**Documentation - vipController.js**

This code file, vipController.js, is responsible for managing the VIP rewards and level system in the software project.

**Constants**

The following constants are defined in the code file:

import {

  REWARD\_STATUS\_TYPES\_MAP,

  REWARD\_TYPES\_MAP,

} from "../constants/reward\_types.js";

* REWARD\_STATUS\_TYPES\_MAP: A map of reward status types.
* REWARD\_TYPES\_MAP: A map of reward types.

**Variable**

The following variable is defined in the code file:

DailyRebateBonusList: The array consists of objects that represent daily rebate bonus information for a software project.

The code initializes the DailyRebateBonusList array with an initial object containing the following properties:

* id: the unique identifier of the rebate bonus.
* rebetAmount: the amount of rebate available on a daily basis.
* bonusAmount: the initial bonus amount, which is initially set to 0.

DailRebateBonusList = [{

    id: 1,

    rebetAmount: 50,

    bonusAmount: 0,

  },

]

* VIP\_REWORDS\_LIST: An array of VIP reward objects, each containing the level, required experience points, level up reward amount, monthly reward amount, safe percentage, and rebate rate.

*const* VIP\_REWORDS\_LIST = [

  {

    level: 0,

    expRequired: 0,

    levelUpReward: 0,

    monthlyReward: 0,

    safePercentage: 0,

    rebateRate: 0,

  },

  {

    level: 1,

    expRequired: 3\_000,

    levelUpReward: 60,

    monthlyReward: 30,

    safePercentage: 0.2,

    rebateRate: 0.05,

  },

  {

    level: 2,

    expRequired: 30\_000,

    levelUpReward: 180,

    monthlyReward: 90,

    safePercentage: 0.2,

    rebateRate: 0.05,

  },

  {

    level: 3,

    expRequired: 500\_000,

    levelUpReward: 600,

    monthlyReward: 280,

    safePercentage: 0.2,

    rebateRate: 0.1,

  },

  {

    level: 4,

    expRequired: 5\_000\_000,

    levelUpReward: 1600,

    monthlyReward: 600,

    safePercentage: 0.3,

    rebateRate: 0.1,

  },

  {

    level: 5,

    expRequired: 10\_000\_000,

    levelUpReward: 4000,

    monthlyReward: 1600,

    safePercentage: 0.3,

    rebateRate: 0.1,

  },

  {

    level: 6,

    expRequired: 80\_000\_000,

    levelUpReward: 16000,

    monthlyReward: 4000,

    safePercentage: 0.3,

    rebateRate: 0.15,

  },

  {

    level: 7,

    expRequired: 300\_000\_000,

    levelUpReward: 65000,

    monthlyReward: 16000,

    safePercentage: 0.4,

    rebateRate: 0.15,

  },

  {

    level: 8,

    expRequired: 1\_000\_000\_000,

    levelUpReward: 170000,

    monthlyReward: 65000,

    safePercentage: 0.4,

    rebateRate: 0.15,

  },

  {

    level: 9,

    expRequired: 5\_000\_000\_000,

    levelUpReward: 700000,

    monthlyReward: 170000,

    safePercentage: 0.4,

    rebateRate: 0.2,

  },

  {

    level: 10,

    expRequired: 9\_999\_999\_999,

    levelUpReward: 1700000,

    monthlyReward: 700000,

    safePercentage: 0.7,

    rebateRate: 0.3,

  },

];

**Helper Functions**

The following helper functions are defined in the code file:

* getSubordinateDataByPhone(phone): Retrieves the total betting amount for a given phone number by querying multiple tables in the database. Returns an object with the total betting amount.

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

*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;

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

    "SELECT SUM(money) as totalBettingAmount FROM trx\_wingo\_bets WHERE phone = ?",

    [*phone*],

  );

*const* gameTrxWingoBettingAmount = gameTrxWingo[0].totalBettingAmount || 0;

  return {

    bettingAmount:

      parseInt(gameWingoBettingAmount) +

      parseInt(gameK3BettingAmount) +

      parseInt(game5DBettingAmount) +

      parseInt(gameTrxWingoBettingAmount),

  };

};

This code defines an asynchronous function getSubordinateDataByPhone that takes a phone parameter.

1. Inside the function, it performs multiple SQL queries to calculate the total betting amounts for different games based on the provided phone.
2. Each SQL query is executed using connection.query with a specific table name and the phone number as a parameter, and the result is stored in a destructured array variable.
3. The total betting amount for each game is extracted from the first element of the result array or default to 0 if the result is falsy. Finally, the total of all game betting amounts is calculated by summing up the parsed integer values of each game's betting amount.
4. The function returns an object containing the total bettingAmount as the sum of all game betting amounts.

* insertRewordClaim(rewardData): Inserts a new reward claim into the database with the provided reward data, including reward ID, reward type, phone number, reward amount, status, and time.

*const* insertRewordClaim = *async* ({

*rewardId*,

*rewardType*,

*phone*,

*amount*,

*status*,

*time*,

}) *=>* {

  await connection.execute(

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

    [*rewardId*, *rewardType*, *phone*, *amount*, *status*, *time*],

  );

};

The given code defines an asynchronous function insertRewordClaim that takes an object as an argument containing properties rewardId, rewardType, phone, amount, status, and time.

1. Inside the function, it uses connection.execute to execute an SQL query to insert data into a table named claimed\_rewards. The values for the query are provided as an array containing rewardId, rewardType, phone, amount, status, and time, corresponding to the placeholders ? in the query.

**Controller Functions**

The following controller functions are defined in the code file:

* releaseRebateCommission(): Releases rebate commission to eligible users. This function calculates the daily rebate amount based on the commissions earned the previous day, checks for any claimable rebate bonuses, and updates the user's balance and reward claim records accordingly.

*const* releaseRebateCommission = *async* () *=>* {

  try {

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

      "SELECT `vip\_level`, `phone`, `money`,`id` FROM `users`",

    );

    for (*let* user of users) {

*const* dailyRebateRewordId = 1;

*const* today = moment().subtract(1, "days").valueOf();

      //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) {

*console*.log("You does not meet the requirements to claim this reward!");

      }

      else{

*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) {

          console.log("Bonus already claimed");

        }

        else{

*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,

          ],

        );

        console.log("Successfully claimed daily betting bonus");

      }

      }

    }

  } catch (error) {

    console.log(error);

  }

}

The given code defines an asynchronous function releaseRebateCommission that processes rebate commissions for users. Here is a breakdown of the code:

1. It retrieves user data from the database with columns 'vip\_level', 'phone', 'money', and 'id'.
2. Calculates the daily rebate amount for each user.
3. Filters claimed rewards and generates a list based on required conditions.
4. Checks if there are claimable bonuses based on specific criteria.
5. If claimable bonuses exist, it updates user's money balance and records the claimed reward in the database.
6. Error handling is done using a try-catch block to log any errors that may occur.
7. The code utilizes asynchronous functions to interact with the database and handle the processing logic based on the retrieved data.

* releaseVIPLevel(): Releases VIP level rewards to eligible users. This function calculates the user's VIP level based on their betting amount, checks for any level up rewards and monthly rewards, and updates the user's balance and reward claim records accordingly.

*const* releaseVIPLevel = *async* () *=>* {

  try {

    // let lastMonth1st2amUnix = moment().subtract(1, "months").startOf("month").add(2, "hours").unix();

    // let lastMonth1st2am = moment(lastMonth1st2amUnix \* 1000).format("YYYY-MM-DD HH:mm:ss");

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

      "SELECT `vip\_level`, `phone`, `money`,`id` FROM `users`",

    );

    for (*let* user of users) {

*const* { phone, money, vip\_level: lastVipLevel, id } = user;

*const* { bettingAmount: expPoint } =

        await getSubordinateDataByPhone(phone);

*const* currentVipLevel = VIP\_REWORDS\_LIST.find((*vip*, *index*) *=>* {

        if (VIP\_REWORDS\_LIST?.[*index* + 1]) {

          return (

            expPoint >= *vip*.expRequired &&

            expPoint < VIP\_REWORDS\_LIST[*index* + 1].expRequired

          );

        } else {

          return expPoint >= *vip*.expRequired;

        }

      });

      if (currentVipLevel.level === 0) {

        continue;

      }

      if (currentVipLevel.level > parseInt(lastVipLevel)) {

        // Update user's VIP level and Insert levelUpReward

        await connection.execute(

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

          [

            currentVipLevel.level,

            currentVipLevel.levelUpReward,

            currentVipLevel.levelUpReward,

            phone,

          ],

        );

        await insertRewordClaim({

          rewardId: currentVipLevel.level,

          rewardType: REWARD\_TYPES\_MAP.VIP\_LEVEL\_UP\_BONUS,

          phone,

          amount: currentVipLevel.levelUpReward,

          status: REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

          time: moment().unix(),

        });

      } else {

        await connection.execute(

          "UPDATE users SET vip\_level = ? WHERE phone = ?",

          [currentVipLevel.level, phone],

        );

      }

      // Insert monthly reward

      await insertRewordClaim({

        rewardId: currentVipLevel.level,

        rewardType: REWARD\_TYPES\_MAP.VIP\_MONTHLY\_REWARD,

        phone,

        amount: currentVipLevel.monthlyReward,

        status: REWARD\_STATUS\_TYPES\_MAP.SUCCESS,

        time: moment().unix(),

      });

*let* sql\_noti = 'INSERT INTO notification SET recipient = ?, description = ?, isread = ?, noti\_type = ?';

      await connection.query(sql\_noti, [id, "Congrates! you received an monthly reward of "+currentVipLevel.monthlyReward+" !", '0', "Reward"]);

      await connection.execute(

        "UPDATE users SET money = money + ? WHERE phone = ?",

        [currentVipLevel.monthlyReward, phone],

      );

    }

  } catch (error) {

    console.log(error);

  }

};

The given code is an asynchronous function named releaseVIPLevel that processes VIP level upgrades and rewards for users based on their betting amounts and conditions.

1. It starts by querying the database to get a list of users' VIP levels, phones, money, and IDs. Then, it iterates over each user and retrieves their betting amount from another function named getSubordinateDataByPhone.
2. Based on the user's current and last VIP level, it determines if there is a level up and updates the user's VIP level, money, and total money accordingly. It also inserts reward claims for level up bonuses and monthly rewards.
3. After processing the VIP level upgrades and rewards, it inserts a notification into a database table and updates the user's money with the monthly reward amount.
4. If any error occurs during the process, it is caught and logged to the console.

* getMyVIPLevelInfo(req, res): Retrieves the VIP level information for the authenticated user. This function fetches the user's phone number and VIP level from the database, calculates their experience points, and returns the VIP level, experience points, and days left until the next payout as JSON.

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

  try {

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

    if (!authToken) {

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

    }

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

      "SELECT `phone`, `vip\_level` FROM `users` WHERE `token` = ?",

      [authToken],

    );

*const* phone = users[0].phone;

*const* vipLevel = users[0].vip\_level;

*const* { bettingAmount: expPoint } = await getSubordinateDataByPhone(phone);

*const* payoutDaysLeft = moment()

      .startOf("month")

      .add(1, "month")

      .add(2, "hours")

      .diff(moment(), "days");

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

      status: true,

      data: {

        expPoint: expPoint,

        vipLevel: vipLevel,

        payoutDaysLeft: payoutDaysLeft,

      },

    });

  } catch (error) {

    console.log(error);

*res*.status(500).json({ status: false, message: "Internal server error" });

  }

};

The provided code is an asynchronous function in Node.js used to retrieve and calculate VIP user information. Here is the explanation of the code:

1. const getMyVIPLevelInfo is an asynchronous function that takes req (request) and res (response) as parameters.
2. It first extracts the authorization token from the request cookies.
3. If the authorization token is missing, it returns a 401 status with a JSON response mentioning 'Unauthorized'.
4. It then executes a SQL query to fetch user data based on the token.
5. The function then retrieves the phone number and VIP level from the fetched user data.
6. It calls another function getSubordinateDataByPhone to get betting amount and renames it to expPoint.
7. Calculates the number of days left until the next payout deadline using the moment library.
8. Returns a 200 status with a JSON response containing the user's experience points, VIP level, and days left for payout.
9. If any error occurs during this process, it logs the error and sends a 500 status response with the message 'Internal server error'.

* getVIPHistory(req, res): Retrieves the reward claim history for the authenticated user. This function fetches the user's phone number from the database and returns a list of claimed rewards (VIP level up bonuses and monthly rewards) as JSON.

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

  try {

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

    if (!authToken) {

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

    }

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

      "SELECT `phone` FROM `users` WHERE `token` = ?",

      [authToken],

    );

*const* phone = users[0].phone;

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

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

      [

        phone,

        REWARD\_TYPES\_MAP.VIP\_LEVEL\_UP\_BONUS,

        REWARD\_TYPES\_MAP.VIP\_MONTHLY\_REWARD,

      ],

    );

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

      status: true,

      list: claimedRewards,

    });

  } catch (error) {

    console.log(error);

*res*.status(500).json({ status: false, message: "Internal server error" });

  }

};

This code defines an asynchronous function getVIPHistory that handles a request and response object. It first tries to extract the authorization token from the request cookies.

1. If the token is missing, it returns a 401 status with an 'Unauthorized' message. Otherwise, it queries the database to fetch the user's phone number using the token.
2. Next, it executes another query to retrieve claimed rewards based on the user's phone number and specific reward types (VIP levels and monthly rewards). The results are stored in claimedRewards.
3. Finally, the function sends a JSON response with a status of true and the list of claimed rewards. In case of any errors during the process, it catches the error, logs it, and responds with a 500 status and an 'Internal server error' message.