**K5D Controller Documentation**

This code file is responsible for handling the K5D game functionality. It includes functions to render different pages, validate bets, handle game results, and distribute commissions.

**Functions**

* K5DPage

A function that renders the K5D page for betting.

*const* K5DPage = *async* (*req*, *res*) *=>* {

    return *res*.render("bet/5d/5d.ejs");

}

* K5DPage3

A function that renders the K5D3 page for betting.

*const* K5DPage3 = *async* (*req*, *res*) *=>* {

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

}

* K5DPage5

A function that renders the K5D5 page for betting.

*const* K5DPage5 = *async* (*req*, *res*) *=>* {

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

}

* K5DPage10

A function that renders the K5D10 page for betting.

*const* K5DPage10 = *async* (*req*, *res*) *=>* {

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

}

* isNumber

A function that checks if the provided parameter is a number.

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

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

    return pattern.test(*params*);

}

* formateT

A function that 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;

    }

* timerJoin

A function that returns a formatted date and time with an optional additional number of 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;

    }

The provided code defines a function named timerJoin that takes two parameters: params (defaulted to an empty string) and addHours (defaulted to 0).

1. It then constructs a date based on the params value if provided (as a number representing milliseconds since the Unix Epoch) or the current date if params is empty. It adds addHours to the hour component of the date.
2. Next, it extracts the year, month, day, hour (in 12-hour format with AM/PM), minutes, and seconds from the date object. Leading zeros are added to ensure the correct format.
3. Finally, it returns a formatted string representing the date in the format: YYYY-MM-DD HH:MM:SS AM/PM.

* rosesPlus

A function that distributes commission to users based on their betting activity.

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

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

*let* level0 = level[0];

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

*let* userInfo = user[0];

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

    if (*money* >= 10000) {

        if (f1.length > 0) {

*let* infoF1 = f1[0];

*let* rosesF1 = (*money* / 100) \* level0.f1;

*const* sql6 = `

                            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(sql6, [infoF1.phone, infoF1.code, infoF1.invite, *money*, *money*]);

            await connection.query('UPDATE users SET money = money + ?, roses\_f1 = roses\_f1 + ?, roses\_f = roses\_f + ?, roses\_today = roses\_today + ? WHERE phone = ? ', [rosesF1, rosesF1, rosesF1, rosesF1, infoF1.phone]);

*const* [f2] = await connection.query('SELECT `phone`, `code`, `invite`, `rank` FROM users WHERE code = ? AND veri = 1  LIMIT 1 ', [infoF1.invite]);

            if (f2.length > 0) {

*let* infoF2 = f2[0];

*let* rosesF2 = (*money* / 100) \* level0.f2;

                await connection.query('UPDATE users SET money = money + ?, roses\_f = roses\_f + ?, roses\_today = roses\_today + ? WHERE phone = ? ', [rosesF2, rosesF2, rosesF2, infoF2.phone]);

*const* [f3] = await connection.query('SELECT `phone`, `code`, `invite`, `rank` FROM users WHERE code = ? AND veri = 1  LIMIT 1 ', [infoF2.invite]);

                if (f3.length > 0) {

*let* infoF3 = f3[0];

*let* rosesF3 = (*money* / 100) \* level0.f3;

                    await connection.query('UPDATE users SET money = money + ?, roses\_f = roses\_f + ?, roses\_today = roses\_today + ? WHERE phone = ? ', [rosesF3, rosesF3, rosesF3, infoF3.phone]);

*const* [f4] = await connection.query('SELECT `phone`, `code`, `invite`, `rank` FROM users WHERE code = ? AND veri = 1  LIMIT 1 ', [infoF3.invite]);

                    if (f4.length > 0) {

*let* infoF4 = f4[0];

*let* rosesF4 = (*money* / 100) \* level0.f4;

                        await connection.query('UPDATE users SET money = money + ?, roses\_f = roses\_f + ?, roses\_today = roses\_today + ? WHERE phone = ? ', [rosesF4, rosesF4, rosesF4, infoF4.phone]);

                    }

                }

            }

        }

    }

}

The given code defines an asynchronous function rosesPlus that takes two parameters auth and money.

1. Inside the function, multiple database queries are made to fetch user and level information from the database using the provided connection. Based on the value of money, certain calculations are performed to update the roses and money values for users in the database.
2. Here's a brief breakdown of the code:
3. It fetches the level information from the database.
4. It then fetches information about the user corresponding to the provided auth token.
5. Further queries are made to fetch related users' information based on invite codes, and calculations are done to update roses and money accordingly.
6. Multiple nested queries are executed to update the roses and money for a chain of users (f1, f2, f3, f4) based on invite relationships.
7. The code logic seems to handle referral-based rewards ensuring that users receive roses and money based on their invite hierarchy and the amount of money being transacted (checking if it's greater than or equal to 10000).

validateBet

A function that validates a bet by checking if the provided parameters are correct.

*const* validateBet = *async* (*join*, *list\_join*, *x*, *money*, *game*) *=>* {

*let* checkJoin = isNumber(*list\_join*);

*let* checkX = isNumber(*x*);

*const* checks = ['a', 'b', 'c', 'd', 'e', 'total'].includes(*join*);

*const* checkGame = ['1', '3', '5', '10'].includes(*String*(*game*));

*const* checkMoney = ['1', '10', '100', '1000'].includes(*money*);

    if (!checks || *list\_join*.length > 10 || !checkX || !checkMoney || !checkGame) {

        return false;

    }

    if (checkJoin) {

*let* arr = *list\_join*.split('');

*let* length = arr.length;

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

*const* joinNum = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'].includes(arr[i]);

            if (!joinNum) {

                return false;

            }

        }

    } else {

*let* arr = *list\_join*.split('');

*let* length = arr.length;

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

*const* joinStr = ["c", "l", "b", "s"].includes(arr[i]);

            if (!joinStr) {

                return false;

            }

        }

    }

    return true;

}

The given code defines an asynchronous function named validateBet that takes five parameters: join, list\_join, x, money, and game.

It performs validation checks based on the conditional statements in the function and returns true if all the checks pass, otherwise false.

Here are the main checks performed:

1. Checks if join is one of the values 'a', 'b', 'c', 'd', 'e', or 'total'.
2. Validates the length of list\_join and ensures all characters are either numeric or strings based on the value of checkJoin.
3. Verifies if x is a number.
4. Validates if money and game are specific predefined values.

The function utilizes a helper function isNumber() to check if the elements in list\_join are numeric.

betK5D

A function that handles the betting process for the K5D game.

*const* betK5D = *async* (*req*, *res*) *=>* {

    try {

*let* { join, list\_join, x, money, game } = *req*.body;

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

*let* validate = await validateBet(join, list\_join, x, money, game);

        if (!validate) {

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

                message: 'Invalid bet',

                status: false

            });

        }

*const* [k5DNow] = await connection.query(`SELECT period FROM d5 WHERE status = 0 AND game = ${game} 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 (k5DNow.length < 1 || user.length < 1) {

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

                message: 'Error!',

                status: false

            });

        }

*let* userInfo = user[0];

*let* period = k5DNow[0];

*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* total = money \* x \* (*String*(list\_join).split('').length);

*let* fee = total \* 0.02;

*let* price = total - fee;

*let* check = userInfo.money - total;

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

        console.log(check);

        if (check >= 0) {

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

*const* sql = `INSERT INTO result\_5d SET id\_product = ?,phone = ?,code = ?,invite = ?,stage = ?,level = ?,money = ?,price = ?,amount = ?,fee = ?,game = ?,join\_bet = ?,bet = ?,status = ?,time = ?,today = ?`;

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

            await connection.execute('UPDATE `users` SET `money` = `money` - ? WHERE `token` = ? ', [total, 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 = total;

*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]);

            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

            });

        }

    } catch (error) {

        if (error) console.log(error);

    }

}

betK5D is an asynchronous function that handles a betting operation. It receives req (request) and res (response) objects.

1. The function first extracts join, list\_join, x, money, and game from the request body, as well as the auth cookie from the request.
2. It then validates the bet using the validateBet function. If the bet is invalid, it returns a response with a message 'Invalid bet' and status false.
3. Next, the function queries the database to get the latest period of the game (k5DNow) and user information based on the authentication token.
4. If either the period or user information is not found, it returns an error response.
5. The function then generates an id\_product based on the current date and a random number, calculates the total bet amount, fee, and price.
6. It checks if the user has enough money for the bet. If so, it proceeds to insert the bet information into the database, update the user's money, perform additional database operations, and finally returns a success response with the new level and money of the user.
7. If the user does not have enough money, it returns a response indicating 'The amount is not enough'.
8. If an error occurs at any point, it is caught and logged. The function structure follows a try-catch pattern to handle errors gracefully.

* listOrderOld

A function that retrieves the list of old orders for a specific game and page range.

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

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

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

*let* checkGame = ['1', '3', '5', '10'].includes(*String*(gameJoin));

    if (!checkGame || pageno < 0 || pageto < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

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

*let* game = *Number*(gameJoin);

*const* [k5d] = await connection.query(`SELECT \* FROM d5 WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT ${pageno}, ${pageto} `);

*const* [k5dAll] = await connection.query(`SELECT \* FROM d5 WHERE status != 0 AND game = '${game}' `);

*const* [period] = await connection.query(`SELECT period FROM d5 WHERE status = 0 AND game = '${game}' ORDER BY id DESC LIMIT 1 `);

    if (k5d.length == 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            page: 1,

            status: false

        });

    }

    if (!pageno || !pageto || !user[0] || !k5d[0] || !period[0]) {

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

            message: 'Error!',

            status: false

        });

    }

*let* page = Math.ceil(k5dAll.length / 10);

    return *res*.status(200).json({

        code: 0,

        msg: "Get success",

        data: {

            gameslist: k5d,

        },

        period: period[0].period,

        page: page,

        status: true

    });

}

This code defines an asynchronous function listOrderOld that handles HTTP requests.

1. It extracts gameJoin, pageno, and pageto from the request body and retrieves auth from cookies.
2. It then performs a series of database queries based on the received parameters, checks conditions, and eventually returns a JSON response containing relevant data.
3. If certain conditions are not met (like no data or missing parameters), appropriate error responses are sent.
4. The final response includes game data, periods, and status based on the query results.

Stat\_listOrderOld

A function that retrieves statistical data for the list of old orders for a specific game and page range.

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

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

*let* checkGame = ['1', '3', '5', '10'].includes(*String*(gameJoin));

*let* game = *Number*(gameJoin);

*const* [k5d] = await connection.query(`SELECT result FROM d5 WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT ${pageno}, ${pageto} `);

    if (k5d.length == 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            page: 1,

            status: false

        });

    }

    if (!pageno || !pageto || !k5d[0]) {

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

            message: 'Error!',

            status: false

        });

    }

    return *res*.status(200).json({

        code: 0,

        msg: "Get success",

        data: {

            gameslist: k5d,

        },

        status: true

    });

  };

The given code defines an asynchronous function Stat\_listOrderOld that takes two arguments req and res.

1. It extracts the values of gameJoin, pageno, and pageto from the request body.
2. It checks if the gameJoin value is one of '1', '3', '5', or '10', and converts gameJoin to a number. Then it queries a database table 'd5' to fetch results where status is not 0 and the game matches the provided number, ordered by id in descending order with pagination defined by pageno and pageto.
3. If no data is found, it returns a JSON response indicating 'No more data' along with an empty list of games.
4. If pageno, pageto, or the first element of k5d is falsy, it returns an error response. Otherwise, it returns a successful response with the fetched data.

* GetMyEmerdList

A function that retrieves the user's list of emerald orders for a specific game and page range.

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

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

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

*let* checkGame = ['1', '3', '5', '10'].includes(*String*(gameJoin));

    if (!checkGame || pageno < 0 || pageto < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

*let* game = *Number*(gameJoin);

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

*const* [result\_5d] = await connection.query(`SELECT \* FROM result\_5d WHERE phone = ? AND game = '${game}' ORDER BY id DESC LIMIT ${*Number*(pageno) + ',' + *Number*(pageto)}`, [user[0].phone]);

*const* [result\_5dAll] = await connection.query(`SELECT \* FROM result\_5d WHERE phone = ? AND game = '${game}' ORDER BY id DESC `, [user[0].phone]);

    if (!result\_5d[0]) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            page: 1,

            status: false

        });

    }

    if (!pageno || !pageto || !user[0] || !result\_5d[0]) {

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

            message: 'Error!',

            status: true

        });

    }

*let* page = Math.ceil(result\_5dAll.length / 10);

*let* datas = result\_5d.map((*data*) *=>* {

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

        return others;

    });

    return *res*.status(200).json({

        code: 0,

        msg: "Get Success",

        data: {

            gameslist: datas,

        },

        page: page,

        status: true

    });

}

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

1. It extracts gameJoin, pageno, and pageto from req.body and assigns req.cookies.auth to auth.
2. It then checks the values of gameJoin, pageno, and pageto against certain conditions and returns a response with an error message if the conditions are not met.
3. Queries the database to fetch user data based on the provided token and verifies if the result exists. Then fetches data from the result\_5d table with certain conditions and calculates the total number of pages.
4. Maps the result data to extract specific fields and returns a JSON response with the extracted data along with the total number of pages.

* add5D

A function that adds a new 5D result to the database for a specific game.

*const* add5D = *async*(*game*) *=>* {

    try {

*let* join = '';

        if (*game* == 1) join = 'k5d';

        if (*game* == 3) join = 'k5d3';

        if (*game* == 5) join = 'k5d5';

        if (*game* == 10) join = 'k5d10';

*let* result2 = makeid(5);

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

*let* [k5D] = await connection.query(`SELECT period FROM d5 WHERE status = 0 AND game = ${*game*} ORDER BY id DESC LIMIT 1 `);

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

*let* period = k5D[0].period;

*let* gameRepresentationId = GameRepresentationIds.G5D[*game*];

*let* NewGamePeriod = generatePeriod(gameRepresentationId);

*let* nextResult = '';

        if (*game* == 1) nextResult = setting[0].k5d;

        if (*game* == 3) nextResult = setting[0].k5d3;

        if (*game* == 5) nextResult = setting[0].k5d5;

        if (*game* == 10) nextResult = setting[0].k5d10;

*let* newArr = '';

        if (nextResult == '-1') {

            await connection.execute(`UPDATE d5 SET result = ?,status = ? WHERE period = ? AND game = "${*game*}"`, [result2, 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 d5 SET result = ?,status = ? WHERE period = ? AND game = ${*game*}`, [result, 1, period]);

        }

*const* sql = `INSERT INTO d5 SET period = ?, result = ?, game = ?, status = ?, time = ?`;

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

        //console.log("insert");

        if (*game* == 1) join = 'k5d';

        if (*game* == 3) join = 'k5d3';

        if (*game* == 5) join = 'k5d5';

        if (*game* == 10) join = 'k5d10';

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

    } catch (error) {

        if (error) {

            console.log(error);

        }

    }

}

The provided code defines an asynchronous function add5D that takes a parameter game. Inside the function, it performs various operations related to updating and inserting data into a database using queries.

Here is a breakdown of the code:

1. It sets the variable join based on the value of game.
2. It generates a random result2 and captures the current time in timeNow.
3. Queries are made to retrieve data from the d5 table and admin table based on certain conditions.
4. It processes the retrieved data, updates records in the d5 table, and inserts new records based on the game and period.
5. Finally, it updates the admin table based on the previous calculations.
6. Any errors that occur during the process are caught and logged to the console.

* handling5D

A function that handles the result of a 5D game by updating the status of bets and distributing the winnings.

*async* *function* funHanding(*game*) {

*const* [k5d] = await connection.query(`SELECT \* FROM d5 WHERE status != 0 AND game = ${*game*} ORDER BY id DESC LIMIT 1 `);

*let* k5dInfo = k5d[0];

    // update ket qua

    await connection.execute(`UPDATE result\_5d SET result = ? WHERE status = 0 AND game = ${*game*}`, [k5dInfo.result]);

*let* result = *String*(k5dInfo.result).split('');

*let* a = result[0];

*let* b = result[1];

*let* c = result[2];

*let* d = result[3];

*let* e = result[4];

*let* total = 0;

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

        total += *Number*(result[i]);

    }

    // xử lý game a

*const* [joinA] = await connection.execute(`SELECT id, bet FROM result\_5d WHERE status = 0 AND game = ${*game*} AND join\_bet = 'a' `);

*let* lengthA = joinA.length;

    for (*let* i = 0; i < lengthA; i++) {

*let* info = joinA[i];

*let* sult = info.bet.split('');

*let* check = isNumber(info.bet);

        if (check) {

*const* joinNum = sult.includes(a);

            if (!joinNum) {

                await connection.execute(`UPDATE result\_5d SET status = 2 WHERE id = ? `, [info.id]);

            }

        }

    }

    if (lengthA > 0) {

        if(a == '0' || a == '1' || a == '2' || a == '3' || a == '4') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'a' AND bet = 'b' `)

        };

        if(a == '5' || a == '6' || a == '7' || a == '8' || a == '9') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'a' AND bet = 's' `)

        };

        if(*Number*(a) % 2 == 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'a' AND bet = 'l' `)

        };

        if(*Number*(a) % 2 != 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'a' AND bet = 'c' `)

        };

    }

    // xử lý game b

*const* [joinB] = await connection.execute(`SELECT id, bet FROM result\_5d WHERE status = 0 AND game = ${*game*} AND join\_bet = 'b' `);

*let* lengthB = joinB.length;

    for (*let* i = 0; i < lengthB; i++) {

*let* info = joinB[i];

*let* sult = info.bet.split('');

*let* check = isNumber(info.bet);

        if (check) {

*const* joinNum = sult.includes(b);

            if (!joinNum) {

                await connection.execute(`UPDATE result\_5d SET status = 2 WHERE id = ? `, [info.id]);

            }

        }

    }

    if (lengthB > 0) {

        if(b == '0' || b == '1' || b == '2' || b == '3' || b == '4') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'b' AND bet = 'b' `);

        };

        if(b == '5' || b == '6' || b == '7' || b == '8' || b == '9') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'b' AND bet = 's' `);

        };

        if(*Number*(b) % 2 == 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'b' AND bet = 'l' `);

        };

        if(*Number*(b) % 2 != 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'b' AND bet = 'c' `);

        };

    }

    // xử lý game c

*const* [joinC] = await connection.execute(`SELECT id, bet FROM result\_5d WHERE status = 0 AND game = ${*game*} AND join\_bet = 'c' `);

*let* lengthC = joinC.length;

    for (*let* i = 0; i < lengthC; i++) {

*let* info = joinC[i];

*let* sult = info.bet.split('');

*let* check = isNumber(info.bet);

        if (check) {

*const* joinNum = sult.includes(c);

            if (!joinNum) {

                await connection.execute(`UPDATE result\_5d SET status = 2 WHERE id = ? `, [info.id]);

            }

        }

    }

    if (lengthC > 0) {

        if(c == '0' || c == '1' || c == '2' || c == '3' || c == '4') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'c' AND bet = 'b' `);

        };

        if(c == '5' || c == '6' || c == '7' || c == '8' || c == '9') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'c' AND bet = 's' `);

        };

        if(*Number*(c) % 2 == 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'c' AND bet = 'l' `);

        };

        if(*Number*(c) % 2 != 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'c' AND bet = 'c' `);

        };

    }

    // xử lý game d

*const* [joinD] = await connection.execute(`SELECT id, bet FROM result\_5d WHERE status = 0 AND game = ${*game*} AND join\_bet = 'd' `);

*let* lengthD = joinD.length;

    for (*let* i = 0; i < lengthD; i++) {

*let* info = joinD[i];

*let* sult = info.bet.split('');

*let* check = isNumber(info.bet);

        if (check) {

*const* joinNum = sult.includes(d);

            if (!joinNum) {

                await connection.execute(`UPDATE result\_5d SET status = 2 WHERE id = ? `, [info.id]);

            }

        }

    }

    if (lengthD > 0) {

        if(d == '0' || d == '1' || d == '2' || d == '3' || d == '4') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'd' AND bet = 'b' `);

        };

        if(d == '5' || d == '6' || d == '7' || d == '8' || d == '9') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'd' AND bet = 's' `);

        };

        if(*Number*(d) % 2 == 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'd' AND bet = 'l' `);

        };

        if(*Number*(d) % 2 != 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'd' AND bet = 'c' `);

        };

    }

    // xử lý game e

*const* [joinE] = await connection.execute(`SELECT id, bet FROM result\_5d WHERE status = 0 AND game = ${*game*} AND join\_bet = 'e' `);

*let* lengthE = joinE.length;

    for (*let* i = 0; i < lengthE; i++) {

*let* info = joinE[i];

*let* sult = info.bet.split('');

*let* check = isNumber(info.bet);

        if (check) {

*const* joinNum = sult.includes(e);

            if (!joinNum) {

                await connection.execute(`UPDATE result\_5d SET status = 2 WHERE id = ? `, [info.id]);

            }

        }

    }

    if (lengthE > 0) {

        if(e == '0' || e == '1' || e == '2' || e == '3' || e == '4') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'e' AND bet = 'b' `);

        };

        if(e == '5' || e == '6' || e == '7' || e == '8' || e == '9') {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'e' AND bet = 's' `);

        };

        if(*Number*(e) % 2 == 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'e' AND bet = 'l' `);

        };

        if(*Number*(e) % 2 != 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'e' AND bet = 'c' `);

        };

    }

    // xử lý game e

*const* [joinTotal] = await connection.execute(`SELECT id, bet FROM result\_5d WHERE status = 0 AND game = ${*game*} AND join\_bet = 'total' `);

    if (joinTotal.length > 0) {

        if(total <= 22) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'total' AND bet = 'b' `);

        };

        if(total > 22) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'total' AND bet = 's' `);

        };

        if(total % 2 == 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'total' AND bet = 'l' `);

        };

        if(total % 2 != 0) {

            await connection.execute(`UPDATE result\_5d SET status = 2 WHERE join\_bet = 'total' AND bet = 'c' `);

        };

    }

}

The code defines an asynchronous function funHanding that processes game results from a database.

1. It fetches the latest game result from the d5 table using a SQL query.
2. It updates the result\_5d table with this result.
3. It splits the result into individual digits (a, b, c, d, e) and calculates their total.
4. For each of the five digits, it checks bets placed on them:

* It retrieves bets for 'a', 'b', 'c', 'd', and 'e' and checks if the corresponding digit is included in the bet.
* If the digit is not included, it updates the status of that bet to 2.
* Depending on the value of each digit, it also sets the bet status to 2 for specific conditions (like if the digit is in a certain range or if it is even/odd).

1. Finally, it handles a special case for bets on the total of the digits, updating their status based on whether the total is less than or equal to 22, greater than 22, and whether it is even or odd.

Overall, the function handles results and updates the status of various bets based on the game outcomes.

distributeCommission

A function that distributes commissions to users based on their betting activity.

*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);

        // 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 result\_5d

            WHERE time > ? AND time <= ?

            GROUP BY phone

            UNION ALL

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

            FROM result\_5d

            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 the provided code snippet:

1. We have an asynchronous function distributeCommission that handles commission distribution logic.
2. The function retrieves timestamps from the yesterdayTime function and level data from the database.
3. It then performs a query to get bets data based on certain conditions using connection.query.
4. For each bet, it processes the data with the rosesPlus1 function and stores the promises.
5. Finally, it awaits all promises to be resolved using Promise.all and returns a success message if everything executes correctly.

### makeid(length)

This function generates a random string of numbers with a specified length.

#### Parameters:

* length: The desired length of the generated string.

#### Return Value:

The generated random string.

*function* makeid(*length*) {

*var* result = '';

*var* characters = '0123456789';

*var* charactersLength = characters.length;

    for (*var* i = 0; i < *length*; i++) {

        result += characters.charAt(Math.floor(Math.random() \* charactersLength));

    }

    return result;

}

Here's a breakdown of the code:

1. var result string: Declares an empty string variable to store the generated ID.
2. var characters = "0123456789": Defines the characters that can be used in the ID.
3. charactersLength := len(characters): Calculates the length of the characters string.
4. rand.Seed(time.Now().UnixNano()): Seeds the random number generator. It's important to seed the generator only once.
5. for i := 0; i < length; i++: Loops 'length' times to generate the ID.
6. result += string(characters[rand.Intn(charactersLength)]): Picks a random character from 'characters' and appends it to the 'result' string.
7. return result: Returns the generated ID after the loop completes.

### rosesPlus1(phone, money, levels = [], timeNow = "")

This function calculates and inserts commissions based on a user's referral levels and the amount of money they have.

#### Parameters:

* phone: The phone number of the user.
* money: The amount of money the user has.
* levels (optional): An array of referral levels.
* timeNow (optional): The current time.

#### Return Value:

An object with the following properties:

* success: A boolean value indicating whether the commissions calculation and insertion were successful.
* message: A message indicating the status of the operation.

*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* userReferrer = userInfo.invite;

*let* commissionsToInsert = [];

*let* usersToUpdate = [];

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

      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 };

    }

  };

In the provided code snippet, an asynchronous function rosesPlus1 is defined with four parameters: phone, money, levels (with a default value of an empty array), and timeNow (with a default value of an empty string).

1. The function executes a series of operations to calculate and insert commissions for a referral system. It first retrieves user information from a database table 'users' based on the provided phone. If the user exists and is verified, it proceeds to calculate commissions based on the specified levels and money amount.
2. For each level in the levels array, it fetches the referrer information, generates a commission ID, adds the commission details to commissionsToInsert array, and updates the referrer's money balance. Finally, it inserts the calculated commissions into a 'commissions' table and updates the users' money balances.
3. If any error occurs during the process, it catches the error, logs it to the console, and returns an object indicating the process was unsuccessful along with an error message.

### handling5D(typeid)

This function handles the processing and updating of results related to a 5D game.

#### Parameters:

* typeid: The ID of the game.

*const* handling5D = *async*(*typeid*) *=>* {

*let* game = *Number*(*typeid*);

    await funHanding(game);

*const* [order] = await connection.execute(`SELECT id, phone, bet, price, money, fee, amount FROM result\_5d WHERE status = 0 AND game = ${game} `);

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

*let* orders = order[i];

*let* id = orders.id;

*let* phone = orders.phone;

*let* nhan\_duoc = 0;

*let* check = isNumber(orders.bet);

        if (check) {

*let* arr = orders.bet.split('');

*let* total = (orders.money / arr.length );

*let* fee = total \* 0.02;

*let* price = total - fee;

            nhan\_duoc += price \* 9;

        } else {

            nhan\_duoc += orders.price \* 2;

        }

        await connection.execute('UPDATE `result\_5d` SET `get` = ?, `status` = 1 WHERE `id` = ? ', [nhan\_duoc, id]);

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

        await connection.execute(sql, [nhan\_duoc, phone]);

    }

}

This code defines an asynchronous function handling5D that takes a parameter typeid.

1. It converts the typeid to a number and then awaits the execution of a function funHanding passing the game as argument.
2. It then executes a SQL query to select specific fields from a table result\_5d for a specific game.
3. A loop iterates over the results, calculates a value based on certain conditions, and updates the rows in the result\_5d table and the users table accordingly.