**Code Documentation: winGoController.js**

This code file contains the implementation of the winGo game controller functions.

**Function: winGoPage**

This function renders the "bet/wingo/win.ejs" view for the winGo page.

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

    return *res*.render("bet/wingo/win.ejs");

}

**Example:**

winGoPage()

**Function: winGoPage3**

This function renders the "bet/wingo/win3.ejs" view for the winGo page with 3 players.

*const* winGoPage3 = *async* (*req*, *res*) *=>* {

    return *res*.render("bet/wingo/win3.ejs");

}

**Example:**

winGoPage3()

**Function: winGoPage5**

This function renders the "bet/wingo/win5.ejs" view for the winGo page with 5 players.

*const* winGoPage5 = *async* (*req*, *res*) *=>* {

    return *res*.render("bet/wingo/win5.ejs");

}

**Example:**

winGoPage5()

**Function: winGoPage10**

This function renders the "bet/wingo/win10.ejs" view for the winGo page with 10 players.

*const* winGoPage10 = *async* (*req*, *res*) *=>* {

    return *res*.render("bet/wingo/win10.ejs");

}

**Example:**

winGoPage10()

**Function: isNumber**

This function takes a parameter and checks if it is a number.

*const* isNumber = (*params*) *=>* {

*let* pattern = /^[0-9]\*\d$/;

    return pattern.test(*params*);

}

**Example:**

isNumber(params)

**Function: formateT**

This function formats a number by adding a leading zero if it is less than 10.

*function* formateT(*params*) {

*let* result = (*params* < 10) ? "0" + *params* : *params*;

    return result;

}

**Example:**

formateT(params)

**Function: timerJoin**

This function generates a formatted string representing the current date and time with an optional offset in hours.

*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;

}

**Example:**

timerJoin(params, addHours)

**Function: rosesPlus**

This function calculates and distributes commissions to referrers based on the user's total money and level.

*const* rosesPlus = *async* (*auth*, *money*) *=>* {

*const* [level] = await connection.query('SELECT \* FROM level ');

*const* [user] = await connection.query('SELECT `phone`, `code`, `invite`, `user\_level`, `total\_money` FROM users WHERE token = ? AND veri = 1 LIMIT 1 ', [*auth*]);

*let* userInfo = user[0];

*const* [f1] = await connection.query('SELECT `phone`, `code`, `invite`, `rank`, `user\_level`, `total\_money` FROM users WHERE code = ? AND veri = 1 LIMIT 1 ', [userInfo.invite]);

    if (userInfo.total\_money >= 100) {

        if (f1.length > 0) {

*let* infoF1 = f1[0];

            for (*let* levelIndex = 1; levelIndex <= 6; levelIndex++) {

*let* rosesF = 0;

                if (infoF1.user\_level >= levelIndex && infoF1.total\_money >= 100) {

                    rosesF = (*money* / 100) \* level[levelIndex - 1].f1;

                    if (rosesF > 0) {

                        await connection.query('UPDATE users SET money = money + ?, roses\_f = roses\_f + ?, roses\_today = roses\_today + ? WHERE phone = ? ', [rosesF, rosesF, rosesF, infoF1.phone]);

*let* timeNow = *Date*.now();

*const* sql2 = `INSERT INTO roses SET

                            phone = ?,

                            code = ?,

                            invite = ?,

                            f1 = ?,

                            time = ?`;

                        await connection.execute(sql2, [infoF1.phone, infoF1.code, infoF1.invite, rosesF, timeNow]);

*const* sql3 = `

                            INSERT INTO turn\_over (phone, code, invite, daily\_turn\_over, total\_turn\_over)

                            VALUES (?, ?, ?, ?, ?)

                            ON DUPLICATE KEY UPDATE

                            daily\_turn\_over = daily\_turn\_over + VALUES(daily\_turn\_over),

                            total\_turn\_over = total\_turn\_over + VALUES(total\_turn\_over)

                            `;

                        await connection.execute(sql3, [infoF1.phone, infoF1.code, infoF1.invite, *money*, *money*]);

                    }

                }

*const* [fNext] = await connection.query('SELECT `phone`, `code`, `invite`, `rank`, `user\_level`, `total\_money` FROM users WHERE code = ? AND veri = 1 LIMIT 1 ', [infoF1.invite]);

                if (fNext.length > 0) {

                    infoF1 = fNext[0];

                } else {

                    break;

                }

            }

        }

    }

}

The provided code is an asynchronous function named rosesPlus that takes two parameters: auth and money. It performs a series of database queries and updates based on certain conditions.

1. It first fetches a row from the level table and assigns it to level.
2. Then it fetches a user's information based on the auth token from the users table and stores it in userInfo.
3. It further fetches another user's information based on the invite field of userInfo and stores it in f1.
4. If the total\_money of userInfo is greater than or equal to 100, it proceeds with additional checks and updates.
5. Within a loop iterating from 1 to 6, it calculates rosesF based on certain conditions and updates the database accordingly.
6. It also inserts records into the roses and turn\_over tables based on the calculated values.
7. At each iteration, it fetches the next user's information and continues the process until a certain condition is met.
8. If no more next user is found, it breaks out of the loop.

**Example:**

rosesPlus(auth, money)

**Function: rosesPlus1**

This function calculates and distributes commissions to referrers based on the user's total money and level using a different logic.

*const* rosesPlus1 = *async* (*phone*, *money*, *levels* = [], *timeNow* = "") *=>* {

    try {

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

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

        [*phone*],

      );

*const* userInfo = userResult[0];

      if (!userInfo) {

        return;

      }

*let* timeNow1 = *Date*.now();

*let* userReferrer = userInfo.invite;

*let* commissionsToInsert = [];

*let* usersToUpdate = [];

      for (*let* i = 0; i < *levels*.length; i++) {

*const* levelCommission = *levels*[i] \* *money*;

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

          "SELECT phone, money, code, invite FROM users WHERE code = ?",

          [userReferrer],

        );

*const* referrerInfo = referrerRows[0];

        if (referrerInfo) {

*const* commissionId = generateCommissionId();

          commissionsToInsert.push([

            commissionId,

            referrerInfo.phone,

            userInfo.phone,

            levelCommission,

            i + 1,

            timeNow1,

          ]);

          usersToUpdate.push([levelCommission, referrerInfo.phone]);

          userReferrer = referrerInfo.invite;

        } else {

          console.log(`Level ${i + 1} referrer not found.`);

          break;

        }

      }

      if (commissionsToInsert.length > 0) {

        await connection.query(

          "INSERT INTO commissions (commission\_id, phone, from\_user\_phone, money, level, time) VALUES ?",

          [commissionsToInsert],

        );

      }

      if (usersToUpdate.length > 0) {

*const* updatePromises = usersToUpdate.map(([*money*, *phone*]) *=>*

          connection.query("UPDATE users SET money = money + ? WHERE phone = ?", [

*money*,

*phone*,

          ]),

        );

        await *Promise*.all(updatePromises);

      }

      return {

        success: true,

        message: "Commissions calculated and inserted successfully.",

      };

    } catch (error) {

      console.error(error);

      return { success: false, message: error.message };

    }

  };

The provided code is an asynchronous function named rosesPlus1 that takes in parameters phone, money, levels (an array, defaulting to an empty array), and timeNow (a string, defaulting to an empty string). It performs a series of database queries and updates based on the input parameters.

Here is a breakdown of the code:

1. It queries the database to retrieve user information based on the provided phone (assuming there is a connection object available).
2. It iterates over the specified levels, calculating commissions and updating user information recursively.
3. It inserts commission records into a table named commissions if there are commissions to insert.
4. It updates user balances if there are users to update.
5. If an error occurs during the process, it logs the error and returns an object with success: false and an error message.
6. Otherwise, it returns an object with success: true and a success message.

**Example:**

rosesPlus1(phone, money, levels, timeNow)

**Function: distributeCommission**

This function distributes commissions to referrers based on the bets placed by users.

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

    try {

*const* { startOfYesterdayTimestamp, endOfYesterdayTimestamp } =

        yesterdayTime();

*const* [levelResult] = await connection.query("SELECT f1 FROM level");

*const* levels = levelResult.map((*row*) *=>* *row*.f1 / 100);

      //console.log(levelResult);

      //console.log(levels);

      // const [bets] = await connection.query('SELECT phone, SUM(money + fee) AS total\_money FROM minutes\_1 WHERE time > ? AND time <= ? GROUP BY phone', [startOfDay, endTime]);

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

        `

        SELECT phone, SUM(total\_money) AS total\_money

        FROM (

          SELECT phone, SUM(money + fee) AS total\_money

          FROM minutes\_1

          WHERE time > ? AND time <= ?

          GROUP BY phone

          UNION ALL

          SELECT phone, SUM(money + fee) AS total\_money

          FROM minutes\_1

          WHERE time > ? AND time <= ?

          GROUP BY phone

        ) AS combined

        GROUP BY phone

        `,

        [

          startOfYesterdayTimestamp,

          endOfYesterdayTimestamp,

          startOfYesterdayTimestamp,

          endOfYesterdayTimestamp,

        ],

      );

*const* promises = bets.map((*bet*) *=>*

        rosesPlus1(*bet*.phone, *bet*.total\_money, levels, endOfYesterdayTimestamp),

      );

*const* response = await *Promise*.all(promises);

      return {

        success: true,

        message: "Commissions distributed successfully.",

      };

    } catch (error) {

      console.error(error);

      return { success: false, message: error.message };

    }

  };

In this code snippet:

1. An asynchronous function distributeCommission is defined using the async keyword.
2. Inside a try-catch block, the code retrieves timestamps for the start and end of yesterday using a function yesterdayTime().
3. The code then queries the database to fetch level information and calculates levels based on the result.
4. Another query is made to fetch phone numbers and total money for bets made within a specific time range, combining two sets of data and summing the total money.
5. For each bet, a promise is created using rosesPlus1 function.
6. These promises are then executed concurrently using Promise.all.
7. If all promises are successful, the function returns an object with success set to true and a success message. Otherwise, it logs the error and returns an object with success set to false along with the error message.

**Example:**

distributeCommission()

**Function: betWinGo**

This function handles the bet placement for the winGo game.

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

*let* { typeid, join, x, money } = *req*.body;

*let* auth = *req*.cookies.auth;

    if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

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

            message: 'Error!',

            status: true

        });

    }

*let* gameJoin = '';

    if (typeid == 1) gameJoin = 'wingo';

    if (typeid == 3) gameJoin = 'wingo3';

    if (typeid == 5) gameJoin = 'wingo5';

    if (typeid == 10) gameJoin = 'wingo10';

*const* [winGoNow] = await connection.query(`SELECT period FROM wingo WHERE status = 0 AND game = '${gameJoin}' ORDER BY id DESC LIMIT 1 `);

*const* [user] = await connection.query('SELECT `phone`, `code`, `invite`, `level`, `money` FROM users WHERE token = ? AND veri = 1  LIMIT 1 ', [auth]);

    if (!winGoNow[0] || !user[0] || !isNumber(x) || !isNumber(money)) {

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

            message: 'Error!',

            status: true

        });

    }

*let* userInfo = user[0];

*let* period = winGoNow[0].period;

*let* fee = (x \* money) \* 0.02;

*let* total = (x \* money) - fee;

*let* timeNow = *Date*.now();

*let* check = userInfo.money - total;

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

*let* years = formateT(date.getFullYear());

*let* months = formateT(date.getMonth() + 1);

*let* days = formateT(date.getDate());

*let* id\_product = years + months + days + Math.floor(Math.random() \* 1000000000000000);

*let* formatTime = timerJoin();

*let* color = '';

    if (join == 'l') {

        color = 'big';

    } else if (join == 'n') {

        color = 'small';

    } else if (join == 't') {

        color = 'violet';

    } else if (join == 'd') {

        color = 'red';

    } else if (join == 'x') {

        color = 'green';

    } else if (join == '0') {

        color = 'red-violet';

    } else if (join == '5') {

        color = 'green-violet';

    } else if (join % 2 == 0) {

        color = 'red';

    } else if (join % 2 != 0) {

        color = 'green';

    }

*let* checkJoin = '';

    if (!isNumber(join) && join == 'l' || join == 'n') {

        checkJoin = `

        <div data-v-a9660e98="" class="van-image" style="width: 30px; height: 30px;">

            <img src="/images/${(join == 'n') ? 'small' : 'big'}.png" class="van-image\_\_img">

        </div>

        `

    } else {

        checkJoin =

            `

        <span data-v-a9660e98="">${(isNumber(join)) ? join : ''}</span>

        `

    }

*let* result = `

    <div data-v-a9660e98="" issuenumber="${period}" addtime="${formatTime}" rowid="1" class="hb">

        <div data-v-a9660e98="" class="item c-row">

            <div data-v-a9660e98="" class="result">

                <div data-v-a9660e98="" class="select select-${(color)}">

                    ${checkJoin}

                </div>

            </div>

            <div data-v-a9660e98="" class="c-row c-row-between info">

                <div data-v-a9660e98="">

                    <div data-v-a9660e98="" class="issueName">

                        ${period}

                    </div>

                    <div data-v-a9660e98="" class="tiem">${formatTime}</div>

                </div>

            </div>

        </div>

        <!---->

    </div>

    `;

*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;

    }

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

    if (check >= 0) {

*const* sql = `INSERT INTO minutes\_1 SET

        id\_product = ?,

        phone = ?,

        code = ?,

        invite = ?,

        stage = ?,

        level = ?,

        money = ?,

        amount = ?,

        fee = ?,

        get = ?,

        game = ?,

        bet = ?,

        status = ?,

        today = ?,

        time = ?`;

        await connection.execute(sql, [id\_product, userInfo.phone, userInfo.code, userInfo.invite, period, userInfo.level, total, x, fee, 0, gameJoin, join, 0, checkTime, timeNow]);

        await connection.execute('UPDATE `users` SET `money` = `money` - ? WHERE `token` = ? ', [money \* x, auth]);

*const* [users] = await connection.query('SELECT `money`, `level` FROM users WHERE token = ? AND veri = 1  LIMIT 1 ', [auth]);

        await rosesPlus(auth, money \* x);

        // const [level] = await connection.query('SELECT \* FROM level ');

        // let level0 = level[0];

        // const sql2 = `INSERT INTO roses SET

        // phone = ?,

        // code = ?,

        // invite = ?,

        // f1 = ?,

        // f2 = ?,

        // f3 = ?,

        // f4 = ?,

        // time = ?`;

        // let total\_m = money \* x;

        // let f1 = (total\_m / 100) \* level0.f1;

        // let f2 = (total\_m / 100) \* level0.f2;

        // let f3 = (total\_m / 100) \* level0.f3;

        // let f4 = (total\_m / 100) \* level0.f4;

        // await connection.execute(sql2, [userInfo.phone, userInfo.code, userInfo.invite, f1, f2, f3, f4, timeNow]);

        // console.log(level);

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

            message: 'Successful bet',

            status: true,

            data: result,

            change: users[0].level,

            money: users[0].money,

        });

    } else {

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

            message: 'The amount is not enough',

            status: false

        });

    }

}

The provided code defines an asynchronous function betWinGo that handles a betting activity.

1. It extracts the values of typeid, join, x, and money from the request body and reads auth from cookies.
2. It then checks if the typeid is one of the allowed values (1, 3, 5, or 10). If not, it returns an error message.
3. Next, the code executes certain database queries based on the extracted values and performs calculations related to the betting process.
4. There is a conditional block that determines the color based on the value of join.
5. Further down, the code generates HTML content stored in the result variable based on multiple conditions.
6. The function timerJoin is defined to format the timestamp.
7. Finally, based on certain conditions, it either processes the bet and returns a success response with relevant data or returns an error message if the amount is insufficient.

**Example:**

betWinGo(req, res)

**Function: listOrderOld**

This function retrieves a list of past winGo game orders based on the game type, page number, and page limit.

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

*let* { typeid, pageno, pageto } = *req*.body;

    if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

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

            message: 'Error!',

            status: true

        });

    }

    if (pageno < 0 || pageto < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

*let* auth = *req*.cookies.auth;

*const* [user] = await connection.query('SELECT `phone`, `code`, `invite`, `level`, `money` FROM users WHERE token = ? AND veri = 1  LIMIT 1 ', [auth]);

*let* game = '';

    if (typeid == 1) game = 'wingo';

    if (typeid == 3) game = 'wingo3';

    if (typeid == 5) game = 'wingo5';

    if (typeid == 10) game = 'wingo10';

*const* [wingo] = await connection.query(`SELECT \* FROM wingo WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT ${pageno}, ${pageto} `);

*const* [wingoAll] = await connection.query(`SELECT \* FROM wingo WHERE status != 0 AND game = '${game}' `);

*const* [period] = await connection.query(`SELECT period FROM wingo WHERE status = 0 AND game = '${game}' ORDER BY id DESC LIMIT 1 `);

   if (!wingo[0]) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!pageno || !pageto || !user[0] || !wingo[0] || !period[0]) {

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

            message: 'Error!',

            status: true

        });

    }

*let* page = Math.ceil(wingoAll.length / 10);

    return *res*.status(200).json({

        code: 0,

        msg: "Receive success",

        data: {

            gameslist: wingo,

        },

        period: period[0].period,

        page: page,

        status: true

    });

}

This code defines an asynchronous function listOrderOld that takes req and res as parameters.

1. It extracts typeid, pageno, and pageto from req.body using destructuring.
2. It then performs checks on typeid, pageno, and pageto values. If any of the conditions fail, it returns an error response. It queries a database using the connection object to fetch user details, game data, and period information based on the provided parameters.
3. Based on the typeid, it assigns a corresponding game value. It retrieves data from the 'wingo' table with specific conditions and sorting.
4. If the fetched data is empty or any required data is missing, it returns an error response. Otherwise, it calculates the total number of pages and constructs a success response with game data, period, and page information.

**Example:**

listOrderOld(req, res)

**Function: GetMyEmerdList**

This function retrieves a list of winGo game orders made by the user based on the game type, page number, and page limit.

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

*let* { typeid, pageno, pageto } = *req*.body;

    // if (!pageno || !pageto) {

    //     pageno = 0;

    //     pageto = 10;

    // }

    if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

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

            message: 'Error!',

            status: true

        });

    }

    if (pageno < 0 || pageto < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

*let* auth = *req*.cookies.auth;

*let* game = '';

    if (typeid == 1) game = 'wingo';

    if (typeid == 3) game = 'wingo3';

    if (typeid == 5) game = 'wingo5';

    if (typeid == 10) game = 'wingo10';

*const* [user] = await connection.query('SELECT `phone`, `code`, `invite`, `level`, `money` FROM users WHERE token = ? AND veri = 1 LIMIT 1 ', [auth]);

*const* [minutes\_1] = await connection.query(`SELECT \* FROM minutes\_1 WHERE phone = ? AND game = '${game}' ORDER BY id DESC LIMIT ${*Number*(pageno) + ',' + *Number*(pageto)}`, [user[0].phone]);

*const* [minutes\_1All] = await connection.query(`SELECT \* FROM minutes\_1 WHERE phone = ? AND game = '${game}' ORDER BY id DESC `, [user[0].phone]);

    if (!minutes\_1[0]) {

        return res.status(200).json({

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!pageno || !pageto || !user[0] || !minutes\_1[0]) {

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

            message: 'Error!',

            status: true

        });

    }

*let* page = Math.ceil(minutes\_1All.length / 10);

*let* datas = minutes\_1.map((*data*) *=>* {

*let* { id, phone, code, invite, level, game, ...others } = *data*;

        return others;

    });

    return *res*.status(200).json({

        code: 0,

        msg: "Receive success",

        data: {

            gameslist: datas,

        },

        page: page,

        status: true

    });

}

The provided code is an asynchronous function named GetMyEmerdList that takes two parameters: req (request) and res (response).

1. It extracts the typeid, pageno, and pageto from the request body.
2. If the typeid is not 1, 3, 5, or 10, it returns an error response. If pageno or pageto is less than 0, it returns a response indicating no more data.
3. It then sets the game based on the typeid value and queries the database to fetch user information and relevant data based on the game and user phone number.
4. If the queried data is empty or any required data is missing, it returns an error response. Otherwise, it calculates the total number of pages based on the fetched data.
5. It then extracts specific properties from the fetched data, excluding some specific fields, and constructs a response JSON object containing the extracted data along with the total page count and a success status.

**Example:**

GetMyEmerdList(req, res)

**Function: Stat\_listOrderOld**

This function retrieves statistical data for past winGo game orders based on the game type, page number, and page limit.

*const* Stat\_listOrderOld = *async* (*req*, *res*) *=>* {

*let* { typeid, pageno, pageto } = *req*.body;

    if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

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

            message: 'Error!',

            status: true

        });

    }

*let* game = '';

    if (typeid == 1) game = 'wingo';

    if (typeid == 3) game = 'wingo3';

    if (typeid == 5) game = 'wingo5';

    if (typeid == 10) game = 'wingo10';

*const* [wingo] = await connection.query(`SELECT amount FROM wingo WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT ${pageno}, ${pageto} `);

   if (!wingo[0]) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!pageno || !pageto || !wingo[0]) {

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

            message: 'Error!',

            status: true

        });

    }

    return *res*.status(200).json({

        code: 0,

        msg: "Receive success",

        data: {

            gameslist: wingo,

        },

        status: true

    });

  };

In this code snippet, an asynchronous function named Stat\_listOrderOld is defined that takes req and res as parameters.

1. It extracts typeid, pageno, and pageto from the request body. If typeid is not 1, 3, 5, or 10, it returns an error response.
2. Based on the typeid, a corresponding game value is assigned. Then a query is made to the database to select an amount from the wingo table where status is not 0 and game matches the assigned game value. The data is then ordered by id in descending order and limited based on pageno and pageto.
3. If no data is returned, an error response is sent. Additionally, if pageno, pageto, or the first element of wingo is falsy, an error response is returned as well.
4. Finally, if there is data available, a success response is sent with the retrieved wingo data along with a success status.

**Example:**

Stat\_listOrderOld(req, res)

**Function: addWinGo**

This function adds a new round of the winGo game with the given game type.

*const* addWinGo = *async* (*game*) *=>* {

    try {

*let* join = '';

        if (*game* == 1) join = 'wingo';

        if (*game* == 3) join = 'wingo3';

        if (*game* == 5) join = 'wingo5';

        if (*game* == 10) join = 'wingo10';

*const* [winGoNow] = await connection.query(`SELECT period FROM wingo WHERE status = 0 AND game = "${join}" ORDER BY id DESC LIMIT 1 `);

*const* [setting] = await connection.query('SELECT \* FROM `admin` ');

*let* period = winGoNow[0].period; // cầu hiện tại

*let* amount = Math.floor(Math.random() \* 10);

*const* [minPlayers] = await connection.query(`SELECT \* FROM minutes\_1 WHERE status = 0 AND game = "${join}"`);

        if (minPlayers.length >= 2) {

*const* betColumns = [

                // red\_small

                { name: 'red\_0', bets: ['0', 't', 'd', 'n'] },

                { name: 'red\_2', bets: ['2', 'd', 'n'] },

                { name: 'red\_4', bets: ['4', 'd', 'n'] },

                // green small

                { name: 'green\_1', bets: ['1', 'x', 'n'] },

                { name: 'green\_3', bets: ['3', 'x', 'n'] },

                // green big

                { name: 'green\_5', bets: ['5', 'x', 't', 'l'] },

                { name: 'green\_7', bets: ['7', 'x', 'l'] },

                { name: 'green\_9', bets: ['9', 'x', 'l'] },

                // red big

                { name: 'red\_6', bets: ['6', 'd', 'l'] },

                { name: 'red\_8', bets: ['8', 'd', 'l'] }

            ];

*const* totalMoneyPromises = betColumns.map(*async* column *=>* {

*const* [result] = await connection.query(`

                SELECT SUM(money) AS total\_money

                FROM minutes\_1

                WHERE game = "${join}" AND status = 0 AND bet IN (${column.bets.map(bet *=>* `"${bet}"`).join(',')})

            `);

                return { name: column.name, total\_money: result[0].total\_money ? parseInt(result[0].total\_money) : 0 };

            });

*const* categories = await *Promise*.all(totalMoneyPromises);

*let* smallestCategory = categories.reduce((smallest, category) *=>*

                (smallest === null || category.total\_money < smallest.total\_money) ? category : smallest

                , null);

*const* colorBets = {

                red\_6: [6],

                red\_8: [8],

                red\_2: [2], //0 removed

                red\_4: [4],

                green\_3: [3],

                green\_7: [7], //5 removed

                green\_9: [9], //

                green\_1: [1],

                green\_5: [5],

                red\_0: [0],

            };

*const* betsForCategory = colorBets[smallestCategory.name] || [];

*const* availableBets = betsForCategory.filter(bet *=>*

                !categories.find(category *=>* category.name === smallestCategory.name && category.total\_money < smallestCategory.total\_money)

            );

*let* lowestBet;

            if (availableBets.length > 0) {

                lowestBet = availableBets[0];

            } else {

                lowestBet = betsForCategory.reduce((lowest, bet) *=>*

                    (bet < lowest) ? bet : lowest

                );

            }

            amount = lowestBet;

        } else if (minPlayers.length === 1 && parseFloat(minPlayers[0].money) >= 20) {

*const* betColumns = [

                { name: 'red\_small', bets: ['0', '2', '4', 'd', 'n'] },

                { name: 'red\_big', bets: ['6', '8', 'd', 'l'] },

                { name: 'green\_big', bets: ['5', '7', '9', 'x', 'l'] },

                { name: 'green\_small', bets: ['1', '3', 'x', 'n'] },

                { name: 'violet\_small', bets: ['0', 't', 'n'] },

                { name: 'violet\_big', bets: ['5', 't', 'l'] }

            ];

*const* categories = await *Promise*.all(betColumns.map(*async* column *=>* {

*const* [result] = await connection.query(`

                    SELECT SUM(money) AS total\_money

                    FROM minutes\_1

                    WHERE game = "${join}" AND status = 0 AND bet IN (${column.bets.map(bet *=>* `"${bet}"`).join(',')})

                `);

                return { name: column.name, total\_money: parseInt(result[0]?.total\_money) || 0 };

            }));

*const* colorBets = {

                red\_big: [6, 8],

                red\_small: [2, 4], //0 removed

                green\_big: [7, 9], //5 removed

                green\_small: [1, 3],

                violet\_big: [5],

                violet\_small: [0],

            };

*const* smallestCategory = categories.reduce((smallest, category) *=>*

                (!smallest || category.total\_money < smallest.total\_money) ? category : smallest

            );

*const* betsForCategory = colorBets[smallestCategory.name] || [];

*const* availableBets = betsForCategory.filter(bet *=>*

                !categories.find(category *=>* category.name === smallestCategory.name && category.total\_money < smallestCategory.total\_money)

            );

*const* lowestBet = availableBets.length > 0 ? availableBets[0] : *Math*.min(...betsForCategory);

            amount = lowestBet;

        }

        // xanh đỏ tím

*let* timeNow = *Date*.now();

*let* nextResult = '';

        if (game == 1) nextResult = setting[0].wingo1;

        if (game == 3) nextResult = setting[0].wingo3;

        if (game == 5) nextResult = setting[0].wingo5;

        if (game == 10) nextResult = setting[0].wingo10;

*let* newArr = '';

        if (nextResult == '-1') {

            await connection.execute(`UPDATE wingo SET amount = ?,status = ? WHERE period = ? AND game = "${join}"`, [amount, 1, period]);

            newArr = '-1';

        } else {

*let* result = '';

*let* arr = nextResult.split('|');

*let* check = arr.length;

            if (check == 1) {

                newArr = '-1';

            } else {

                for (*let* i = 1; i < arr.length; i++) {

                    newArr += arr[i] + '|';

                }

                newArr = newArr.slice(0, -1);

            }

            result = arr[0];

            await connection.execute(`UPDATE wingo SET amount = ?,status = ? WHERE period = ? AND game = "${join}"`, [result, 1, period]);

        }

*let* gameRepresentationId = GameRepresentationIds.WINGO[*game*];

*let* NewGamePeriod = generatePeriod(gameRepresentationId);

*const* sql = `INSERT INTO wingo SET

        period = ?,

        amount = ?,

        game = ?,

        status = ?,

        time = ?`;

        await connection.execute(sql, [NewGamePeriod, 0, join, 0, timeNow]);

        if (*game* == 1) join = 'wingo1';

        if (*game* == 3) join = 'wingo3';

        if (*game* == 5) join = 'wingo5';

        if (*game* == 10) join = 'wingo10';

        await connection.execute(`UPDATE admin SET ${join} = ?`, [newArr]);

    } catch (error) {

        if (error) {

            console.log(error);

        }

    }

}

The provided code defines an asynchronous function addWinGo that takes a parameter game.

1. It first checks the value of game and sets the join variable based on the condition.
2. The function then executes database queries using connection.query to fetch data from the database tables wingo, admin, and minutes\_1 based on certain criteria.
3. Depending on the results of the queries and conditions, the function calculates and assigns values to variables such as amount, categories, and smallestCategory.
4. It processes the data further to determine the smallest category and corresponding bets, updating the amount accordingly.
5. Subsequently, it handles updating database records and inserting new records based on the calculated values.
6. The function also includes error handling using a try...catch block to log errors to the console if any.

**Example:**

addWinGo(game)

**Function: handlingWinGo1P**

This function handles the result and payout for the winGo game with 1 player.

*const* handlingWinGo1P = *async* (*typeid*) *=>* {

*let* game = '';

    if (*typeid* == 1) game = 'wingo';

    if (*typeid* == 3) game = 'wingo3';

    if (*typeid* == 5) game = 'wingo5';

    if (*typeid* == 10) game = 'wingo10';

*const* [winGoNow] = await connection.query(`SELECT \* FROM wingo WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT 1 `);

    // update ket qua

    await connection.execute(`UPDATE minutes\_1 SET result = ? WHERE status = 0 AND game = '${game}'`, [winGoNow[0].amount]);

*let* result = *Number*(winGoNow[0].amount);

    switch (result) {

        case 0:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "d" AND bet != "0" AND bet != "t" `, []);

            break;

        case 1:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "x" AND bet != "1" `, []);

            break;

        case 2:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "d" AND bet != "2" `, []);

            break;

        case 3:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "x" AND bet != "3" `, []);

            break;

        case 4:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "d" AND bet != "4" `, []);

            break;

        case 5:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "x" AND bet != "5" AND bet != "t" `, []);

            break;

        case 6:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "d" AND bet != "6" `, []);

            break;

        case 7:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "x" AND bet != "7" `, []);

            break;

        case 8:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "d" AND bet != "8" `, []);

            break;

        case 9:

            await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet != "l" AND bet != "n" AND bet != "x" AND bet != "9" `, []);

            break;

        default:

            break;

    }

    if (result < 5) {

        await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet = "l" `, []);

    } else {

        await connection.execute(`UPDATE minutes\_1 SET status = 2 WHERE status = 0 AND game = "${game}" AND bet = "n" `, []);

    }

    // lấy ra danh sách đặt cược chưa xử lý

*const* [order] = await connection.execute(`SELECT \* FROM minutes\_1 WHERE status = 0 AND game = '${game}' `);

    for (*let* i = 0; i < order.length; i++) {

*let* orders = order[i];

*let* result = orders.result;

*let* bet = orders.bet;

*let* total = orders.money;

*let* id = orders.id;

*let* phone = orders.phone;

*var* nhan\_duoc = 0;

        // x - green

        // t - Violet

        // d - red

        // Sirf 1-4 aur 6-9 tk hi \*9 aana chahiye

        // Aur 0 aur 5 pe \*4.5

        // Aur red aur green pe \*2

        // 1,2,3,4,6,7,8,9

        if (bet == 'l' || bet == 'n') {

            nhan\_duoc = total \* 2;

        } else {

            if (result == 0 || result == 5) {

                if (bet == 'd' || bet == 'x') {

                    nhan\_duoc = total \* 1.5;

                } else if (bet == 't') {

                    nhan\_duoc = total \* 4.5;

                } else if (bet == "0" || bet == "5") {

                    nhan\_duoc = total \* 4.5;

                }

            } else {

                if (result == 1 && bet == "1") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 1 && bet == 'x') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 2 && bet == "2") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 2 && bet == 'd') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 3 && bet == "3") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 3 && bet == 'x') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 4 && bet == "4") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 4 && bet == 'd') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 6 && bet == "6") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 6 && bet == 'd') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 7 && bet == "7") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 7 && bet == 'x') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 8 && bet == "8") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 8 && bet == 'd') {

                        nhan\_duoc = total \* 2;

                    }

                }

                if (result == 9 && bet == "9") {

                    nhan\_duoc = total \* 9;

                } else {

                    if (result == 9 && bet == 'x') {

                        nhan\_duoc = total \* 2;

                    }

                }

            }

        }

*const* [users] = await connection.execute('SELECT `money` FROM `users` WHERE `phone` = ?', [phone]);

*let* totals = parseFloat(users[0].money) + parseFloat(nhan\_duoc);

        await connection.execute('UPDATE `minutes\_1` SET `get` = ?, `status` = 1 WHERE `id` = ? ', [parseFloat(nhan\_duoc), id]);

*const* sql = 'UPDATE `users` SET `money` = ? WHERE `phone` = ? ';

        await connection.execute(sql, [totals, phone]);

    }

}

The provided code is an asynchronous function in JavaScript that handles a specific scenario related to updating game results and processing bets stored in a database. Here's a breakdown of the code:

1. A function named handlingWinGo1P is declared with an async keyword, indicating it is an asynchronous function that can use the await keyword to handle promises.
2. The function takes a parameter called typeid.
3. Based on the value of typeid, the variable game is set to correspond to different game names ('wingo', 'wingo3', 'wingo5', 'wingo10').
4. A database query is made to retrieve the latest record from the 'wingo' table based on certain conditions.
5. The retrieved result is used to update the 'minutes\_1' table with specific conditions and criteria based on the game type and result.
6. Subsequent database queries and updates are made to process the bets, calculate winnings, and update user balances accordingly.
7. The code contains multiple conditional statements using if and switch statements to handle different scenarios based on the result of the game and the bets placed.
8. Overall, the code processes game results and bets stored in the 'minutes\_1' table, updating statuses, calculating winnings, and updating user balances.

**Example:**

handlingWinGo1P(typeid)