.amountOfRechargePerPerson,

        },

        current: {

          numberOfInvitedMembers: Math.min(

            directSubordinatesCount,

*item*.numberOfInvitedMembers,

          ),

          numberOfDeposits: Math.min(

            currentNumberOfDeposits,

*item*.numberOfDeposits,

          ),

          amountOfRechargePerPerson: Math.min(

            directSubordinatesRechargeAmount,

*item*.amountOfRechargePerPerson,

          ),

        },

        bonusAmount: *item*.bonusAmount,

      };

    });

*const* claimableBonusData = invitationBonusData.filter(

      (*item*) *=>* *item*.isFinished && *item*.id === parseInt(invitationBonusId),

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reword!",

      });

    }

*const* claimedRewardsData = invitationBonusData.find(

      (*item*) *=>* *item*.isClaimed && *item*.id === parseInt(invitationBonusId),

    );

    if (claimedRewardsData?.id === parseInt(invitationBonusId)) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === parseInt(invitationBonusId),

    );

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        parseInt(invitationBonusId),

        REWARD\_TYPES\_MAP.INVITATION\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed invitation bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. Verifies the authentication token.
2. Retrieves user data from the database based on the token.
3. Calculates the number of direct subordinates and their total recharge amount.
4. Retrieves claimed rewards data for the user.
5. Calculates the invitation bonus data based on the InvitationBonusList and direct subordinates' data.
6. Checks if the bonus can be claimed based on the user's eligibility and the claim ID.
7. If the bonus can be claimed, updates the user's money in the database and inserts a new row in the claimed\_rewards table.
8. Returns a successful response if the bonus is claimed, or an error response if the bonus cannot be claimed.

Example usage:

POST /claim-invitation-bonus

Request:

{

"headers": {

"Cookie": "auth=AUTH\_TOKEN"

},

"body": {

"claim\_id": 1

}

}

Response:

{

"status": true,

"message": "Successfully claimed invitation bonus"

}

### getInvitedMembers

This function retrieves the list of members invited by the authenticated user. It requires an authentication token to identify the user in the database. The function performs the following steps:

*const* getInvitedMembers = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `code`, `invite`, `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*let* [invitedMembers] = await connection.execute(

      "SELECT `phone`, `time`, `id\_user`, `id\_user`, `name\_user` FROM `users` WHERE `invite` = ?",

      [user.code],

    );

    for (*let* index = 0; index < invitedMembers.length; index++) {

*const* invitedMember = invitedMembers[index];

*const* [[row\_2]] = await connection.execute(

        "SELECT SUM(money) AS `sum` FROM `recharge` WHERE `phone` = ? AND `status` = ?",

        [invitedMember.phone, 1],

      );

*const* rechargeAmount = row\_2.sum;

      invitedMembers[index]["rechargeAmount"] = rechargeAmount;

    }

    return *res*.status(200).json({

      data: invitedMembers.map((*invitedMember*) *=>* ({

        uid: *invitedMember*.id\_user,

        phone: *invitedMember*.phone,

        create\_time: moment(*invitedMember*.time, "x").format(

          "YYYY-MM-DD HH:mm:ss",

        ),

        amount: *invitedMember*.rechargeAmount,

        username: *invitedMember*.name\_user,

      })),

      status: true,

      message: "Successfully fetched invited members",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. Verifies the authentication token.
2. Retrieves user data from the database based on the token.
3. Retrieves the list of invited members from the database based on the user's invite code.
4. Calculates the recharge amount for each invited member.
5. Returns the list of invited members in the response.

Example usage:

GET /invited-members

Request:

{

"headers": {

"Cookie": "auth=AUTH\_TOKEN"

}

}

Response:

{

"data": [

{

"uid": 123,

"phone": "1234567890",

"create\_time": "2022-01-01 12:00:00",

"amount": 1000,

"username": "John Doe"

},

{

...

}

],

"status": true,

"message": "Successfully fetched invited members"

}

## Daily Recharge Bonus Functions

### getDailyRechargeReword

This function retrieves the daily recharge bonus data for the authenticated user. It requires an authentication token to identify the user in the database. The function performs the following steps:

*const* getDailyRechargeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [todayRechargeRow] = await connection.execute(

      "SELECT SUM(money) AS `sum` FROM `recharge` WHERE `phone` = ? AND `status` = ? AND `time` >= ?",

      [user.phone, 1, today],

    );

*const* todayRechargeAmount = todayRechargeRow[0].sum || 0;

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS, user.phone, today],

    );

*const* dailyRechargeRewordList = DailyRechargeBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        rechargeAmount: Math.min(todayRechargeAmount, *item*.rechargeAmount),

        requiredRechargeAmount: *item*.rechargeAmount,

        bonusAmount: *item*.bonusAmount,

        isFinished: todayRechargeAmount >= *item*.rechargeAmount,

        isClaimed: claimedRewardsRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

      };

    });

    return *res*.status(200).json({

      data: dailyRechargeRewordList,

      status: true,

      message: "Successfully fetched daily recharge bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

1. Verifies the authentication token.
2. Retrieves user data from the database based on the token.
3. Calculates the total recharge amount for the user for the current day.
4. Retrieves claimed rewards data for the user.
5. Calculates the daily recharge bonus data based on the DailyRechargeBonusList and the user's recharge amount.
6. Returns the daily recharge bonus data in the response.

Example usage:

GET /daily-recharge-bonus

Request:

{

"headers": {

"Cookie": "auth=AUTH\_TOKEN"

}

}

Response:

{

"data": [

{

"id": 1,

"rechargeAmount": 100,

"requiredRechargeAmount": 1000,

"bonusAmount": 38,

"isFinished": false,

"isClaimed": false

},

{

...

}

],

"status": true,

"message": "Successfully fetched daily recharge bonus data"

}

## claimDailyRechargeReword

This function is used to claim the daily recharge reward. It takes the user's authentication token and the ID of the reward as input and returns a response indicating whether the reward was successfully claimed or not.

*const* claimDailyRechargeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* dailyRechargeRewordId = *req*.body.claim\_id;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [todayRechargeRow] = await connection.execute(

      "SELECT SUM(money) AS `sum` FROM `recharge` WHERE `phone` = ? AND `status` = ? AND `time` >= ?",

      [user.phone, 1, today],

    );

*const* todayRechargeAmount = todayRechargeRow[0].sum || 0;

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS, user.phone, today],

    );

*const* dailyRechargeRewordList = DailyRechargeBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        rechargeAmount: todayRechargeAmount,

        requiredRechargeAmount: *item*.rechargeAmount,

        bonusAmount: *item*.bonusAmount,

        isFinished: todayRechargeAmount >= *item*.rechargeAmount,

        isClaimed: claimedRewardsRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.rechargeAmount,

        ),

      };

    });

*const* claimableBonusData = dailyRechargeRewordList.filter(

      (*item*) *=>*

*item*.isFinished && *item*.rechargeAmount >= *item*.requiredRechargeAmount,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reword!",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === parseInt(dailyRechargeRewordId),

    );

*const* [bonusList] = await connection.query(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ? AND `reward\_id` = ?",

      [

        REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS,

        user.phone,

        today,

        claimedBonusData.id,

      ],

    );

    if (bonusList.length > 0) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed daily recharge bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The provided code is an asynchronous function named claimDailyRechargeReward that handles the logic for claiming a daily recharge reward. Here is a detailed explanation of the code:

1. It first extracts the authorization token and daily recharge reward ID from the request.
2. Queries the database to retrieve user information based on the token.
3. Calculates the total recharge amount for the current day for the user.
4. Fetches claimed rewards data and constructs a list of daily recharge rewards with information like ID, recharge amounts, bonus amounts, etc.
5. Filters the list to find claimable bonus data based on specific conditions.
6. If no claimable bonus is found, it returns a response indicating the user does not meet the requirements.
7. Processes the claimed bonus by updating user's money, total money, and adding a record in the claimed rewards table.
8. Returns success response after claiming the daily recharge bonus.
9. If any error occurs during the process, it logs the error and returns a 500 status response with the error message.

Example usage:

POST /claim-daily-recharge-reward

Content-Type: application/json

{

"claim\_id": 1

}

## dailyRechargeRewordRecord

This function is used to fetch the daily recharge reward records for a user. It retrieves the claimed rewards data for the daily recharge bonus and returns it as a response.

*const* dailyRechargeRewordRecord = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS, user.phone],

    );

*const* claimedRewardsData = claimedRewardsRow.map((*claimedReward*) *=>* {

*const* currentDailyRechargeReword = DailyRechargeBonusList.find(

        (*item*) *=>* *item*?.id === *claimedReward*?.reward\_id,

      );

      return {

        id: *claimedReward*.reward\_id,

        requireRechargeAmount: currentDailyRechargeReword?.rechargeAmount || 0,

        amount: *claimedReward*.amount,

        status: *claimedReward*.status,

        time: moment.unix(*claimedReward*.time).format("YYYY-MM-DD HH:mm:ss"),

      };

    });

    return *res*.status(200).json({

      data: claimedRewardsData,

      status: true,

      message: "Successfully fetched daily recharge bonus record",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: "Something went wrong" });

  }

};

The provided code is an asynchronous function named dailyRechargeRewordRecord that handles fetching daily recharge bonus records. Here’s a breakdown of the code:

1. It tries to extract the authToken from the cookies of the request object.
2. Queries the database to fetch a user's phone number based on the provided token and verification status.
3. If the user is not found, it returns a 401 status with a message 'Unauthorized'.
4. Retrieves claimed rewards data from the database based on reward type and user's phone.
5. Processes the claimed rewards data, including formatting timestamps and mapping reward details.
6. Finally, returns a JSON response with the fetched data, a success status, and a message indicating a successful fetch operation.
7. If any errors occur during the process, it catches the error, logs it, and returns a 500 status with a message 'Something went wrong'.

Example usage:

GET /daily-recharge-reward-record

## getFirstRechargeRewords

This function is used to retrieve the available first recharge rewards for a user. It fetches the claimed rewards data for the first recharge bonus and returns it as a response.

*const* getFirstRechargeRewords = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.FIRST\_RECHARGE\_BONUS, user.phone],

    );

*const* [rechargeRow] = await connection.execute(

      "SELECT \* FROM `recharge` WHERE `phone` = ? AND `status` = ? ORDER BY id DESC LIMIT 1 ",

      [user.phone, 1],

    );

*const* firstRecharge = rechargeRow?.[0];

*const* firstRechargeRewordList = firstRechargeBonusList.map(

      (*item*, *index*) *=>* {

*const* currentRechargeAmount = firstRecharge?.money || 0;

        return {

          id: *item*.id,

          currentRechargeAmount: Math.min(

*item*.rechargeAmount,

            currentRechargeAmount,

          ),

          requiredRechargeAmount: *item*.rechargeAmount,

          bonusAmount: *item*.bonusAmount,

          agentBonus: *item*.agentBonus,

          isFinished:

*index* === 0

              ? currentRechargeAmount >= *item*.rechargeAmount

              : currentRechargeAmount >= *item*.rechargeAmount &&

                firstRechargeBonusList[*index* - 1]?.rechargeAmount >

                  currentRechargeAmount,

          isClaimed: claimedRewardsRow.some(

            (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

          ),

        };

      },

    );

    return *res*.status(200).json({

      data: firstRechargeRewordList,

      isExpired: firstRechargeRewordList.some(

        (*item*) *=>* *item*.isFinished && *item*.isClaimed,

      ),

      status: true,

      message: "Successfully fetched first recharge bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

This function getFirstRechargeRewards is an asynchronous function that handles retrieving first recharge bonus data. Here's a breakdown of the code:

1. It first tries to extract the authToken from the request cookies.
2. Queries the database to fetch user information based on the token. If no user is found, it returns an unauthorized response.
3. Retrieves claimed rewards and the latest recharge row for the user.
4. Calculates bonus details based on the user's current recharge amount and whether they have claimed the rewards.
5. Constructs a firstRechargeRewardList array based on the bonus details.
6. Returns a JSON response with the bonus data, expiration status, and a success message.
7. If an error occurs during the process, it logs the error and returns a 500 status with an error message.

Example usage:

GET /first-recharge-rewards

## claimFirstRechargeReword

This function is used to claim a first recharge reward for a user. It takes the user's authentication token and the ID of the reward as input and returns a response indicating whether the reward was successfully claimed or not.

*const* claimFirstRechargeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* firstRechargeRewordId = *req*.body.id;

*const* [userRow] = await connection.execute(

      "SELECT \* FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.FIRST\_RECHARGE\_BONUS, user.phone],

    );

*const* [rechargeRow] = await connection.execute(

      "SELECT \* FROM `recharge` WHERE `phone` = ? AND `status` = ? ORDER BY id DESC LIMIT 1 ",

      [user.phone, 1],

    );

*const* firstRecharge = rechargeRow?.[0];

*const* firstRechargeRewordList = firstRechargeBonusList.map(

      (*item*, *index*) *=>* {

*const* currentRechargeAmount = firstRecharge?.money || 0;

        return {

          id: *item*.id,

          currentRechargeAmount: Math.min(

*item*.rechargeAmount,

            currentRechargeAmount,

          ),

          requiredRechargeAmount: *item*.rechargeAmount,

          bonusAmount: *item*.bonusAmount,

          agentBonus: *item*.agentBonus,

          isFinished:

*index* === 0

              ? currentRechargeAmount >= *item*.rechargeAmount

              : currentRechargeAmount >= *item*.rechargeAmount &&

                firstRechargeBonusList[*index* - 1]?.rechargeAmount >

                  currentRechargeAmount,

          isClaimed: claimedRewardsRow.some(

            (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

          ),

        };

      },

    );

*const* claimableBonusData = firstRechargeRewordList.filter(

      (*item*) *=>* *item*.isFinished,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reword!",

      });

    }

*const* isExpired = firstRechargeRewordList.some(

      (*item*) *=>* *item*.isFinished && *item*.isClaimed,

    );

    if (isExpired) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === firstRechargeRewordId,

    );

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.FIRST\_RECHARGE\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed first recharge bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The provided code defines an asynchronous function named claimFirstRechargeReword that processes a request to claim the first recharge bonus reward. Here is a breakdown of the code:

1. It retrieves the authentication token and the firstRechargeRewordId from the request object.
2. It checks if the user associated with the token exists and returns an error if not.
3. It fetches the claimed rewards and recharge data for the user.
4. It processes the bonus rewards based on certain conditions like current recharge amount, required recharge amount, bonus amounts, etc.
5. It checks for claimable bonus data and handles scenarios where the bonus can be claimed or is expired.
6. It updates the user's money and total money based on the claimed bonus data and inserts a record in the claimed\_rewards table.
7. Finally, it returns a success response if the bonus is claimed successfully or an error response if any exception occurs.

Example usage:

POST /claim-first-recharge-reward

Content-Type: application/json

{

"id": 1

}

## getAttendanceBonus

This function is used to fetch the attendance bonus data for a user. It retrieves the claimed attendance bonus data and returns it as a response.

*const* getAttendanceBonus = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.ATTENDANCE\_BONUS, user.phone],

    );

*let* attendanceBonusId = 0;

    if (claimedRewardsRow.length === 0) {

      attendanceBonusId = 0;

    } else {

*const* lastClaimedReword =

        claimedRewardsRow?.[claimedRewardsRow.length - 1];

*const* lastClaimedRewordTime = lastClaimedReword?.time || 0;

*let* clamed\_date = new *Date*(timerJoin(lastClaimedRewordTime));

*let* todays\_date = new *Date*(timerJoin(moment().valueOf()));

*let* Difference\_In\_Time =  todays\_date.getTime() - clamed\_date.getTime();

*let* Difference\_In\_Days =  Math.round(Difference\_In\_Time / (1000 \* 3600 \* 24));

      if (parseInt(Difference\_In\_Days)  < 1) {

        attendanceBonusId = lastClaimedReword.reward\_id;

      } else if (parseInt(Difference\_In\_Days) >= 2) {

        attendanceBonusId = 0;

      } else {

        attendanceBonusId = lastClaimedReword.reward\_id;

      }

    }

*const* claimedBonusData = AttendanceBonusList.find(

      (*item*) *=>* *item*.id === attendanceBonusId,

    );

    return *res*.status(200).json({

      status: true,

      data: {

        id: claimedBonusData?.id || 0,

        days: claimedBonusData?.days || 0,

        bonusAmount: claimedBonusData?.bonusAmount || 0,

      },

      message: "Successfully fetched attendance bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({

      status: true,

      message: error.message,

    });

  }

};

The provided code defines an asynchronous function getAttendanceBonus that handles the logic for fetching attendance bonus data. Here's a breakdown of the code:

1. We first attempt to retrieve the authorization token from the request cookies.
2. We then query the database to fetch the user's phone number based on the token and verification status.
3. If the user is not found, it returns a 401 status with an 'Unauthorized' message.
4. The function then queries the database to check for claimed rewards related to attendance bonus for the user.
5. Based on the claimed rewards data, it calculates the attendance bonus ID considering time differences.
6. The function looks for the corresponding attendance bonus data in a list (presumably AttendanceBonusList).
7. Finally, it returns a JSON response with status, bonus data (if found), and a success message. In case of an error, it logs the error and returns a 500 status with the error message.

Example usage:

GET /attendance-bonus

## claimAttendanceBonus

This function is used to claim an attendance bonus for a user. It takes the user's authentication token as input and returns a response indicating whether the bonus was successfully claimed or not.

*const* claimAttendanceBonus = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.ATTENDANCE\_BONUS, user.phone],

    );

    if (claimedRewardsRow.map((*item*) *=>* *item*.reward\_id).includes(7)) {

      return *res*.status(200).json({

        status: false,

        message: "You have already claimed the attendance bonus for 7 days",

      });

    }

*let* attendanceBonusId = 0;

    if (claimedRewardsRow.length === 0) {

      attendanceBonusId = 1;

    } else {

*const* lastClaimedReword =

        claimedRewardsRow?.[claimedRewardsRow.length - 1];

*const* lastClaimedRewordTime = lastClaimedReword?.time || 0;

*let* clamed\_date = new *Date*(timerJoin(lastClaimedRewordTime));

*let* todays\_date = new *Date*(timerJoin(moment().valueOf()));

*let* Difference\_In\_Time =  todays\_date.getTime() - clamed\_date.getTime();

*let* Difference\_In\_Days =  Math.round(Difference\_In\_Time / (1000 \* 3600 \* 24));

      if (parseInt(Difference\_In\_Days) < 1) {

        return *res*.status(200).json({

          status: false,

          message: "You have already claimed the attendance bonus today",

        });

      } else if (parseInt(Difference\_In\_Days) >= 2) {

        attendanceBonusId = 1;

      } else {

        attendanceBonusId = lastClaimedReword.reward\_id + 1;

      }

    }

*const* claimedBonusData = AttendanceBonusList.find(

      (*item*) *=>* *item*.id === attendanceBonusId,

    );

*const* [rechargeTotal] = await connection.query(

      "SELECT SUM(money) AS total\_recharge FROM recharge WHERE status = 1 AND phone = ?",

      [user.phone],

    );

*const* totalRecharge = +rechargeTotal[0].total\_recharge || 0;

*const* check = totalRecharge >= claimedBonusData.requiredAmount;

    if (!check)

      return *res*.status(200).json({

        status: false,

        message: "Total Recharge amount doesn't met the Required Amount !",

      });

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.ATTENDANCE\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: `Successfully claimed attendance bonus for ${getOrdinal(claimedBonusData.days)} day`,

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({

      status: true,

      message: error.message,

    });

  }

};

In the provided code, the claimAttendanceBonus function is an asynchronous function used to handle the process of claiming attendance bonuses for users based on certain conditions. Here is a breakdown of the code:

1. The function first tries to fetch the authToken from the request cookies and retrieves the user information based on the token from the database.
2. It checks if the user exists and is authorized. If not, it returns a response with status 401 indicating unauthorized.
3. It then checks if the user has already claimed the attendance bonus for 7 days. If claimed, it returns a response stating the user has already claimed the bonus.
4. The function calculates the attendance bonus id based on the last claimed reward and the time difference between claims.
5. It then verifies if the total recharge amount meets the required amount for the claimed bonus. If not met, it returns a response indicating the issue.
6. If all conditions are met, it updates the user's money in the database, adds the claimed reward entry, and returns a success response with the claimed bonus details.
7. Any errors that occur during the process are caught in the catch block and a corresponding error response is sent.

Example usage:

POST /claim-attendance-bonus

## getAttendanceBonusRecord

This function is used to fetch the attendance bonus records for a user. It retrieves the claimed attendance bonus data and returns it as a response.

*const* getAttendanceBonusRecord = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.ATTENDANCE\_BONUS, user.phone],

    );

*const* claimedRewardsData = claimedRewardsRow.map((*claimedReward*) *=>* {

*const* currentAttendanceBonus = AttendanceBonusList.find(

        (*item*) *=>* *item*?.id === *claimedReward*?.reward\_id,

      );

      return {

        id: *claimedReward*.reward\_id,

        days: currentAttendanceBonus?.days || 0,

        amount: *claimedReward*.amount,

        status: *claimedReward*.status,

        time: moment.unix(*claimedReward*.time).format("YYYY-MM-DD HH:mm:ss"),

      };

    });

    return *res*.status(200).json({

      data: claimedRewardsData,

      status: true,

      message: "Successfully fetched attendance bonus record",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({

      status: true,

      message: error.message,

    });

  }

};

The provided code is an asynchronous function named getAttendanceBonusRecord that fetches attendance bonus records based on the user's authentication token. Here's a breakdown of the code:

1. It first tries to extract the authentication token from the request cookies.
2. Queries the database to retrieve the user's phone number based on the token and verification status.
3. If the user is not found, it returns a 401 Unauthorized response.
4. Queries the database to fetch claimed rewards data related to attendance bonus for the user.
5. Maps the claimed rewards data to a new format with additional information like days, amount, status, and formatted time.
6. Returns the formatted claimed rewards data along with a success message if everything is successful.
7. If any error occurs during the process, it logs the error and returns a 500 Internal Server Error response with the error message.

Example usage:

GET /attendance-bonus-record

## getDailyBettingeReword

This function is used to fetch the available daily betting rewards for a user. It retrieves the claimed daily betting bonus data and returns it as a response.

*const* getDailyBettingeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [minutes\_1] = await connection.query('SELECT \* FROM minutes\_1 WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*const* [k3\_bet\_money] = await connection.query('SELECT \* FROM result\_k3 WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*const* [d5\_bet\_money] = await connection.query('SELECT \* FROM result\_5d WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*const* [trx\_bet\_money] = await connection.query('SELECT \* FROM trx\_wingo\_bets WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

  minutes\_1.forEach((*data*) *=>* {

        total\_w += parseFloat(*data*.money);

    });

    k3\_bet\_money.forEach((*data*) *=>* {

        total\_k3 += parseFloat(data.money);

    });

    d5\_bet\_money.forEach((*data*) *=>* {

        total\_5d += parseFloat(*data*.money);

    });

    trx\_bet\_money.forEach((*data*) *=>* {

        total\_trx += parseFloat(*data*.money);

    });

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d) + parseInt(total\_trx);

    console.log(total2);

*const* todayBettingAmount = total2;

*const* [claimedBettingRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.DAILY\_BETTING\_BONUS, user.phone, today],

    );

*const* dailyBetingRewordList = DailyBettingBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        bettingAmount: Math.min(todayBettingAmount, *item*.bettingAmount),

        requiredBettingAmount: *item*.bettingAmount,

        bonusAmount: *item*.bonusAmount,

        isFinished: todayBettingAmount >= *item*.bettingAmount,

        isClaimed: claimedBettingRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

      };

    });

    return *res*.status(200).json({

      data: dailyBetingRewordList,

      status: true,

      message: "Successfully fetched daily recharge bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

This code defines an asynchronous function getDailyBettingeReword that takes req and res as parameters. It tries to fetch user information based on an authentication token from a database, calculates the total betting amounts for different categories, and processes the daily recharge bonus data.

Explanation:

1. It retrieves the authToken from req.cookies and queries the database to get the user's phone number.
2. If the user is not found, it returns an unauthorized response.
3. Calculates the start of the current day and queries various tables to get betting amounts for different categories.
4. Calculates the total betting amounts for different categories and stores it in todayBettingAmount.
5. Fetches claimed reward information from the database based on reward type and user phone number for the current day.
6. Maps the DailyBettingBonusList to create a list of daily betting rewards with details like betting amount, bonus amount, and claim status.
7. Returns the list of daily betting rewards along with a success message if all operations are successful. Otherwise, logs the error and returns a server error response.

Example usage:

GET /daily-betting-rewards

## getDailyRebateReword

This function is used to fetch the available daily rebate rewards for a user. It retrieves the claimed daily rebate bonus data and returns it as a response.

*const* getDailyRebateReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [commissions] = await connection.query('SELECT SUM(`money`) as `sum` FROM commissions WHERE phone = ? AND `time` >= ? AND `level` <= 6; ', [user.phone, today]);

*let* comm\_amt = 0;

    comm\_amt = commissions[0].sum || 0;

*const* todayRebateAmount = comm\_amt;

*const* [week\_commissions] = await connection.query('SELECT SUM(`money`) as `sum` FROM commissions WHERE phone = ? AND `level` <= 6;', [user.phone]);

*let* w\_comm\_amt = 0;

    w\_comm\_amt = week\_commissions[0].sum || 0;

*const* week\_RebateAmount = w\_comm\_amt;

*const* [claimedBettingRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.REBATE\_BONUS, user.phone, today],

    );

*const* dailyRebateRewordList = DailyRebateBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        bettingAmount: Math.min(todayRebateAmount, *item*.rebetAmount),

        requiredBettingAmount: *item*.rebetAmount,

        bonusAmount: todayRebateAmount,

        weeklyBetAmount:week\_RebateAmount,

        isFinished: todayRebateAmount >= *item*.rebetAmount,

        isClaimed: claimedBettingRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

      };

    });

    return *res*.status(200).json({

      data: dailyRebateRewordList,

      status: true,

      message: "Successfully fetched daily recharge bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The provided code is an asynchronous function named getDailyRebateReword. It fetches daily rebate reward-related data based on certain conditions.

Here is a breakdown of the code:

1. It first tries to fetch the authToken from the req.cookies.
2. Queries the database to get specific user data based on the authToken.
3. Calculates today's and weekly rebate amounts by querying the database for commissions.
4. Checks for claimed betting rewards based on certain conditions.
5. Constructs a list of daily rebate rewards combining various data points.
6. Returns the constructed data with a success status if everything runs without errors.
7. If any error occurs during the process, it logs the error and returns a response with the error message and a status of 500.

Example usage:

GET /daily-rebate-rewards

## claimDailyBettingReword

This function is used to claim a daily betting reward for a user. It takes the user's authentication token and the ID of the reward as input and returns a response indicating whether the reward was successfully claimed or not.

*const* claimDailyBettingReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

*const* dailyBettingRewordId = *req*.body.claim\_id;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [minutes\_1] = await connection.query('SELECT \* FROM minutes\_1 WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*const* [k3\_bet\_money] = await connection.query('SELECT \* FROM result\_k3 WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*const* [d5\_bet\_money] = await connection.query('SELECT \* FROM result\_5d WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*const* [trx\_bet\_money] = await connection.query('SELECT \* FROM trx\_wingo\_bets WHERE phone = ? AND `time` >= ?', [user.phone, today]);

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

    minutes\_1.forEach((*data*) *=>* {

        total\_w += parseFloat(*data*.money);

    });

    k3\_bet\_money.forEach((data) *=>* {

        total\_k3 += parseFloat(data.money);

    });

    d5\_bet\_money.forEach((data) *=>* {

        total\_5d += parseFloat(data.money);

    });

    trx\_bet\_money.forEach((data) *=>* {

        total\_trx += parseFloat(data.money);

    });

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d) + parseInt(total\_trx);

*const* todayBettingAmount = total2;

*const* [claimedBettingRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.DAILY\_BETTING\_BONUS, user.phone, today],

    );

*const* dailyBetingRewordList = DailyBettingBonusList.map((item) *=>* {

      return {

        id: item.id,

        bettingAmount: todayBettingAmount,

        requiredBettingAmount: item.bettingAmount,

        bonusAmount: item.bonusAmount,

        isFinished: todayBettingAmount >= *item*.bettingAmount,

        isClaimed: claimedBettingRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.rechargeAmount,

        ),

      };

    });

*const* claimableBonusData = dailyBetingRewordList.filter(

      (*item*) *=>*

*item*.isFinished && *item*.bettingAmount >= *item*.requiredBettingAmount,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reward!",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === parseInt(dailyBettingRewordId),

    );

*const* [bonusList] = await connection.query(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ? AND `reward\_id` = ?",

      [

        REWARD\_TYPES\_MAP.DAILY\_BETTING\_BONUS,

        user.phone,

        today,

        claimedBonusData.id,

      ],

    );

    if (bonusList.length > 0) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.DAILY\_BETTING\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed daily betting bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

This code defines an asynchronous function claimDailyBettingReword that takes two parameters req and res, which are typically the request and response objects in a Node.js application.

1. The function attempts to claim a daily betting reward by performing various database queries based on the user's phone number and authorization token.
2. If the user is not authorized, it returns a 401 Unauthorized response with a message.
3. The code calculates the total betting amount for the current day by querying different tables and summing up the betting amounts.
4. It then checks if the user meets the requirements to claim the daily betting bonus by comparing the betting amount with the required amount.
5. If the user is eligible for the bonus, it updates the user's money balance, inserts a record into the claimed\_rewards table, and returns a success response.
6. If any error occurs during the process, it logs the error and returns a 500 Internal Server Error response with an error message.

Example usage:

POST /claim-daily-betting-reward

Content-Type: application/json

{

"claim\_id": 1

}

## claimDailyRechargeReword

This function is used to claim the daily recharge reward. It takes the user's authentication token and the ID of the reward as input and returns a response indicating whether the reward was successfully claimed or not.

*const* claimDailyRechargeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* dailyRechargeRewordId = *req*.body.claim\_id;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [todayRechargeRow] = await connection.execute(

      "SELECT SUM(money) AS `sum` FROM `recharge` WHERE `phone` = ? AND `status` = ? AND `time` >= ?",

      [user.phone, 1, today],

    );

*const* todayRechargeAmount = todayRechargeRow[0].sum || 0;

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS, user.phone, today],

    );

*const* dailyRechargeRewordList = DailyRechargeBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        rechargeAmount: todayRechargeAmount,

        requiredRechargeAmount: *item*.rechargeAmount,

        bonusAmount: *item*.bonusAmount,

        isFinished: todayRechargeAmount >= *item*.rechargeAmount,

        isClaimed: claimedRewardsRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.rechargeAmount,

        ),

      };

    });

*const* claimableBonusData = dailyRechargeRewordList.filter(

      (*item*) *=>*

*item*.isFinished && *item*.rechargeAmount >= *item*.requiredRechargeAmount,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reword!",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === parseInt(dailyRechargeRewordId),

    );

*const* [bonusList] = await connection.query(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ? AND `reward\_id` = ?",

      [

        REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS,

        user.phone,

        today,

        claimedBonusData.id,

      ],

    );

    if (bonusList.length > 0) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed daily recharge bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The provided code is an asynchronous function named claimDailyRebateReword that handles claiming daily rebate rewards.

1. It first extracts authToken and dailyRebateRewordId from the request object. It then performs a database query to retrieve the user's phone number based on the provided token.
2. If the user is not found (i.e., unauthorized), it returns a 401 status with a message 'Unauthorized'. It calculates the start of the current day using moment() library, fetches commissions for the user, determines the daily rebate amount, and looks up claimed betting rewards for the day.
3. The function then constructs a list of daily rebate rewards, filters out claimable bonuses, checks if the specified bonus can be claimed, and updates the user's balance accordingly. Finally, it inserts a new record in the claimed rewards table and returns a success message if the process is completed without errors.
4. The provided code is well-structured and handles potential errors by catching exceptions using a try-catch block. It also logs any errors encountered during the process for debugging purposes.

Example usage:

POST /claim-daily-recharge-reward

Content-Type: application/json

{

"claim\_id": 1

}

## dailyRechargeRewordRecord

This function is used to fetch the daily recharge reward records for a user. It retrieves the claimed rewards data for the daily recharge bonus and returns it as a response.

*const* dailyRechargeRewordRecord = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.DAILY\_RECHARGE\_BONUS, user.phone],

    );

*const* claimedRewardsData = claimedRewardsRow.map((*claimedReward*) *=>* {

*const* currentDailyRechargeReword = DailyRechargeBonusList.find(

        (*item*) *=>* *item*?.id === *claimedReward*?.reward\_id,

      );

      return {

        id: *claimedReward*.reward\_id,

        requireRechargeAmount: currentDailyRechargeReword?.rechargeAmount || 0,

        amount: *claimedReward*.amount,

        status: *claimedReward*.status,

        time: moment.unix(*claimedReward*.time).format("YYYY-MM-DD HH:mm:ss"),

      };

    });

    return *res*.status(200).json({

      data: claimedRewardsData,

      status: true,

      message: "Successfully fetched daily recharge bonus record",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: "Something went wrong" });

  }

};

The provided code defines an asynchronous function dailyBettingRewardRecord that retrieves a user's daily betting reward record.

Here is an explanation of the code:

1. It first tries to extract the authToken from the request cookies.
2. It then queries the database to fetch the user's information based on the token, verifying the user.
3. If the user is not found, it responds with a 401 status and an 'Unauthorized' message.
4. It then retrieves claimed rewards data from the database, filtering by reward type and user's phone number.
5. A mapping function is applied to the claimed rewards data to format and structure the response data.
6. The formatted data includes the reward id, required betting amount, claimed amount, status, and time formatted as per specified format.
7. Finally, it responds with a 200 status along with the retrieved data, a success status, and a message indicating the successful fetch of daily recharge bonus records.
8. If any error occurs during the process, it logs the error and responds with a 500 status and a 'Something went wrong' message.

Example usage:

GET /daily-recharge-reward-record

## getFirstRechargeRewords

This function is used to retrieve the available first recharge rewards for a user. It fetches the claimed rewards data for the first recharge bonus and returns it as a response.

*const* getFirstRechargeRewords = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.FIRST\_RECHARGE\_BONUS, user.phone],

    );

*const* [rechargeRow] = await connection.execute(

      "SELECT \* FROM `recharge` WHERE `phone` = ? AND `status` = ? ORDER BY id DESC LIMIT 1 ",

      [user.phone, 1],

    );

*const* firstRecharge = rechargeRow?.[0];

*const* firstRechargeRewordList = firstRechargeBonusList.map(

      (*item*, *index*) *=>* {

*const* currentRechargeAmount = firstRecharge?.money || 0;

        return {

          id: *item*.id,

          currentRechargeAmount: Math.min(

*item*.rechargeAmount,

            currentRechargeAmount,

          ),

          requiredRechargeAmount: *item*.rechargeAmount,

          bonusAmount: *item*.bonusAmount,

          agentBonus: *item*.agentBonus,

          isFinished:

*index* === 0

              ? currentRechargeAmount >= *item*.rechargeAmount

              : currentRechargeAmount >= *item*.rechargeAmount &&

                firstRechargeBonusList[*index* - 1]?.rechargeAmount >

                  currentRechargeAmount,

          isClaimed: claimedRewardsRow.some(

            (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

          ),

        };

      },

    );

    return *res*.status(200).json({

      data: firstRechargeRewordList,

      isExpired: firstRechargeRewordList.some(

        (*item*) *=>* *item*.isFinished && *item*.isClaimed,

      ),

      status: true,

      message: "Successfully fetched first recharge bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

This code defines an asynchronous function getFirstRechargeRewards that fetches data related to first recharge rewards based on the user's authentication token and phone number. Here's a breakdown of the code:

1. It retrieves the user's phone number from the database using the authentication token.
2. If the user is not found, it returns a 401 status with an 'Unauthorized' message.
3. It then retrieves claimed rewards and recharge details for the user.
4. The function calculates various reward-related information based on the current recharge amount and bonus details.
5. Finally, it constructs a response object containing the reward data, expiration status, and a success message.
6. The code is wrapped in a try-catch block to handle any errors that may occur during database operations. If an error occurs, it logs the error and returns a 500 status with an error message.

Example usage:

GET /first-recharge-rewards

## claimFirstRechargeReword

This function is used to claim a first recharge reward for a user. It takes the user's authentication token and the ID of the reward as input and returns a response indicating whether the reward was successfully claimed or not.

*const* claimFirstRechargeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* firstRechargeRewordId = *req*.body.id;

*const* [userRow] = await connection.execute(

      "SELECT \* FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.FIRST\_RECHARGE\_BONUS, user.phone],

    );

*const* [rechargeRow] = await connection.execute(

      "SELECT \* FROM `recharge` WHERE `phone` = ? AND `status` = ? ORDER BY id DESC LIMIT 1 ",

      [user.phone, 1],

    );

*const* firstRecharge = rechargeRow?.[0];

*const* firstRechargeRewordList = firstRechargeBonusList.map(

      (*item*, *index*) *=>* {

*const* currentRechargeAmount = firstRecharge?.money || 0;

        return {

          id: *item*.id,

          currentRechargeAmount: Math.min(

*item*.rechargeAmount,

            currentRechargeAmount,

          ),

          requiredRechargeAmount: *item*.rechargeAmount,

          bonusAmount: *item*.bonusAmount,

          agentBonus: *item*.agentBonus,

          isFinished:

*index* === 0

              ? currentRechargeAmount >= *item*.rechargeAmount

              : currentRechargeAmount >= *item*.rechargeAmount &&

                firstRechargeBonusList[*index* - 1]?.rechargeAmount >

                  currentRechargeAmount,

          isClaimed: claimedRewardsRow.some(

            (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

          ),

        };

      },

    );

*const* claimableBonusData = firstRechargeRewordList.filter(

      (*item*) *=>* *item*.isFinished,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reword!",

      });

    }

*const* isExpired = firstRechargeRewordList.some(

      (*item*) *=>* *item*.isFinished && *item*.isClaimed,

    );

    if (isExpired) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === firstRechargeRewordId,

    );

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.FIRST\_RECHARGE\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed first recharge bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

In this code, an asynchronous function claimFirstRechargeReword is defined with req and res parameters for handling requests and responses.

1. It tries to claim the first recharge reward based on certain conditions.
2. The function first retrieves the authToken and firstRechargeRewordId from the request objects. It then queries the database using the connection.execute method to fetch user information and check if the user is authorized. If not authorized, it returns a 401 status with an 'Unauthorized' message.
3. Subsequent queries retrieve claimed rewards and recharge information from the database. The code then processes the first recharge bonus list to determine claimable bonuses based on certain conditions.
4. If there are claimable bonuses and they are not already claimed or expired, the function updates the user's money and total money in the database and adds a new entry to the claimed\_rewards table. It then returns a success status with a corresponding message.
5. If any errors occur during the process, it catches the error, logs it, and returns a 500 status with the error message in the response.

Example usage:

POST /claim-first-recharge-reward

Content-Type: application/json

{

"id": 1

}

## Function: claimDailyRebateReword

*const* claimDailyRebateReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

*const* dailyRebateRewordId = *req*.body.claim\_id;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [commissions] = await connection.query('SELECT SUM(`money`) as `sum` FROM commissions WHERE phone = ? AND `time` >= ? AND `level` <= 6;', [user.phone, today]);

*let* comm\_amt = 0;

    comm\_amt = commissions[0].sum || 0;

*const* todayRebateAmount = comm\_amt;

*const* [claimedBettingRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.REBATE\_BONUS, user.phone, today],

    );

*const* dailyRebateRewordList = DailyRebateBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        bettingAmount: todayRebateAmount,

        requiredBettingAmount: *item*.rebetAmount,

        bonusAmount: todayRebateAmount,

        isFinished: todayRebateAmount >= *item*.rebetAmount,

        isClaimed: claimedBettingRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

      };

    });

*const* claimableBonusData = dailyRebateRewordList.filter(

      (*item*) *=>*

*item*.isFinished && *item*.bettingAmount >= *item*.requiredBettingAmount,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reward!",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === parseInt(dailyRebateRewordId),

    );

*const* [bonusList] = await connection.query(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ? AND `reward\_id` = ?",

      [

        REWARD\_TYPES\_MAP.REBATE\_BONUS,

        user.phone,

        today,

        claimedBonusData.id,

      ],

    );

    if (bonusList.length > 0) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.REBATE\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed daily betting bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The claimDailyRebateReword function is responsible for allowing a user to claim their daily rebate reward. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies and the daily rebate reward ID from the request body.
2. It then retrieves the user's phone number from the database.
3. If the user does not exist, the function returns an Unauthorized status.
4. The function calculates today's start timestamp using the moment() library.
5. It retrieves the total commission earned by the user today.
6. The function generates a list of available daily rebate rewards based on the total commission earned and the required betting amount for each reward.
7. It filters the claimable bonus data based on the requirements.
8. If no claimable bonus data is found, the function returns an appropriate response.
9. The function retrieves the claimed bonus data based on the provided ID.
10. It checks if the bonus for the provided ID has already been claimed. If so, the function returns an appropriate response.
11. The function updates the user's money and total money in the database.
12. It inserts the claimed reward into the database.
13. Finally, the function returns a success response if the reward is successfully claimed.

## Function: dailyBetttingRewordRecord

*const* dailyBetttingRewordRecord = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.DAILY\_BETTING\_BONUS, user.phone],

    );

*const* claimedRewardsData = claimedRewardsRow.map((*claimedReward*) *=>* {

*const* currentDailyBetttingReword = DailyBettingBonusList.find(

        (*item*) *=>* *item*?.id === *claimedReward*?.reward\_id,

      );

      return {

        id: *claimedReward*.reward\_id,

        requireBettingAmount: currentDailyBetttingReword?.bettingAmount || 0,

        amount: *claimedReward*.amount,

        status: *claimedReward*.status,

        time: moment.unix(*claimedReward*.time).format("YYYY-MM-DD HH:mm:ss"),

      };

    });

    return *res*.status(200).json({

      data: claimedRewardsData,

      status: true,

      message: "Successfully fetched daily recharge bonus record",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: "Something went wrong" });

  }

};

The dailyBetttingRewordRecord function is responsible for retrieving the daily betting bonus records for a user. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies.
2. It retrieves the user's phone number from the database.
3. If the user does not exist, the function returns an Unauthorized status.
4. The function retrieves the claimed rewards data for the user based on the reward type.
5. It generates the claimed rewards data with the required information, such as reward ID, required betting amount, claimed amount, status, and time.
6. Finally, the function returns the claimed rewards data as a response.

## Function: dailyRebateRewordRecord

*const* dailyRebateRewordRecord = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.REBATE\_BONUS, user.phone],

    );

*const* claimedRewardsData = claimedRewardsRow.map((*claimedReward*) *=>* {

*const* currentDailyBetttingReword = DailyRebateBonusList.find(

        (*item*) *=>* *item*?.id === *claimedReward*?.reward\_id,

      );

      return {

        id: *claimedReward*.reward\_id,

        requireBettingAmount: currentDailyBetttingReword?.rebetAmount || 0,

        amount: *claimedReward*.amount,

        status: *claimedReward*.status,

        time: moment.unix(*claimedReward*.time).format("YYYY-MM-DD HH:mm:ss"),

      };

    });

    return *res*.status(200).json({

      data: claimedRewardsData,

      status: true,

      message: "Successfully fetched daily recharge bonus record",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: "Something went wrong" });

  }

};

The dailyRebateRewordRecord function is responsible for retrieving the daily rebate bonus records for a user. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies.
2. It retrieves the user's phone number from the database.
3. If the user does not exist, the function returns an Unauthorized status.
4. The function retrieves the claimed rewards data for the user based on the reward type.
5. It generates the claimed rewards data with the required information, such as reward ID, required betting amount, claimed amount, status, and time.
6. Finally, the function returns the claimed rewards data as a response.

## Function: getweeklyBettingeReword

*const* getweeklyBettingeReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [claimedBettingRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? ",

      [REWARD\_TYPES\_MAP.WEEKLY\_BETTING\_BONUS, user.phone],

    );

*const* lastClaimedReword = claimedBettingRow?.[claimedBettingRow.length - 1];

*const* lastClaimedRewordTime = lastClaimedReword?.time || 0;

*let* clamed\_date = new *Date*(timerJoin(lastClaimedRewordTime));

*let* todays\_date = new *Date*(timerJoin(moment().valueOf()));

*let* Difference\_In\_Time =  todays\_date.getTime() - clamed\_date.getTime();

*let* Difference\_In\_Days =  Math.round(Difference\_In\_Time / (1000 \* 3600 \* 24));

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

    if (parseInt(Difference\_In\_Days) > 0) {

*const* [curr\_minutes\_1] = await connection.query("SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

*const* [curr\_k3\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_k3 WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

*const* [curr\_d5\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_5d WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

*const* [curr\_trx\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

        total\_w = curr\_minutes\_1[0].sum || 0;

        total\_k3 = curr\_k3\_bet\_money[0].sum || 0;

        total\_5d = curr\_d5\_bet\_money[0].sum || 0;

        total\_trx = curr\_trx\_bet\_money[0].sum || 0;

    }

    else{

*const* [last\_minutes\_1] = await connection.query("SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

*const* [last\_k3\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_k3 WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

*const* [last\_d5\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_5d WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

*const* [last\_trx\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

        total\_w = last\_minutes\_1[0].sum || 0;

        total\_k3 = last\_k3\_bet\_money[0].sum || 0;

        total\_5d = last\_d5\_bet\_money[0].sum || 0;

        total\_trx = last\_trx\_bet\_money[0].sum || 0;

    }

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d) + parseInt(total\_trx);

*const* weekBettingAmount = total2;

*const* weekBetingRewordList = WeekBettingBonusList.map((*item*) *=>* {

      return {

        id: *item*.id,

        bettingAmount: Math.min(weekBettingAmount, *item*.bettingAmount),

        requiredBettingAmount: *item*.bettingAmount,

        bonusAmount: *item*.bonusAmount,

        isFinished: weekBettingAmount >= *item*.bettingAmount,

        isClaimed: claimedBettingRow.some(

          (*claimedReward*) *=>* *claimedReward*.reward\_id === *item*.id,

        ),

      };

    });

    return *res*.status(200).json({

      data: weekBetingRewordList,

      status: true,

      message: "Successfully fetched daily betting bonus data",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The getweeklyBettingeReword function is responsible for retrieving the weekly betting bonus data for a user. It takes in the req and res objects as parameters.

* The function retrieves the authentication token from the request cookies.
* It retrieves the user's phone number from the database.
* If the user does not exist, the function returns an Unauthorized status.
* The function calculates today's start timestamp using the moment() library.
* It retrieves the claimed betting rewards for the user.
* The function calculates the time difference between today and the last claimed reward.
* Variables are initialized for calculating different betting amounts.
* Based on the time difference, the function calculates the betting amounts for different types of bets.
* It calculates the total betting amount as the summed value of the different betting amounts.
* Based on the week's betting amount, the function generates a list of available weekly betting rewards with their required betting amounts, bonus amounts, and claim statuses.
* Finally, the function returns the weekly betting rewards as a response.

## Function: claimWeeklyBettingReword

*const* claimWeeklyBettingReword = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;;

    //const authToken = "5O3Bzs-TIZvFz5wfy7CG47mf7iZ-rXW6oS7XJxMpcE4";

*const* weeklyBettingRewordId = *req*.body.claim\_id;

    //const weeklyBettingRewordId = 1;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* today = moment().startOf("day").valueOf();

*const* [claimedBettingRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ?",

      [REWARD\_TYPES\_MAP.WEEKLY\_BETTING\_BONUS, user.phone, today],

    );

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

    if (claimedBettingRow.length > 0) {

*const* [curr\_minutes\_1] = await connection.query("SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

*const* [curr\_k3\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_k3 WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

*const* [curr\_d5\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_5d WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

*const* [curr\_trx\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE phone = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1);", [user.phone]);

        total\_w = curr\_minutes\_1[0].sum || 0;

        total\_k3 = curr\_k3\_bet\_money[0].sum || 0;

        total\_5d = curr\_d5\_bet\_money[0].sum || 0;

        total\_trx = curr\_trx\_bet\_money[0].sum || 0;

    }

    else{

*const* [last\_minutes\_1] = await connection.query("SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

*const* [last\_k3\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_k3 WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

*const* [last\_d5\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_5d WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

*const* [last\_trx\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE phone = ? AND `today` > current\_date - interval '7' day;", [user.phone]);

        total\_w = last\_minutes\_1[0].sum || 0;

        total\_k3 = last\_k3\_bet\_money[0].sum || 0;

        total\_5d = last\_d5\_bet\_money[0].sum || 0;

        total\_trx = last\_trx\_bet\_money[0].sum || 0;

    }

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d) + parseInt(total\_trx);

*const* weekBettingAmount = total2;

*const* weekBetingRewordList = WeekBettingBonusList.map((item) *=>* {

      return {

        id: item.id,

        bettingAmount: *Math*.min(weekBettingAmount, item.bettingAmount),

        requiredBettingAmount: item.bettingAmount,

        bonusAmount: item.bonusAmount,

        isFinished: weekBettingAmount >= item.bettingAmount,

        isClaimed: claimedBettingRow.some(

          (claimedReward) *=>* claimedReward.reward\_id === item.id,

        ),

      };

    });

*const* claimableBonusData = weekBetingRewordList.filter(

      (*item*) *=>*

*item*.isFinished && *item*.bettingAmount >= *item*.requiredBettingAmount,

    );

    if (claimableBonusData.length === 0) {

      return *res*.status(200).json({

        status: false,

        message: "You does not meet the requirements to claim this reward!",

      });

    }

*const* claimedBonusData = claimableBonusData?.find(

      (*item*) *=>* *item*.id === parseInt(weeklyBettingRewordId),

    );

*const* [bonusList] = await connection.query(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ? AND `time` >= ? AND `reward\_id` = ?",

      [

        REWARD\_TYPES\_MAP.WEEKLY\_BETTING\_BONUS,

        user.phone,

        today,

        claimedBonusData.id,

      ],

    );

    if (bonusList.length > 0) {

      return *res*.status(200).json({

        status: false,

        message: "Bonus already claimed",

      });

    }

*const* time = moment().valueOf();

    await connection.execute(

      "UPDATE `users` SET `money` = `money` + ?, `total\_money` = `total\_money` + ? WHERE `phone` = ?",

      [claimedBonusData.bonusAmount, claimedBonusData.bonusAmount, user.phone],

    );

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`) VALUES (?, ?, ?, ?, ?, ?)",

      [

        claimedBonusData.id,

        REWARD\_TYPES\_MAP.WEEKLY\_BETTING\_BONUS,

        user.phone,

        claimedBonusData.bonusAmount,

        REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time,

      ],

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully claimed weekly betting bonus",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: error.message });

  }

};

The claimWeeklyBettingReword function is responsible for allowing a user to claim their weekly betting bonus. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies and the weekly betting reward ID from the request body.
2. It retrieves the user's phone number from the database.
3. If the user does not exist, the function returns an Unauthorized status.
4. The function calculates today's start timestamp using the moment() library.
5. It retrieves the claimed betting rewards for the user.
6. Variables are initialized for calculating different betting amounts.
7. Based on the time difference, the function calculates the betting amounts for different types of bets.
8. It calculates the total betting amount as the summed value of the different betting amounts.
9. Based on the week's betting amount, the function generates a list of available weekly betting rewards with their required betting amounts, bonus amounts, and claim statuses.
10. The function filters the claimable bonus data based on the requirements.
11. If no claimable bonus data is found, the function returns an appropriate response.
12. The function retrieves the claimed bonus data based on the provided ID.
13. It checks if the bonus for the provided ID has already been claimed. If so, the function returns an appropriate response.
14. The function updates the user's money and total money in the database.
15. It inserts the claimed reward into the database.
16. Finally, the function returns a success response if the reward is successfully claimed.

## Function: weeklyBetttingRewordRecord

*const* weeklyBetttingRewordRecord = *async* (*req*, *res*) *=>* {

  try {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

      "SELECT `phone` FROM `users` WHERE `token` = ? AND `veri` = 1",

      [authToken],

    );

*const* user = userRow?.[0];

    if (!user) {

      return *res*.status(401).json({ message: "Unauthorized" });

    }

*const* [claimedRewardsRow] = await connection.execute(

      "SELECT \* FROM `claimed\_rewards` WHERE `type` = ? AND `phone` = ?",

      [REWARD\_TYPES\_MAP.WEEKLY\_BETTING\_BONUS, user.phone],

    );

*const* claimedRewardsData = claimedRewardsRow.map((*claimedReward*) *=>* {

*const* currentDailyBetttingReword = WeekBettingBonusList.find(

        (*item*) *=>* *item*?.id === *claimedReward*?.reward\_id,

      );

      return {

        id: *claimedReward*.reward\_id,

        requireBettingAmount: currentDailyBetttingReword?.bettingAmount || 0,

        amount: *claimedReward*.amount,

        status: *claimedReward*.status,

        time: moment.unix(*claimedReward*.time).format("YYYY-MM-DD HH:mm:ss"),

      };

    });

    return *res*.status(200).json({

      data: claimedRewardsData,

      status: true,

      message: "Successfully fetched weekly betting bonus record",

    });

  } catch (error) {

    console.log(error);

    return *res*.status(500).json({ message: "Something went wrong" });

  }

};

The weeklyBetttingRewordRecord function is responsible for retrieving the weekly betting bonus records for a user. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies.
2. It retrieves the user's phone number from the database.
3. If the user does not exist, the function returns an Unauthorized status.
4. The function retrieves the claimed rewards data for the user based on the reward type.
5. It generates the claimed rewards data with the required information, such as reward ID, required betting amount, claimed amount, status, and time.
6. Finally, the function returns the claimed rewards data as a response.

## Function: addonstake

*const* addonstake = *async* (*req*, *res*) *=>* {

*const* authToken = *req*.cookies.auth;

*let* stack\_amt = *req*.body.stack\_amt;

*let* stack\_period = *req*.body.stack\_period;

*let* stack\_monthly\_roi = *req*.body.stack\_monthly\_roi;

*let* stack\_yealy\_roi = *req*.body.stack\_yealy\_roi;

  if (!stack\_amt && !stack\_period ) {

      return *res*.status(200).json({

          message: 'Failed',

          status: false,

          timeStamp: timeNow,

      })

  }

*const* [userRow] = await connection.execute(

    "SELECT `phone`,`money` FROM `users` WHERE `token` = ? AND `veri` = 1",

    [authToken],

  );

*const* user = userRow?.[0];

  if (!user) {

    return *res*.status(401).json({ message: "Unauthorized" });

  }

*var* rewardid\_bonus = '';

  if(parseInt(stack\_period) < 3)

  {

    rewardid\_bonus = REWARD\_TYPES\_MAP.ROI\_STAKE\_1;

  }

  else if(parseInt(stack\_period) < 6)

  {

    rewardid\_bonus = REWARD\_TYPES\_MAP.ROI\_STAKE\_3;

  }

  else if(parseInt(stack\_period) < 9)

  {

    rewardid\_bonus = REWARD\_TYPES\_MAP.ROI\_STAKE\_6;

  }

  else

  {

    rewardid\_bonus = REWARD\_TYPES\_MAP.ROI\_STAKE\_12;

  }

  if (user.money - stack\_amt >= 0) {

*let* date = new *Date*();

*let* checkTime = timerJoin(date.getTime());

*var* toDate = new *Date*(new *Date*(date).setMonth(date.getMonth() + parseInt(stack\_period)));

*let* checkTime2 = timerJoin(toDate.getTime());

    await connection.execute(

      "INSERT INTO `claimed\_rewards` (`reward\_id`, `type`, `phone`, `amount`, `status`, `time`, `stake\_amnt`, `from\_date`, `to\_date`) VALUES (?, ?, ?, ?, ?, ?,?, ?, ?)",

      [

        parseInt("136"),

        rewardid\_bonus,

        user.phone,

        stack\_yealy\_roi,

        2,

        timeNow,

        stack\_amt,

        checkTime,

        checkTime2

      ],

      await connection.query('UPDATE users SET money = money - ? WHERE phone = ? ', [parseFloat(stack\_amt).toFixed(4).toString(), user.phone])

    );

    return *res*.status(200).json({

      status: true,

      message: "Successfully raised stake",

    });

  }

  else

  {

    return *res*.status(200).json({

      message: 'The balance is not enough to fulfill the request',

      status: false,

      timeStamp: timeNow,

  });

  }

}

The addonstake function is responsible for allowing a user to add a stake to their account. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies and the stake amount, period, monthly ROI, and yearly ROI from the request body.
2. If the stake amount and period are not provided, the function returns a failure status.
3. The function retrieves the user's phone number and money from the database.
4. If the user does not exist, the function returns an Unauthorized status.
5. The function assigns the appropriate reward ID based on the stake period.
6. If the user has enough money to fulfill the stake request, the function calculates the stake end date based on the stack period and inserts the stake data into the database.
7. Finally, the function returns a success response if the stake is successfully added.

## Function: getstakedetails

*const* getstakedetails = *async* (*req*, *res*) *=>* {

*const* authToken = *req*.cookies.auth;

*const* [userRow] = await connection.execute(

    "SELECT `phone`,`money` FROM `users` WHERE `token` = ? AND `veri` = 1",

    [authToken],

  );

*const* user = userRow?.[0];

  if (!user) {

    return *res*.status(401).json({ message: "Unauthorized" });

  }

*const* [row\_1] = await connection.execute(

    "SELECT phone AS phone FROM `claimed\_rewards` where `reward\_id` = 136 GROUP BY phone;"

  );

*const* stakeusers = row\_1.length || 0;

*const* [[row\_2]] = await connection.execute(

    "SELECT SUM(stake\_amnt) AS `sum` FROM `claimed\_rewards` WHERE `reward\_id` = 136;"

  );

*const* stakeamount = row\_2.sum || 0;

*const* [[row\_3]] = await connection.execute(

    "SELECT SUM(amount) AS `sum` FROM `claimed\_rewards` WHERE `reward\_id` = 136 AND status = 1;"

  );

*const* stakerewards =  row\_3.sum || 0;

*const* [userstackes] = await connection.query('SELECT \* FROM claimed\_rewards WHERE `phone` = ? AND `reward\_id` = 136 ', [user.phone]);

*const* [useractivestakes] = await connection.query('SELECT \* FROM claimed\_rewards WHERE status = 2 AND `phone` = ? AND `reward\_id` = 136 ', [user.phone]);

  return *res*.status(200).json({

    message: 'Success',

    status: true,

    totalstakes : stakeamount,

    totalstakeusers : stakeusers,

    totalclaimedrewards : stakerewards,

    userstakedetails : userstackes,

    useractivestakes: useractivestakes,

  });

}

The getstakedetails function is responsible for retrieving the stake details for a user. It takes in the req and res objects as parameters.

1. The function retrieves the authentication token from the request cookies.
2. The function retrieves the user's phone number and money from the database.
3. If the user does not exist, the function returns an Unauthorized status.
4. The function retrieves the total stake details from the database, such as the number of stake users, total stake amount, and total claimed rewards.
5. The function retrieves the stake details for the user from the claimed\_rewards table.
6. The function retrieves the active stake details for the user from the claimed\_rewards table.
7. Finally, the function returns the stake details as a response.

## Function: formateT

*function* formateT(*params*) {

*let* result = (*params* < 10) ? "0" + *params* : *params*;

  return result;

}

The formateT function is a utility function that formats a given value by adding a leading zero if it is less than 10. It takes in a single params parameter and returns the formatted value.

## Function: timerJoin

The timerJoin function is a utility function that formats a given timestamp into a date and time string. It takes in two parameters: params (the timestamp) and addHours (the number of hours to add to the date). If no timestamp is provided, the function uses the current date and time.

*function* timerJoin(*params* = '', *addHours* = 0) {

*let* date = '';

  if (*params*) {

      date = new *Date*(*Number*(*params*));

  } else {

      date = new *Date*();

  }

  date.setHours(date.getHours() + *addHours*);

*let* years = formateT(date.getFullYear());

*let* months = formateT(date.getMonth() + 1);

*let* days = formateT(date.getDate());

*let* hours = date.getHours() % 12;

  hours = hours === 0 ? 12 : hours;

*let* ampm = date.getHours() < 12 ? "AM" : "PM";

*let* minutes = formateT(date.getMinutes());

*let* seconds = formateT(date.getSeconds());

  return years + '-' + months + '-' + days + ' ' + hours + ':' + minutes + ':' + seconds + ' ' + ampm;

}

1. The function creates a new date object based on the provided timestamp or the current date if no timestamp is provided.
2. If the addHours parameter is provided, the function adds the specified number of hours to the date.
3. The function extracts the year, month, day, hours, minutes, and seconds from the date object.
4. It formats the extracted values with leading zeros using the formateT utility function.
5. Finally, the function returns the formatted date and time string.