**Code Documentation - dailyController.js**

This code file is located at src/controllers/dailyController.js and contains functions related to daily operations in the software project.

### **dailyPage()**

This function renders the "statistical.ejs" page for the daily management system.

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

    return *res*.render("daily/statistical.ejs");

}

### **listMeber()**

This function renders the "members.ejs" page for the daily management system.

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

    return *res*.render("daily/members.ejs");

}

### **profileMember()**

This function renders the "profileMember.ejs" page for the daily management system.

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

    return *res*.render("daily/profileMember.ejs");

}

### **settingPage()**

This function renders the "settings.ejs" page for the daily management system.

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

    return *res*.render("daily/settings.ejs");

}

### 

### **listRecharge()**

This function renders the "listRecharge.ejs" page for the daily management system.

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

    return *res*.render("daily/listRecharge.ejs");

}

### **listWithdraw()**

This function renders the "listWithdraw.ejs" page for the daily management system.

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

    return *res*.render("daily/listWithdraw.ejs");

}

### 

### **pageInfo()**

This function retrieves the phone parameter from the request and renders the "profileMember.ejs" view with the phone data as a variable.

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

*let* phone = *req*.params.phone;

    return *res*.render("daily/profileMember.ejs", {phone});

}

### 

### **giftPage()**

This function retrieves the auth cookie value from the request and fetches data from the database. It renders the "giftPage.ejs" view with the fetched data as variables.

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

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

*const* [rows] = await connection.execute('SELECT `phone` FROM `users` WHERE `token` = ? AND veri = 1', [auth]);

*let* money = 0;

*let* money2 = 0;

    if(rows.length != 0) {

*const* [point\_list] = await connection.execute('SELECT `money`, `money\_us` FROM `point\_list` WHERE `phone` = ?', [rows[0].phone]);

        money = point\_list[0].money;

        money2 = point\_list[0].money\_us;

    }

    return *res*.render("daily/giftPage.ejs", {money, money2});

}

const giftPage = async(req, res) defines an asynchronous function named giftPage that takes req and res as parameters.

1. The function first extracts the auth value from the cookies of the request object. It then queries the database using the extracted auth value to retrieve the phone number of a user whose token is verified.
2. Two variables money and money2 are initialized with default values. If the query results contain at least one row, another query is made to fetch the money and money\_us values based on the phone number obtained from the previous query.
3. Finally, the function renders an EJS template giftPage.ejs passing the retrieved money and money2 values as context objects.

### **support()**

This function renders the "support.ejs" page for the daily management system.

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

    return *res*.render("daily/support.ejs");

}

### 

### **deleteBankRechargeById(id)**

This function deletes a bank recharge entry from the database based on the provided ID.

*const* deleteBankRechargeById = *async* (*id*) *=>* {

*const* [recharge] = await connection.query("DELETE FROM bank\_recharge WHERE id = ?", [*id*]);

    return recharge

}

### 

### **settingCollo\_Details(req, res)**

This function handles the setting details update for the daily management system. It retrieves data from the request body and performs database operations accordingly.

*const* settingCollo\_Details = *async* (*req*, *res*) *=>* {

    try {

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

*let* name\_bank = *req*.body.name\_bank;

*let* name = *req*.body.name;

*let* info = *req*.body.info;

*let* qr = *req*.body.qr;

*let* typer = *req*.body.typer;

        if (!auth || !typer) {

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

                message: 'Failed',

                status: false,

                timeStamp: timeNow,

            });

        }

*const* [users] = await connection.query('SELECT \* FROM users WHERE token = ?', [auth]);

        if (typer == 'bank') {

            await connection.query(`UPDATE bank\_recharge SET name\_bank = ?, name\_user = ?, stk = ? WHERE type = 'bank' AND phone = ?`, [name\_bank, name, info, users[0].phone]);

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

                message: 'Successful change',

                status: true,

                datas: [],

            });

        }

        if (typer == 'momo') {

*const* [bank\_recharge] = await connection.query(`SELECT \* FROM bank\_recharge WHERE phone = ?;`, [users[0].phone]);

*var* transfer\_mode = '';

            if(bank\_recharge.length != 0)

            {

                transfer\_mode = bank\_recharge[0].transfer\_mode;

            }

            else{

                transfer\_mode = "manual";

            }

*const* deleteRechargeQueries = bank\_recharge.map(*recharge* *=>* {

                return deleteBankRechargeById(*recharge*.id)

            });

            await *Promise*.all(deleteRechargeQueries)

            // await connection.query(`UPDATE bank\_recharge SET name\_bank = ?, name\_user = ?, stk = ?, qr\_code\_image = ? WHERE type = 'upi'`, [name\_bank, name, info, qr]);

*const* bankName = *req*.body.bank\_name

*const* username = *req*.body.username

*const* upiId = *req*.body.upi\_id

*const* usdtWalletAddress = *req*.body.usdt\_wallet\_address

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

            await connection.query("INSERT INTO bank\_recharge SET name\_bank = ?, name\_user = ?, stk = ?, qr\_code\_image = ?, transfer\_mode = ?,phone=?, colloborator\_action = ?, time = ?, type = 'momo'", [

                bankName, username, upiId, usdtWalletAddress,transfer\_mode,users[0].phone, "off", timeNow

            ])

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

                message: 'Successfully changed',

                status: true,

                datas: [],

            });

        }

} catch (error) {

    console.log(error)

    return *res*.status(500).json({

        message: 'Failed',

        status: false,

    });

}

}

The provided code is an asynchronous function settingCollo\_Details that handles different actions based on the value of typer property received in the request body. Here is a breakdown of its functionality:

1. It first extracts necessary data from the request object using req.body.
2. It checks if auth and typer are present in the request. If not, it returns a JSON response with a failure message.
3. It queries the database to fetch user details based on the provided token (auth).
4. If typer is 'bank', it updates a record in the bank\_recharge table.
5. If typer is 'momo', it performs the following actions:

* Queries the database to get bank recharge details based on the user's phone number.
* Determines the transfer\_mode based on the retrieved data or sets it to 'manual' if no data is found.
* Deletes existing bank recharge records asynchronously using Promise.all.
* Inserts a new record into the bank\_recharge table with the provided information.

1. If any error occurs during the execution, it logs the error and returns a failure JSON response.

### **settingGet(req, res)**

This function handles fetching settings and bank recharge information for the daily management system. It retrieves the auth cookie value from the request and performs database queries to fetch the required information.

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

    try {

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

        if (!auth) {

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

                message: 'Failed',

                status: false,

                timeStamp: timeNow,

            });

        }

*const* [rows] = await connection.execute('SELECT \* FROM `users` WHERE `token` = ? AND veri = 1', [auth]);

*const* [bank\_recharge] = await connection.query("SELECT \* FROM bank\_recharge where `phone` = ?", [rows[0].phone]);

*const* [bank\_recharge\_momo] = await connection.query("SELECT \* FROM bank\_recharge WHERE type = 'momo' AND `phone` = ?", [rows[0].phone]);

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

*let* bank\_recharge\_momo\_data

        if (bank\_recharge\_momo.length) {

            bank\_recharge\_momo\_data = bank\_recharge\_momo[0]

        }

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

            message: 'Success',

            status: true,

            settings: settings,

            datas: bank\_recharge,

            momo: {

                bank\_name: bank\_recharge\_momo\_data?.name\_bank || "",

                username: bank\_recharge\_momo\_data?.name\_user || "",

                upi\_id: bank\_recharge\_momo\_data?.stk || "",

                usdt\_wallet\_address: bank\_recharge\_momo\_data?.qr\_code\_image || "",

            }

        });

    } catch (error) {

        console.log(error)

        return *res*.status(500).json({

            message: 'Failed',

            status: false,

        });

    }

}

The provided code defines an asynchronous function settingGet that handles a GET request. Here is a breakdown of the code:

1. Accesses the auth value from request cookies.
2. If auth is not available, it returns a JSON response with a failure message.
3. Executes multiple SQL queries using the connection object to fetch user-related data and admin settings.
4. Determines data related to bank recharge with a specific type (e.g., 'momo').
5. Constructs a JSON response with success status and includes retrieved data like settings, bank recharge details, and momo details.
6. If any error occurs in the try block, it logs the error and sends a failure JSON response with status 500.

### **getUserDataByPhone(phone)**

This function retrieves user data from the database based on the provided phone number.

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

*let* [users] = await connection.query('SELECT `phone`, `code`,`name\_user`,`invite` FROM users WHERE `phone` = ? ', [*phone*]);

*const* user = users?.[0]

    if (user === undefined || user === null) {

        throw *Error*("Unable to get user data!")

    }

    return {

        phone: user.phone,

        code: user.code,

        username: user.name\_user,

        invite: user.invite,

    }

}

In this code snippet, an asynchronous function getUserDataByPhone is defined that takes phone as a parameter. Inside the function, a SQL query is executed to fetch user data based on the provided phone number.

1. The query result is stored in the users variable, and then the first user object (if any) is extracted using optional chaining into the user variable.
2. If no user is found (i.e., user is undefined or null), an error is thrown.
3. Finally, the function returns an object containing specific properties (phone, code, username, invite) extracted from the user object.

### **addUserAccountBalance()**

This function adds account balance for a user and their inviter in the daily management system. It takes an object with money, phone, and invite properties as input.

*const* addUserAccountBalance = *async* ({ *money*, *phone*, *invite* }) *=>* {

*const* user\_money = *money* + (*money* / 100) \* 5

*const* inviter\_money = (*money* / 100) \* 5

    await connection.query('UPDATE users SET money = money + ?, total\_money = total\_money + ? WHERE `phone` = ?', [user\_money, user\_money, *phone*]);

*const* [inviter] = await connection.query('SELECT phone FROM users WHERE `code` = ?', [*invite*]);

    if (inviter.length) {

        console.log(inviter)

        console.log(inviter\_money, inviter\_money, *invite*, inviter?.[0].phone)

        await connection.query('UPDATE users SET money = money + ?, total\_money = total\_money + ? WHERE `code` = ? AND `phone` = ?', [inviter\_money, inviter\_money, *invite*, inviter?.[0].phone]);

        console.log("SUCCESSFULLY ADD MONEY TO inviter")

    }

}

In the given code:

1. A function addUserAccountBalance is declared as asynchronous, taking an object containing money, phone, and invite as parameters.
2. user\_money is calculated by adding 5% of money to itself.
3. inviter\_money is set to 5% of money.
4. An SQL query is executed to update the money and total\_money fields in the users table based on the provided phone.
5. Another SQL query fetches the phone of a user based on the code received.
6. If a valid inviter is found, their money is updated similarly to the user's money update.
7. Finally, a success message is logged if money is successfully added to the inviter.

### **collo\_rechargeDuyet(req, res)**

This function handles the confirmation and deletion of a recharge request in the daily management system. It retrieves data from the request body and performs database operations accordingly.

*const* collo\_rechargeDuyet = *async* (*req*, *res*) *=>* {

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

*let* id = *req*.body.id;

*let* type = *req*.body.type;

    if (!auth || !id || !type) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

    if (type == 'confirm') {

        await connection.query(`UPDATE recharge SET status = 1 WHERE id = ?`, [id]);

*const* [info] = await connection.query(`SELECT \* FROM recharge WHERE id = ?`, [id]);

*const* [receiinfo] = await connection.query(`SELECT \* FROM users WHERE phone = ?`, [info?.[0]?.phone]);

        if(info?.[0]?.type.trim() == 'wallet')

        {

*const* [withdrainfo] = await connection.query(`SELECT \* FROM withdraw WHERE id\_order = ?`, [info?.[0]?.id\_order]);

*let* withInfo = withdrainfo[0];

*const* [senderinfo] = await connection.query(`SELECT \* FROM users WHERE phone = ?`, [withInfo.phone]);

            await connection.query(`UPDATE withdraw SET status = 1 WHERE id\_order = ?`, [info?.[0]?.id\_order]);

            await connection.query('UPDATE users SET money = money + ? WHERE phone = ?', [info?.[0]?.money, info?.[0]?.phone]);

            await connection.query(`UPDATE users SET money = money - ? WHERE phone = ?`, [info?.[0]?.money, withInfo.phone]);

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

            await connection.query(sql\_noti, [receiinfo?.[0]?.id, "Congrates! you received an reward of "+info?.[0]?.money+" from your friend " + senderinfo?.[0]?.code +".", '0', "Recharge"]);

*let* sql\_noti1 = "INSERT INTO notification SET recipient = ?, description = ?, isread = ?, noti\_type = ?";

*let* withdrdesc = "Amount of "+ info?.[0]?.money+ " have been transferred successfully.";

            await connection.query(sql\_noti1, [senderinfo?.[0]?.id, withdrdesc , "0", "Withdraw"]);

        }

        else{

*const* user = await getUserDataByPhone(info?.[0]?.phone)

            addUserAccountBalance({

                money: info[0].money,

                phone: user.phone,

                invite: user.invite

            });

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

            await connection.query(sql\_noti, [receiinfo?.[0]?.id, "Recharge of Amount "+info?.[0]?.money+" is Successfull. ", '0', "Recharge"]);

        }

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

            message: 'Successful application confirmation',

            status: true,

            datas: [],

        });

    }

    if (type == 'delete') {

        await connection.query(`UPDATE recharge SET status = 2 WHERE id = ?`, [id]);

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

            message: 'Cancellation successful',

            status: true,

            datas: [],

        });

    }

}

In this code snippet, an asynchronous function collo\_rechargeDuyet is defined that handles recharge confirmation and cancellation.

1. It first extracts auth, id, and type from the request object. If any of these values is missing, it returns a failure response.
2. For confirmation type, it updates the recharge status, fetches relevant information from the database, and performs different operations based on the type. If the type is 'wallet', it updates user balances, sends notifications, and if not, it updates user account balance and sends a recharge notification.
3. Finally, it returns a success response for both confirmation and cancellation types along with appropriate messages and status.

## **collo\_handlWithdraw**

This function is responsible for handling withdrawals made by users.

### Parameters:

* req: The request object containing information about the request made by the client.
* res: The response object used to send a response back to the client.

*const* collo\_handlWithdraw = *async* (*req*, *res*) *=>* {

    console.log("fired");

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

*let* id = *req*.body.id;

*let* type = *req*.body.type;

    if (!auth || !id || !type) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

    if (type == 'confirm') {

        await connection.query(`UPDATE withdraw SET status = 1 WHERE id = ?`, [id]);

*const* [winfo] = await connection.query(`SELECT \* FROM withdraw WHERE id = ?`, [id]);

*let* withInfo = winfo[0];

*const* [senderinfo] = await connection.query(`SELECT \* FROM users WHERE phone = ?`, [withInfo.phone]);

        if(withInfo.with\_type.trim() == 'transfer')

        {

            await connection.query(`UPDATE withdraw SET status = 1 WHERE id = ?`, [id]);

*const* [recharge] = await connection.query(`SELECT \* FROM recharge WHERE id\_order = ?`, [withInfo.id\_order]);

*let* rechInfo = recharge[0];

*const* [receiinfo] = await connection.query(`SELECT \* FROM users WHERE phone = ?`, [rechInfo.phone]);

            await connection.query(`UPDATE recharge SET status = 1 WHERE id\_order = ?`, [withInfo.id\_order]);

            await connection.query('UPDATE users SET money =  money - ? WHERE phone = ?', [withInfo.money, withInfo.phone]);

            await connection.query(`UPDATE users SET money = money + ? WHERE phone = ?`, [withInfo.money, rechInfo.phone]);

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

            await connection.query(sql\_noti, [receiinfo?.[0]?.id, "Congrates! you received an reward of "+rechInfo.money+" from your friend " + senderinfo?.[0]?.code +".", '0', "Recharge"]);

*let* sql\_noti1 = 'INSERT INTO notification SET recipient = ?, description = ?, isread = ?, noti\_type = ?';

            await connection.query(sql\_noti1, [senderinfo?.[0]?.id, "Amount of "+withInfo.money+ " have been transferred successfully.", '0', "Withdraw"]);

        }

        else

        {

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

            await connection.query(sql\_noti, [senderinfo?.[0]?.id, "Your withdraw of amoount "+withInfo.money+" approved my admin.", '0', "Withdraw"]);

        }

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

            message: 'Successful application confirmation',

            status: true,

            datas: [],

        });

    }

    if (type == 'delete') {

        await connection.query(`UPDATE withdraw SET status = 2 WHERE id = ?`, [id]);

*const* [info] = await connection.query(`SELECT \* FROM withdraw WHERE id = ?`, [id]);

        await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [info[0].money, info[0].phone]);

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

            message: 'Cancel successfully',

            status: true,

            datas: [],

        });

    }

}

The code defines an asynchronous function collo\_handlWithdraw that handles withdrawal requests. It first checks for authentication, id, and type in the request body. If any of them is missing, it responds with a failure message.

1. If the type is 'confirm', it updates the withdrawal status, fetches related information, processes transfer or approval accordingly, notifies the users involved, and responds with a success message.
2. If the type is 'delete', it updates the withdrawal status, adjusts the users' money balance, and responds with a cancellation success message.
3. It seems to be missing a definition or import of the connection variable, and the usage of timeNow needs to be defined to include the timestamp in the failure response object.

## **settings**

This function handles the settings related to the user account.

### Parameters:

* req: The request object containing information about the request made by the client.
* res: The response object used to send a response back to the client.

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

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

*let* type = *req*.body.type;

*let* value = *req*.body.value;

*const* [rows] = await connection.execute('SELECT `phone` FROM `users` WHERE `token` = ? AND veri = 1', [auth]);

    if (rows.length == 0) {

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

            message: 'Error',

            status: false,

        });

    }

    if (!type) {

*const* [point\_list] = await connection.execute('SELECT `telegram` FROM `point\_list` WHERE phone = ?', [rows[0].phone]);

*const* [settings] = await connection.execute('SELECT `telegram` FROM `admin`');

*let* telegram = settings[0].telegram;

*let* telegram2 = point\_list[0].telegram;

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

            message: 'Get success',

            status: true,

            telegram: telegram,

            telegram2: telegram2,

        });

    } else {

        await connection.execute('UPDATE `point\_list` SET telegram = ? WHERE phone = ?', [value ,rows[0].phone]);

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

            message: 'Successfully edited',

            status: true,

        });

    }

}

The provided code is an async function named settings that handles different cases based on the value of the type parameter received in the request body. Here is an explanation of the code:

1. It first extracts the auth, type, and value from the request.
2. It then executes a SQL query to retrieve a phone number based on the provided auth token.
3. If the query returns no rows, it sends a JSON response with an error message and status.
4. If type is empty or falsy, it fetches the Telegram details from the point\_list and admin tables and sends a JSON response with the retrieved data.
5. Otherwise, it updates the Telegram value in the point\_list table based on the provided value and phone number, then sends a successful update message.

## **middlewareDailyController**

This is the middleware function used to verify the authentication of an admin.

### Parameters:

* req: The request object containing information about the request made by the client.
* res: The response object used to send a response back to the client.
* next: The next function to be called in the middleware chain.

*const* middlewareDailyController = *async*(*req*, *res*, *next*) *=>* {

    // xác nhận token

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

    if (!auth) {

        return *res*.redirect("/login");

    }

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

    if (!rows) {

        return *res*.redirect("/login");

    }

    try {

        if (auth == rows[0].token && rows[0].status == 1) {

            if (rows[0].level == 2) {

                next();

            } else {

                return *res*.redirect("/home");

            }

        } else {

            return *res*.redirect("/login");

        }

    } catch (error) {

        return *res*.redirect("/login");

    }

}

The provided code defines an asynchronous middleware function in Node.js that checks the authorization token stored in the request cookies.

1. The function first retrieves the authorization token from the request cookies.
2. If no token is found (auth is falsy), it redirects the user to the '/login' page.
3. It then queries the database to fetch user records based on the received token, checking if the token is verified.
4. If no matching user records are found (rows is falsy), it redirects the user to the '/login' page.
5. Inside a try-catch block, it compares the retrieved token with the stored token in the first row of the query result and checks the user's status.
6. If the token and status are valid, it further checks if the user level is 2, allowing the execution to proceed by calling the 'next()' function.
7. If the user level is not 2, it redirects the user to the '/home' page.
8. If the token or status is invalid, or an error occurs during the checks, the user is redirected to the '/login' page.

## **statistical**

This function generates statistical information about the number of users and their activities.

### Parameters:

* req: The request object containing information about the request made by the client.
* res: The response object used to send a response back to the client.

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

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

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

*let* userInfo = user[0];

    // cấp dưới trực tiếp all

*const* [f1s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [userInfo.code]);

    // cấp dưới trực tiếp hôm nay

*let* f1\_today = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_time = f1s[i].time; // Mã giới thiệu f1

*let* check = (timerJoin(f1\_time) == timerJoin()) ? true : false;

        if(check) {

            f1\_today += 1;

        }

    }

    // tất cả cấp dưới hôm nay

*let* f\_all\_today = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* f1\_time = f1s[i].time; // time f1

*let* check\_f1 = (timerJoin(f1\_time) == timerJoin()) ? true : false;

        if(check\_f1) f\_all\_today += 1;

        // tổng f1 mời đc hôm nay

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code; // Mã giới thiệu f2

*const* f2\_time = f2s[i].time; // time f2

*let* check\_f2 = (timerJoin(f2\_time) == timerJoin()) ? true : false;

            if(check\_f2) f\_all\_today += 1;

            // tổng f2 mời đc hôm nay

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f2\_code]);

            for (*let* i = 0; i < f3s.length; i++) {

*const* f3\_code = f3s[i].code; // Mã giới thiệu f3

*const* f3\_time = f3s[i].time; // time f3

*let* check\_f3 = (timerJoin(f3\_time) == timerJoin()) ? true : false;

                if(check\_f3) f\_all\_today += 1;

*const* [f4s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f3\_code]);

                // tổng f3 mời đc hôm nay

                for (*let* i = 0; i < f4s.length; i++) {

*const* f4\_code = f4s[i].code; // Mã giới thiệu f4

*const* f4\_time = f4s[i].time; // time f4

*let* check\_f4 = (timerJoin(f4\_time) == timerJoin()) ? true : false;

                    if(check\_f4) f\_all\_today += 1;

                    // tổng f3 mời đc hôm nay

                }

            }

        }

    }

    // Tổng số f2

*let* f2 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        f2 += f2s.length;

    }

    // Tổng số f3

*let* f3 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code;

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f2\_code]);

            if(f3s.length > 0) f3 += f3s.length;

        }

    }

    // Tổng số f4

*let* f4 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code;

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f2\_code]);

            for (*let* i = 0; i < f3s.length; i++) {

*const* f3\_code = f3s[i].code;

*const* [f4s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f3\_code]);

                if(f4s.length > 0) f4 += f4s.length;

            }

        }

    }

    // const [recharge] = await connection.query('SELECT SUM(`money`) as total FROM recharge WHERE phone = ? AND status = 1 ', [phone]);

    // const [withdraw] = await connection.query('SELECT SUM(`money`) as total FROM withdraw WHERE phone = ? AND status = 1 ', [phone]);

    // const [bank\_user] = await connection.query('SELECT \* FROM user\_bank WHERE phone = ? ', [phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: user,

        f1: f1s.length,

        f2: f2,

        f3: f3,

        f4: f4,

    });

}

The code defines an asynchronous function named statistical that takes req and res as parameters.

1. It fetches the auth cookie from the request and queries the database to get user information based on the token stored in the cookie.
2. It then performs multiple queries to gather information about different levels of referrals and their counts for the current day.
3. There are loops that iterate through the referral structure up to the 4th level to calculate total counts at different levels.
4. The logic for checking and incrementing counts based on the timer is also present within the loops.
5. Finally, there are additional queries commented out that fetch information about recharges, withdraws, and bank details, but not used in the current response.
6. The function responds with a JSON object containing success status and various counts related to different referral levels.

## **formateT**

This function formats a number to have leading zero if it's less than 10.

### Parameters:

* params: The number to be formatted.

*function* formateT(*params*) {

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

    return result;

    }

The given code defines a function named formatT that takes an integer parameter params and returns a string.

1. The result is calculated using a ternary operator which checks if the params is less than 10 or not.
2. If the params is less than 10, then it prepends a '0' before the value using string concatenation.
3. However, the optimized code uses fmt.Sprintf to format the integer with leading zeros so that it always returns a two-digit string.
4. Finally, the formatted result is returned.

## **timerJoin**

This function returns the current time or the time derived from a given parameter.

### Parameters:

* params (optional): The timestamp to be formatted. If not provided, the current time will be used.
* addHours (optional): The number of hours to add to the time.

*function* timerJoin(*params* = '', *addHours* = 0) {

*let* date = '';

        if (*params*) {

            date = new *Date*(*Number*(*params*));

        } else {

            date = new *Date*();

        }

        date.setHours(date.getHours() + *addHours*);

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

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

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

*let* hours = date.getHours() % 12;

        hours = hours === 0 ? 12 : hours;

*let* ampm = date.getHours() < 12 ? "AM" : "PM";

*let* minutes = formateT(date.getMinutes());

*let* seconds = formateT(date.getSeconds());

        return years + '-' + months + '-' + days + ' ' + hours + ':' + minutes + ':' + seconds + ' ' + ampm;

    }

The provided code defines a function timerJoin that takes two parameters, params (defaulted to an empty string) and addHours (defaulted to 0). It then creates a new Date object based on the input params (if provided) or the current date. It then adds the specified addHours to the hours of the date.

1. Next, it extracts the year, month, day, hour (in 12-hour format along with AM/PM), minutes, and seconds from the date. It uses a helper function formatT to ensure that single-digit values are formatted with a leading zero.
2. Finally, it constructs and returns a string in the format 'YYYY-MM-DD HH:MM:SS AM/PM' by joining the extracted values.
3. The provided code had a syntax error in the function call to formatT, which is fixed by updating formateT to formatT. Additionally, the missing definition of the formatT helper function is added to the optimized code.

## **userInfo**

This function retrieves information about a specific user.

### Parameters:

* req: The request object containing information about the request made by the client.
* res: The response object used to send a response back to the client.

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

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

*let* phone = *req*.params.phone;

    if (!phone) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE phone = ? ', [phone]);

*const* [auths] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0 || auths.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* { token, password, otp, level,...userInfo } = user[0];

    if (auths[0].phone != userInfo.ctv) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

    // cấp dưới trực tiếp all

*const* [f1s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [userInfo.code]);

    // cấp dưới trực tiếp hôm nay

*let* f1\_today = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_time = f1s[i].time; // Mã giới thiệu f1

*let* check = (timerJoin(f1\_time) == timerJoin()) ? true : false;

        if(check) {

            f1\_today += 1;

        }

    }

    // tất cả cấp dưới hôm nay

*let* f\_all\_today = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* f1\_time = f1s[i].time; // time f1

*let* check\_f1 = (timerJoin(f1\_time) == timerJoin()) ? true : false;

        if(check\_f1) f\_all\_today += 1;

        // tổng f1 mời đc hôm nay

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code; // Mã giới thiệu f2

*const* f2\_time = f2s[i].time; // time f2

*let* check\_f2 = (timerJoin(f2\_time) == timerJoin()) ? true : false;

            if(check\_f2) f\_all\_today += 1;

            // tổng f2 mời đc hôm nay

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f2\_code]);

            for (*let* i = 0; i < f3s.length; i++) {

*const* f3\_code = f3s[i].code; // Mã giới thiệu f3

*const* f3\_time = f3s[i].time; // time f3

*let* check\_f3 = (timerJoin(f3\_time) == timerJoin()) ? true : false;

                if(check\_f3) f\_all\_today += 1;

*const* [f4s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f3\_code]);

                // tổng f3 mời đc hôm nay

                for (*let* i = 0; i < f4s.length; i++) {

*const* f4\_code = f4s[i].code; // Mã giới thiệu f4

*const* f4\_time = f4s[i].time; // time f4

*let* check\_f4 = (timerJoin(f4\_time) == timerJoin()) ? true : false;

                    if(check\_f4) f\_all\_today += 1;

                    // tổng f3 mời đc hôm nay

                }

            }

        }

    }

    // Tổng số f2

*let* f2 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        f2 += f2s.length;

    }

    // Tổng số f3

*let* f3 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code;

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f2\_code]);

            if(f3s.length > 0) f3 += f3s.length;

        }

    }

    // Tổng số f4

*let* f4 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code;

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f2\_code]);

            for (*let* i = 0; i < f3s.length; i++) {

*const* f3\_code = f3s[i].code;

*const* [f4s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f3\_code]);

                if(f4s.length > 0) f4 += f4s.length;

            }

        }

    }

*const* [recharge] = await connection.query('SELECT SUM(`money`) as total FROM recharge WHERE phone = ? AND status = 1 ', [phone]);

*const* [withdraw] = await connection.query('SELECT SUM(`money`) as total FROM withdraw WHERE phone = ? AND status = 1 ', [phone]);

*const* [bank\_user] = await connection.query('SELECT \* FROM user\_bank WHERE phone = ? ', [phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: userInfo,

        total\_r: recharge,

        total\_w: withdraw,

        f1: f1s.length,

        f2: f2,

        f3: f3,

        f4: f4,

        bank\_user: bank\_user,

    });

}

### Example:

The provided code defines an asynchronous function userInfo that handles a request and response. It first extracts auth from req.cookies and phone from req.params. If phone is missing, it returns a 'Failed' response with status and a timestamp.

1. It then queries the database to fetch user and auths based on the phone number and token, respectively. If no data is found, it returns a similar 'Failed' response.
2. Next, it extracts specific properties from the user object and checks if auths[0].phone matches userInfo.ctv. If not, it returns a 'Failed' response.
3. The code then proceeds to fetch data about the network structure of users and performs some calculations. Finally, it queries the database to get recharge, withdraw, and bank details of the user and constructs a JSON response containing various gathered data.

## **infoCtv**

This function retrieves information about a specific collaborator (CTV).

### Parameters:

* req: The request object containing information about the request made by the client.
* res: The response object used to send a response back to the client.

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

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

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0) {

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

            message: 'Phone Error',

            status: false,

        });

    }

*let* userInfo = user[0];

*let* phone = userInfo.phone;

    // cấp dưới trực tiếp all

*const* [f1s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [userInfo.code]);

    // cấp dưới trực tiếp hôm nay

*let* f1\_today = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_time = f1s[i].time; // Mã giới thiệu f1

*let* check = (timerJoin(f1\_time) == timerJoin()) ? true : false;

        if(check) {

            f1\_today += 1;

        }

    }

    // tất cả cấp dưới hôm nay

*let* f\_all\_today = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* f1\_time = f1s[i].time; // time f1

*let* check\_f1 = (timerJoin(f1\_time) == timerJoin()) ? true : false;

        if(check\_f1) f\_all\_today += 1;

        // tổng f1 mời đc hôm nay

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code; // Mã giới thiệu f2

*const* f2\_time = f2s[i].time; // time f2

*let* check\_f2 = (timerJoin(f2\_time) == timerJoin()) ? true : false;

            if(check\_f2) f\_all\_today += 1;

            // tổng f2 mời đc hôm nay

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f2\_code]);

            for (*let* i = 0; i < f3s.length; i++) {

*const* f3\_code = f3s[i].code; // Mã giới thiệu f3

*const* f3\_time = f3s[i].time; // time f3

*let* check\_f3 = (timerJoin(f3\_time) == timerJoin()) ? true : false;

                if(check\_f3) f\_all\_today += 1;

*const* [f4s] = await connection.query('SELECT `phone`, `code`,`invite`, `time` FROM users WHERE `invite` = ? ', [f3\_code]);

                // tổng f3 mời đc hôm nay

                for (*let* i = 0; i < f4s.length; i++) {

*const* f4\_code = f4s[i].code; // Mã giới thiệu f4

*const* f4\_time = f4s[i].time; // time f4

*let* check\_f4 = (timerJoin(f4\_time) == timerJoin()) ? true : false;

                    if(check\_f4) f\_all\_today += 1;

                    // tổng f3 mời đc hôm nay

                }

            }

        }

    }

    // Tổng số f2

*let* f2 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        f2 += f2s.length;

    }

    // Tổng số f3

*let* f3 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code;

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f2\_code]);

            if(f3s.length > 0) f3 += f3s.length;

        }

    }

    // Tổng số f4

*let* f4 = 0;

    for (*let* i = 0; i < f1s.length; i++) {

*const* f1\_code = f1s[i].code; // Mã giới thiệu f1

*const* [f2s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f1\_code]);

        for (*let* i = 0; i < f2s.length; i++) {

*const* f2\_code = f2s[i].code;

*const* [f3s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f2\_code]);

            for (*let* i = 0; i < f3s.length; i++) {

*const* f3\_code = f3s[i].code;

*const* [f4s] = await connection.query('SELECT `phone`, `code`,`invite` FROM users WHERE `invite` = ? ', [f3\_code]);

                if(f4s.length > 0) f4 += f4s.length;

            }

        }

    }

*const* [list\_mem] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 1 AND veri = 1 ', [phone]);

*const* [list\_mem\_baned] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 2 AND veri = 1 ', [phone]);

*let* total\_recharge = 0;

*let* total\_withdraw = 0;

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [recharge] = await connection.query('SELECT SUM(money) as money FROM recharge WHERE phone = ? AND status = 1 ', [phone]);

*const* [withdraw] = await connection.query('SELECT SUM(money) as money FROM withdraw WHERE phone = ? AND status = 1 ', [phone]);

        if (recharge[0].money) {

            total\_recharge += *Number*(recharge[0].money);

        }

        if (withdraw[0].money) {

            total\_withdraw += *Number*(withdraw[0].money);

        }

    }

*let* total\_recharge\_today = 0;

*let* total\_withdraw\_today = 0;

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [recharge\_today] = await connection.query('SELECT `money`, `time` FROM recharge WHERE phone = ? AND status = 1 ', [phone]);

*const* [withdraw\_today] = await connection.query('SELECT `money`, `time` FROM withdraw WHERE phone = ? AND status = 1 ', [phone]);

        for (*let* i = 0; i < recharge\_today.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(recharge\_today[i].time);

            if (time == today) {

                total\_recharge\_today += recharge\_today[i].money;

            }

        }

        for (*let* i = 0; i < withdraw\_today.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(withdraw\_today[i].time);

            if (time == today) {

                total\_withdraw\_today += withdraw\_today[i].money;

            }

        }

    }

*let* win = 0;

*let* loss = 0;

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [wins] = await connection.query('SELECT `money`, `time` FROM minutes\_1 WHERE phone = ? AND status = 1 ', [phone]);

*const* [losses] = await connection.query('SELECT `money`, `time` FROM minutes\_1 WHERE phone = ? AND status = 2 ', [phone]);

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

*let* today = timerJoin();

*let* time = timerJoin(wins[i].time);

            if (time == today) {

                win += wins[i].money;

            }

        }

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

*let* today = timerJoin();

*let* time = timerJoin(losses[i].time);

            if (time == today) {

                loss += losses[i].money;

            }

        }

    }

*let* list\_mems = [];

*const* [list\_mem\_today] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 1 AND veri = 1 ', [phone]);

    for (*let* i = 0; i < list\_mem\_today.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(list\_mem\_today[i].time);

        if (time == today) {

*const* [phone\_invites] = await connection.query('SELECT `phone` FROM users WHERE code = ? ', [list\_mem\_today[i].invite]);

*let* phone\_invite = phone\_invites[0].phone;

*let* data = {

                ...list\_mem\_today[i],

                phone\_invite: phone\_invite,

            };

            list\_mems.push(data);

        }

    }

*const* [point\_list] = await connection.query('SELECT \* FROM point\_list WHERE phone = ? ', [phone]);

*let* moneyCTV = point\_list[0].money;

*let* list\_recharge\_news = [];

*let* list\_withdraw\_news = [];

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [recharge\_today] = await connection.query('SELECT `id`, `status`, `type`,`phone`, `money`, `time` FROM recharge WHERE phone = ? AND status = 1 ', [phone]);

*const* [withdraw\_today] = await connection.query('SELECT `id`, `status`,`phone`, `money`, `time` FROM withdraw WHERE phone = ? AND status = 1 ', [phone]);

        for (*let* i = 0; i < recharge\_today.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(recharge\_today[i].time);

            if (time == today) {

                list\_recharge\_news.push(recharge\_today[i]);

            }

        }

        for (*let* i = 0; i < withdraw\_today.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(withdraw\_today[i].time);

            if (time == today) {

                list\_withdraw\_news.push(withdraw\_today[i]);

            }

        }

    }

*const* [redenvelopes\_used] = await connection.query('SELECT \* FROM redenvelopes\_used WHERE phone = ? ', [phone]);

*let* redenvelopes\_used\_today = [];

    for (*let* i = 0; i < redenvelopes\_used.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(redenvelopes\_used[i].time);

        if (time == today) {

            redenvelopes\_used\_today.push(redenvelopes\_used[i]);

        }

    }

*const* [financial\_details] = await connection.query('SELECT \* FROM financial\_details WHERE phone = ? ', [phone]);

*let* financial\_details\_today = [];

    for (*let* i = 0; i < financial\_details.length; i++) {

*let* today = timerJoin();

*let* time = timerJoin(financial\_details[i].time);

        if (time == today) {

            financial\_details\_today.push(financial\_details[i]);

        }

    }

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: user,

        f1: f1s.length,

        f2: f2,

        f3: f3,

        f4: f4,

        list\_mems: list\_mems,

        total\_recharge: total\_recharge,

        total\_withdraw: total\_withdraw,

        total\_recharge\_today: total\_recharge\_today,

        total\_withdraw\_today: total\_withdraw\_today,

        list\_mem\_baned: list\_mem\_baned.length,

        win: win,

        loss: loss,

        list\_recharge\_news: list\_recharge\_news,

        list\_withdraw\_news: list\_withdraw\_news,

        moneyCTV: moneyCTV,

        redenvelopes\_used: redenvelopes\_used\_today,

        financial\_details\_today: financial\_details\_today,

    });

}

The provided code defines an async function named infoCtv that takes req and res as parameters. It queries a database using connection.query to retrieve user information based on a token stored in cookies.

1. If the user information is found, it then performs various calculations related to a multi-level referral system, total recharge and withdrawal amounts, wins and losses, and other financial details for the user.
2. After processing the data, it returns a JSON response with a success message, status, and various computed data points such as f1, f2, f3, f4, total recharge, total withdrawal, etc.

### **infoCtv2**

This function retrieves information about a user's CTVs, including any relevant news such as recharge and withdrawal transactions made by the CTV's members. The function takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

*const* infoCtv2 = *async*(*req*, *res*) *=>* {

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

*const* timeDate = *req*.body.timeDate;

*function* formateT(*params*) {

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

    return result;

    }

*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;

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0) {

        return res.status(200).json({

            message: 'Phone Error',

            status: false,

        });

    }

*let* userInfo = user[0];

*let* phone = userInfo.phone;

*const* [list\_mem] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 1 AND veri = 1 ', [phone]);

*let* list\_mems = [];

*const* [list\_mem\_today] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 1 AND veri = 1 ', [phone]);

    for (*let* i = 0; i < list\_mem\_today.length; i++) {

*let* today = timeDate;

*let* time = timerJoin(list\_mem\_today[i].time);

        if (time == today) {

*const* [phone\_invites] = await connection.query('SELECT `phone` FROM users WHERE code = ? ', [list\_mem\_today[i].invite]);

*let* phone\_invite = phone\_invites[0].phone;

*let* data = {

                ...list\_mem\_today[i],

                phone\_invite: phone\_invite,

            };

            list\_mems.push(data);

        }

    }

*let* list\_recharge\_news = [];

*let* list\_withdraw\_news = [];

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [recharge\_today] = await connection.query('SELECT `id`, `status`, `type`,`phone`, `money`, `time` FROM recharge WHERE phone = ? AND status = 1 ', [phone]);

*const* [withdraw\_today] = await connection.query('SELECT `id`, `status`,`phone`, `money`, `time` FROM withdraw WHERE phone = ? AND status = 1 ', [phone]);

        for (*let* i = 0; i < recharge\_today.length; i++) {

*let* today = timeDate;

*let* time = timerJoin(recharge\_today[i].time);

            if (time == today) {

                list\_recharge\_news.push(recharge\_today[i]);

            }

        }

        for (*let* i = 0; i < withdraw\_today.length; i++) {

*let* today = timeDate;

*let* time = timerJoin(withdraw\_today[i].time);

            if (time == today) {

                list\_withdraw\_news.push(withdraw\_today[i]);

            }

        }

    }

*const* [redenvelopes\_used] = await connection.query('SELECT \* FROM redenvelopes\_used WHERE phone = ? ', [phone]);

*let* redenvelopes\_used\_today = [];

    for (*let* i = 0; i < redenvelopes\_used.length; i++) {

*let* today = timeDate;

*let* time = timerJoin(redenvelopes\_used[i].time);

        if (time == today) {

            redenvelopes\_used\_today.push(redenvelopes\_used[i]);

        }

    }

*const* [financial\_details] = await connection.query('SELECT \* FROM financial\_details WHERE phone = ? ', [phone]);

*let* financial\_details\_today = [];

    for (*let* i = 0; i < financial\_details.length; i++) {

*let* today = timeDate;

*let* time = timerJoin(financial\_details[i].time);

        if (time == today) {

            financial\_details\_today.push(financial\_details[i]);

        }

    }

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: user,

        list\_mems: list\_mems,

        list\_recharge\_news: list\_recharge\_news,

        list\_withdraw\_news: list\_withdraw\_news,

        redenvelopes\_used: redenvelopes\_used\_today,

        financial\_details\_today: financial\_details\_today,

    });

}

The provided code is an asynchronous function infoCtv2 that fetches some data from a database based on certain conditions and returns a JSON response. Here is a breakdown of the code:

1. auth and timeDate are extracted from req object.
2. Two helper functions formateT and timerJoin are defined to format and manipulate date/time values.
3. The function queries the database to retrieve user information based on the provided authentication token (auth).
4. If no user is found, it returns an error response.
5. It then queries the database for various types of data related to the user identified by their phone number.
6. Iterates over the fetched records to filter data based on the given timeDate.
7. Finally, it constructs a JSON response containing the fetched data and returns it.

### **createBonus**

This function is responsible for creating a bonus or gift for a user. It generates a random string for the bonus ID and deducts the corresponding money from the user's account. The function takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

*const* randomString = (*length*) *=>* {

*var* result = '';

*var* characters = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';

*var* charactersLength = characters.length;

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

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

                charactersLength));

        }

        return result;

    }

*function* formateT(*params*) {

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

    return result;

    }

*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;

    }

*const* d = new *Date*();

*const* time = d.getTime();

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

*let* money = *req*.body.money;

    if (!money || !auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* userInfo = user[0];

*const* [point\_list] = await connection.query('SELECT \* FROM point\_list WHERE phone = ? ', [userInfo.phone]);

    if (point\_list.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* ctv = point\_list[0];

    if (ctv.money - money >= 0) {

*let* id\_redenvelops = randomString(16);

        await connection.execute('UPDATE `point\_list` SET money = money - ? WHERE phone = ?', [money ,ctv.phone]);

*let* sql = `INSERT INTO redenvelopes SET id\_redenvelope = ?, phone = ?, money = ?, used = ?, amount = ?, status = ?, time = ?`;

        await connection.query(sql, [id\_redenvelops, userInfo.phone, money, 1, 1, 0, time]);

*const* [point\_list] = await connection.query('SELECT `money` FROM point\_list WHERE phone = ? ', [userInfo.phone]);

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

            message: 'Successful gift creation',

            status: true,

            id: id\_redenvelops,

            money: point\_list[0].money,

        });

    } else {

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

            message: 'The balance is not enough to create gifts',

            status: false,

        });

    }

}

The code defines an asynchronous function createBonus that handles the creation of a bonus gift based on certain conditions.

Explanation of the code:

1. A helper function randomString is defined to generate a random alphanumeric string of a specified length.
2. Another helper function formateT is defined to format a parameter by appending a '0' if it's less than 10.
3. There is a function timerJoin that takes parameters for date and additional hours to adjust the date and formats it into a specific format.
4. The current time is obtained using new Date().
5. The auth and money values are extracted from request objects req.cookies and req.body.
6. Various checks are performed on money and auth, and appropriate error responses are returned if conditions fail.
7. Queries are executed to fetch user details and point lists from a database based on the provided authentication token.
8. Further checks are made on the available balance for the user to create a gift, and if the balance is sufficient, a new gift entry is added to the database.
9. Finally, the updated balance and gift information are returned as a successful response, or an error message is returned if the balance is insufficient.

### **listRedenvelops**

This function retrieves a list of red envelopes for a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* userInfo = user[0];

*let* [redenvelopes] = await connection.query('SELECT \* FROM redenvelopes WHERE phone = ? ORDER BY id DESC', [userInfo.phone]);

    return *res*.status(200).json({

        message: 'Successful change',

        status: true,

        redenvelopes: redenvelopes,

    });

}

The provided code is an asynchronous function named listRedenvelops that takes req and res as parameters.

1. It first retrieves the authorization token from cookies and then queries the database to fetch user information based on the token.
2. If no user is found, it returns a JSON response with a message of 'Failed', status set to false, and a timestamp.
3. Otherwise, it retrieves the user's information, queries the redenvelopes table based on the user's phone number, orders the results by ID in descending order, and returns a JSON response with a message of 'Successful change', status set to true, and the fetched redenvelopes.

### **listMember**

This function retrieves a list of members for a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*let* {pageno, limit } = *req*.body;

*let* [checkInfo] = await connection.execute('SELECT \* FROM users WHERE token = ?', [auth]);

    if(checkInfo.length == 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

*let* userInfo = checkInfo[0];

    if (!pageno || !limit) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (pageno < 0 || limit < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

*const* [users] = await connection.query(`SELECT id\_user, phone, money, total\_money, status, time FROM users WHERE veri = 1 AND level = 0 AND ctv = ? ORDER BY id DESC LIMIT ${pageno}, ${limit} `, [userInfo.phone]);

*const* [total\_users] = await connection.query(`SELECT \* FROM users WHERE veri = 1 AND level = 0 AND ctv = ? `, [userInfo.phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: users,

        page\_total: Math.ceil(total\_users.length / limit)

    });

}

The provided code defines an asynchronous function listMember that handles a request and response in a Node.js environment. Here's a breakdown of the code:

1. It extracts the auth, pageno, and limit variables from the request object.
2. It queries the database to check for user information based on the provided auth token.
3. If no user information is found, it returns a JSON response indicating no data was found.
4. It then checks if pageno or limit is missing or negative and returns a similar JSON response if so.
5. Next, it executes SQL queries to fetch a list of users meeting certain criteria based on pageno and limit.
6. It calculates the total number of users based on the criteria.
7. Finally, it sends a JSON response with the list of users, success message, and the total number of pages based on the provided limit.

### **listRechargeP**

This function retrieves a list of recharge transactions for a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

*const* [bank\_user] = await connection.query('SELECT \* FROM bank\_recharge WHERE phone = ? ', [user[0].phone]);

    if (user.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* userInfo = user[0];

*const* [list\_mem] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 1 AND veri = 1 ', [userInfo.phone]);

*let* list\_recharge\_news = [];

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [recharge\_today] = await connection.query('SELECT \* FROM recharge WHERE phone = ? ORDER BY id DESC LIMIT 100', [phone]);

        for (*let* i = 0; i < recharge\_today.length; i++) {

            list\_recharge\_news.push(recharge\_today[i]);

        }

    }

    return *res*.status(200).json({

        message: 'Failed',

        status: true,

        list\_recharge\_news: list\_recharge\_news,

        timeStamp: timeNow,

        pay\_status: bank\_user[0].colloborator\_action

    });

}

The given code is an asynchronous function named listRechargeP that takes req and res as parameters.

Explanation of the code:

1. It retrieves the authentication token from the request cookies.
2. Queries the database to fetch user data based on the received token.
3. Queries the database to find bank recharge information based on the user's phone number.
4. Checks if the user exists, and if not, returns a response with an error message.
5. Retrieves user information.
6. Queries the database to get a list of users with specific conditions and populates list\_recharge\_news with relevant data.
7. Formats and returns a JSON response with details including message, status, list\_recharge\_news, timeStamp, and pay\_status extracted from the database queries.

### **listWithdrawP**

This function retrieves a list of withdrawal transactions for a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*const* [user] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

*const* [bank\_user] = await connection.query('SELECT \* FROM bank\_recharge WHERE phone = ? ', [user[0].phone]);

    if (user.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* userInfo = user[0];

*const* [list\_mem] = await connection.query('SELECT \* FROM users WHERE ctv = ? AND status = 1 AND veri = 1 ', [userInfo.phone]);

*let* list\_withdraw\_news = [];

    for (*let* i = 0; i < list\_mem.length; i++) {

*let* phone = list\_mem[i].phone;

*const* [withdraw\_today] = await connection.query('SELECT \* FROM withdraw WHERE phone = ? ORDER BY id DESC LIMIT 100', [phone]);

        for (*let* i = 0; i < withdraw\_today.length; i++) {

            list\_withdraw\_news.push(withdraw\_today[i]);

        }

    }

    return *res*.status(200).json({

        message: 'Failed',

        status: true,

        list\_withdraw\_news: list\_withdraw\_news,

        timeStamp: timeNow,

        pay\_status: bank\_user[0].colloborator\_action

    });

}

1. The provided code defines an asynchronous function listWithdrawP that expects req and res as parameters. It then retrieves user and bank information from the database based on the authentication token stored in the request cookies. If the user is not found (user length equals 0), it responds with a JSON object containing a failed message and status along with the current timestamp.
2. Subsequently, it fetches a list of members related to the user, iterates over these members, and for each member, retrieves recent withdrawal records from the database and adds them to list\_withdraw\_news. Finally, it constructs a JSON response object with a success message, list\_withdraw\_news, current timestamp, and the payment status of the bank user.

### **listRechargeMem**

This function retrieves a list of recharge transactions for a member of a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*let* phone = *req*.params.phone;

*let* {pageno, limit } = *req*.body;

    if (!pageno || !limit) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (pageno < 0 || limit < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!phone) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE phone = ? ', [phone]);

*const* [auths] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0 || auths.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* { token, password, otp, level,...userInfo } = user[0];

    if (auths[0].phone != userInfo.ctv) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [recharge] = await connection.query(`SELECT \* FROM recharge WHERE phone = ? ORDER BY id DESC LIMIT ${pageno}, ${limit} `, [phone]);

*const* [total\_users] = await connection.query(`SELECT \* FROM recharge WHERE phone = ?`, [phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: recharge,

        page\_total: Math.ceil(total\_users.length / limit)

    });

}

The provided code defines an asynchronous function listRechargeMem that handles a GET request. It extracts authentication data, phone number, page number, and limit from the request object. The function then checks if both pageno and limit are present and positive. If not, it returns a response indicating no more data available.

1. Next, it queries the database to retrieve user information and authentication details using the provided phone and auth. If either the user or authentication records are not found, it returns a failure response.
2. It then ensures that the user's phone matches the ctv value in the auth record; otherwise, it returns a failure response.
3. Subsequently, it queries the database for recharge data based on the phone, sorting it in descending order, and limiting the results by page number and limit. Additionally, it fetches the total number of users for the given phone. Finally, it constructs a success response with the recharge data, total pages, and sends it back.

### **listWithdrawMem**

This function retrieves a list of withdrawal transactions for a member of a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*let* phone = *req*.params.phone;

*let* {pageno, limit } = *req*.body;

    if (!pageno || !limit) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (pageno < 0 || limit < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!phone) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE phone = ? ', [phone]);

*const* [auths] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0 || auths.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* { token, password, otp, level,...userInfo } = user[0];

    if (auths[0].phone != userInfo.ctv) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [withdraw] = await connection.query(`SELECT \* FROM withdraw WHERE phone = ? ORDER BY id DESC LIMIT ${pageno}, ${limit} `, [phone]);

*const* [total\_users] = await connection.query(`SELECT \* FROM withdraw WHERE phone = ?`, [phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: withdraw,

        page\_total: Math.ceil(total\_users.length / limit)

    });

}

The provided code is an asynchronous function named listWithdrawMem that handles withdrawal list requests based on certain conditions.

Here's a breakdown of the code:

1. It extracts data from request objects like req.body and req.params.
2. Checks if pageno and limit exist and are greater than or equal to 0. If not, it returns a response indicating no more data.
3. Performs various database queries using connection.query to fetch user and authentication details.
4. Verifies user and authentication information and returns failure response if not found or invalid.
5. Executes a query to fetch withdrawal information based on the provided pageno and limit.
6. Calculates the total number of users and returns a response with success message along with withdrawal data and total pages based on pagination size.

### **listRedenvelope**

This function retrieves a list of red envelopes used by a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*let* phone = *req*.params.phone;

*let* {pageno, limit } = *req*.body;

    if (!pageno || !limit) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (pageno < 0 || limit < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!phone) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE phone = ? ', [phone]);

*const* [auths] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0 || auths.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* { token, password, otp, level,...userInfo } = user[0];

    if (auths[0].phone != userInfo.ctv) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [redenvelopes\_used] = await connection.query(`SELECT \* FROM redenvelopes\_used WHERE phone\_used = ? ORDER BY id DESC LIMIT ${pageno}, ${limit} `, [phone]);

*const* [total\_users] = await connection.query(`SELECT \* FROM redenvelopes\_used WHERE phone\_used = ?`, [phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: redenvelopes\_used,

        page\_total: Math.ceil(total\_users.length / limit)

    });

}

Here is a breakdown of the provided code:

1. The function listRedenvelope is an asynchronous function accepting req and res parameters for request and response handling.
2. It first extracts auth from cookies, phone from request parameters, and pageno and limit from the request body.
3. It then checks for missing or invalid data and returns appropriate JSON responses if any condition fails.
4. It queries the database for user and auth details based on the provided phone and auth.
5. If the user or auth does not exist or match certain conditions, it returns failure responses.
6. It further queries the database for redenvelopes used and total users based on the phone number, calculates page total, and returns a success response with relevant data.

### **listBet**

This function retrieves a list of bets made by a user. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*let* phone = *req*.params.phone;

*let* {pageno, limit } = *req*.body;

    if (!pageno || !limit) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (pageno < 0 || limit < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!phone) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT \* FROM users WHERE phone = ? ', [phone]);

*const* [auths] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (user.length == 0 || auths.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* { token, password, otp, level,...userInfo } = user[0];

    if (auths[0].phone != userInfo.ctv) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [listBet] = await connection.query(`SELECT \* FROM minutes\_1 WHERE phone = ? AND status != 0 ORDER BY id DESC LIMIT ${pageno}, ${limit} `, [phone]);

*const* [total\_users] = await connection.query(`SELECT \* FROM minutes\_1 WHERE phone = ? AND status != 0`, [phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: listBet,

        page\_total: Math.ceil(total\_users.length / limit)

    });

}

This code defines an asynchronous function listBet that handles API requests. It expects two parameters, req for request object and res for response object.

1. The function first extracts auth, phone, pageno, and limit from the request object.
2. It then checks if pageno or limit is missing or less than 0 and returns an error response if the conditions are met.
3. Next, it queries the database to fetch user and auth information based on the provided phone and auth.
4. If the user or auth is not found, it returns a failure response.
5. It then further checks the relationship between the authenticated user and the phone number's associated user.
6. Afterwards, it executes queries to retrieve betting data and total user count data from the database.
7. Finally, it sends a successful response with the fetched betting data along with pagination information.

### **buffMoney**

This function adds or subtracts money from a user's account. It takes the following parameters:

* req: The request object containing user data
* res: The response object used to send the response back to the client

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

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

*let* phone = *req*.body.username;

*let* select = *req*.body.select;

*let* money = *req*.body.money;

    if (!phone || !select || !money) {

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

            message: 'Fail',

            status: false,

        });

    }

*const* [users] = await connection.query('SELECT \* FROM users WHERE phone = ? ', [phone]);

*const* [auths] = await connection.query('SELECT \* FROM users WHERE token = ? ', [auth]);

    if (users.length == 0) {

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

            message: 'Account does not exist',

            status: false,

        });

    }

*let* userInfo = users[0];

*let* authInfo = auths[0];

*const* [point\_list] = await connection.query('SELECT `money\_us` FROM point\_list WHERE phone = ? ', [authInfo.phone]);

*let* check = point\_list[0].money\_us;

    if (select == '1') {

        if (check - money >= 0) {

*const* d = new *Date*();

*const* time = d.getTime();

            await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [money, userInfo.phone]);

            await connection.query('UPDATE point\_list SET money\_us = money\_us - ? WHERE phone = ? ', [money, authInfo.phone]);

*let* sql = 'INSERT INTO financial\_details SET phone = ?, phone\_used = ?, money = ?, type = ?, time = ?';

            await connection.query(sql, [authInfo.phone, userInfo.phone, money, '1', time]);

*const* [moneyN] = await connection.query('SELECT `money\_us` FROM point\_list WHERE phone = ? ', [authInfo.phone]);

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

                message: 'Success',

                status: true,

                money: moneyN[0].money\_us,

            });

        } else {

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

                message: 'Insufficient balance',

                status: false,

            });

        }

    } else {

*const* d = new *Date*();

*const* time = d.getTime();

        await connection.query('UPDATE users SET money = money - ? WHERE phone = ? ', [money, userInfo.phone]);

        await connection.query('UPDATE point\_list SET money = money + ? WHERE phone = ? ', [money, authInfo.phone]);

*let* sql = 'INSERT INTO financial\_details SET phone = ?, phone\_used = ?, money = ?, type = ?, time = ?';

        await connection.query(sql, [authInfo.phone, userInfo.phone, money, '2', time]);

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

            message: 'Success',

            status: true,

        });

    }

}

This code defines an asynchronous function buffMoney that takes req and res as parameters. It retrieves data from the request body such as username, select, and money and checks if any of them are falsy. If any of them is falsy, it returns a response with a failure message.

1. Then, it queries the database to check for a user with a specific phone number and token. If the user does not exist, it returns a response with a corresponding message.
2. Based on the select value, it performs different operations. If select is '1', it checks if the user has enough balance, updates the user's and point list's money values accordingly, inserts a financial transaction record, and returns a success response with the updated money balance. If select is not '1', it updates the user's and point list's money values in a different way, inserts a financial transaction record, and returns a success response.