**Documentation for adminController.js**

This file serves as a controller for managing various administrative features within the application. It handles tasks such as rendering specific admin pages, processing user requests, and interacting with the database.

**Functions Overview**

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

**Page Rendering Functions**

**adminPage**

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

    return *res*.render("manage/index.ejs");

}

This function renders the main administration page at the path /manage/index.ejs.

**Adminmainpage**

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

    return *res*.render("manage/dashboard.ejs");

}

Renders the admin dashboard page at /manage/dashboard.ejs.

**adminPage3**

*const* adminPage3 = *async* (*req*, *res*) *=>* {

    return *res*.render("manage/a-index-bet/index3.ejs");

}

Displays page 3 of the admin management system at /manage/a-index-bet/index3.ejs.

**adminPage5**

*const* adminPage5 = *async* (*req*, *res*) *=>* {

    return *res*.render("manage/a-index-bet/index5.ejs");

}

Renders page 5 of the admin management system at /manage/a-index-bet/index5.ejs.

**adminPage10**

*const* adminPage10 = *async* (*req*, *res*) *=>* {

    return *res*.render("manage/a-index-bet/index10.ejs");

}

Displays page 10 of the admin management system located at /manage/a-index-bet/index10.ejs.

**adminPage5d**

*const* adminPage5d = *async* (*req*, *res*) *=>* {

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

}

Renders the five-digits management page located at /manage/5d.ejs.

**adminPageK3**

*const* adminPageK3 = *async* (*req*, *res*) *=>* {

    return *res*.render("manage/k3.ejs");

}

Displays the K3 management page at /manage/k3.ejs.

**ctvProfilePage**

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

*var* phone = *req*.params.phone;

    return *res*.render("manage/profileCTV.ejs", { phone });

}

Renders the profile page for CTV (Collaborators) based on the phone parameter provided in the URL.

**giftPage**

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

    return *res*.render("manage/giftPage.ejs");

}

Renders the gift management page at /manage/giftPage.ejs.

**membersPage**

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

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

}

Displays the members management page at /manage/members.ejs.

**adminChatPage**

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

*var* f\_api= process.env.Firebase\_Apikey;

*var* f\_authdomain= process.env.Firebase\_AuthDomain;

*var* f\_dburl= process.env.Firebase\_Dburl;

*var* f\_projid= process.env.Firebase\_ProjId;

*var* f\_stobck= process.env.Firebase\_StorageBucket;

*var* f\_messId= process.env.Firebase\_MessageSenId;

*var* f\_appid= process.env.Firebase\_AppId;

*var* f\_mesuareId= process.env.Firebase\_MeasurementId;

return *res*.render("manage/aChat.ejs",{d\_f\_api: f\_api, d\_f\_authdomain :f\_authdomain,d\_f\_dburl:f\_dburl,d\_f\_projid:f\_projid,d\_f\_stobck:f\_stobck,d\_f\_messId:f\_messId,d\_f\_appid:f\_appid,d\_f\_mesuareId:f\_mesuareId });

}

The code defines an asynchronous function called adminChatPage. This function takes two inputs: req (request) and res (response).

Inside the function, it retrieves several environment variables related to Firebase, such as the API key, authentication domain, database URL, project ID, storage bucket, messaging sender ID, app ID, and measurement ID.

Finally, it sends a response that renders a webpage called aChat.ejs and passes these Firebase values to that page as data.

**k3chatPage**

*const* k3chatPage = *async* (*req*, *res*) *=>* {

*var* f\_api= process.env.Firebase\_Apikey;

*var* f\_authdomain= process.env.Firebase\_AuthDomain;

*var* f\_dburl= process.env.Firebase\_Dburl;

*var* f\_projid= process.env.Firebase\_ProjId;

*var* f\_stobck= process.env.Firebase\_StorageBucket;

*var* f\_messId= process.env.Firebase\_MessageSenId;

*var* f\_appid= process.env.Firebase\_AppId;

*var* f\_mesuareId= process.env.Firebase\_MeasurementId;

    return *res*.render("manage/ChatK3.ejs",{d\_f\_api: f\_api, d\_f\_authdomain :f\_authdomain,d\_f\_dburl:f\_dburl,d\_f\_projid:f\_projid,d\_f\_stobck:f\_stobck,d\_f\_messId:f\_messId,d\_f\_appid:f\_appid,d\_f\_mesuareId:f\_mesuareId}

    );

}

This code defines a function called k3chatPage that handles a web request. It gets some settings for Firebase from environment variables, like API key, authentication domain, database URL, project ID, storage bucket, message sender ID, app ID, and measurement ID. Then, it sends a response to the user by rendering a template called "ChatK3.ejs" with those Firebase settings included.

**d5chatPage**

*const* d5chatPage = *async* (*req*, *res*) *=>* {

*var* f\_api= process.env.Firebase\_Apikey;

*var* f\_authdomain= process.env.Firebase\_AuthDomain;

*var* f\_dburl= process.env.Firebase\_Dburl;

*var* f\_projid= process.env.Firebase\_ProjId;

*var* f\_stobck= process.env.Firebase\_StorageBucket;

*var* f\_messId= process.env.Firebase\_MessageSenId;

*var* f\_appid= process.env.Firebase\_AppId;

*var* f\_mesuareId= process.env.Firebase\_MeasurementId;

    return *res*.render("manage/Chat5d.ejs",  {d\_f\_api: f\_api, d\_f\_authdomain :f\_authdomain,d\_f\_dburl:f\_dburl,d\_f\_projid:f\_projid,d\_f\_stobck:f\_stobck,d\_f\_messId:f\_messId,d\_f\_appid:f\_appid,d\_f\_mesuareId:f\_mesuareId});

}

This code is a function called d5chatPage. It takes two inputs: req (request) and res (response).

1. It retrieves several environment variables related to Firebase, which is a platform for building apps. These variables include:
   * Firebase\_Apikey
   * Firebase\_AuthDomain
   * Firebase\_Dburl
   * Firebase\_ProjId
   * Firebase\_StorageBucket
   * Firebase\_MessageSenId
   * Firebase\_AppId
   * Firebase\_MeasurementId
   * After getting these values, the function sends a response to the client. It uses the res.render method to display a webpage called "Chat5d.ejs".
   * The function also passes the Firebase values to the webpage so that it can use them. Each value is sent with a specific name, starting with d\_, to help the webpage access them.

**ctvPage**

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

    return *res*.render("manage/ctv.ejs");

}

Renders the CTV management page.

**User and Member Management Functions**

### userInfo

This function retrieves user information based on the phone number passed in the request body. It checks if the user exists, gathers hierarchical user data (like referrals), and compiles the user's transactions and financial details before returning the data in the response.

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

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

*let* phone = *req*.body.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]);

    if (user.length == 0) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* userInfo = user[0];

    // direct subordinate 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]);

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

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

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: user,

        total\_r: recharge,

        total\_w: withdraw,

        f1: f1s.length,

        f2: f2,

        f3: f3,

        f4: f4,

        bank\_user: bank\_user,

        telegram: telegram\_ctv[0],

        ng\_moi: ng\_moi[0],

        daily: userInfo.ctv,

    });

}

* It first checks if a phone number is provided in the request body.
* Returns a failure response if the phone number is missing or if the user is not found in the database.
* Retrieves the user information from the 'users' table based on the provided phone number.
* Performs a series of nested queries to calculate direct and indirect subordinate counts for different levels in the user's network.
* Calculates total recharges, withdrawals, and fetches bank details of the user.
* Fetches additional information such as the Telegram username of the user's direct superior and the phone number of the user's inviter.
* Finally, it returns a success response with various calculated data and fetched information.

### recharge

Balances and retrieves recharge and withdrawal data for the users. It categorizes based on their statuses to present relevant information to the requester.

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [recharge] = await connection.query('SELECT \* FROM recharge WHERE status = 0 ');

*const* [recharge2] = await connection.query('SELECT \* FROM recharge WHERE status != 0 ');

*const* [withdraw] = await connection.query('SELECT \* FROM withdraw WHERE status = 0 ');

*const* [withdraw2] = await connection.query('SELECT \* FROM withdraw WHERE status != 0 ');

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: recharge,

        datas2: recharge2,

        datas3: withdraw,

        datas4: withdraw2,

    });

}

It checks if the auth cookie exists in the request. If not, it returns a JSON response with status code 200 containing a message of 'Failed', a status of false, and a timestamp generated using new Date().getTime().

Then, it executes four asynchronous queries using connection.query() to retrieve data from the database tables recharge and withdraw based on their status values.

Finally, it returns a JSON response with status code 200 containing a message of 'Success', a status of true, and the queried data stored in variables recharge, recharge2, withdraw, and withdraw2.

### settingGet

Retrieves various application and user-specific settings including banking 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,

        });

    }

}

Within a try-catch block, the function attempts to retrieve the authentication token from the request cookies. If no token is found, it returns a JSON response with a status of 'Failed'.

Subsequently, SQL queries are executed using an active database connection to fetch specific data from tables like users, bank\_recharge, bank\_recharge\_momo, and admin.

If data is found in bank\_recharge\_momo based on certain conditions, it extracts the first element from the result.

Finally, a JSON response with status 'Success' is returned with retrieved settings and data related to bank recharge and momo services.

If any errors occur during the process, a 500 status response with 'Failed' message is sent along with the error logged to the console.

### rechargeDuyet

Processes recharge requests based on user confirmation and updates the status accordingly. It handles both confirmations and deletions of recharge requests.

*const* 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: [],

        });

    }

}

* It first extracts auth, id, and type from req.cookies and req.body.
* If any of these values is missing, it returns a failure response.
* If type is 'confirm', it updates the status of a recharge in the database based on the provided id.
* Based on the type of recharge, it performs different operations like updating balances, sending notifications, and handling rewards.
* For 'delete' type, it updates the status of the recharge to indicate cancellation.
* Finally, it returns a success or cancellation response with appropriate messages.

### getUserDataByPhone

Fetches user data such as phone, code, and invite information 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,

    }

}

The provided code defines an asynchronous function getUserDataByPhone that accepts a phone parameter. Inside the function, it queries a database using connection.query to fetch user data based on the provided phone number.

The fetched users array is destructured to extract the first element (if it exists) into the user variable using optional chaining (?.) to handle cases where users may be undefined or empty.

If the user is not found (undefined or null), an Error is thrown with the message 'Unable to get user data!'

Finally, the function returns an object containing specific properties (phone, code, username, invite) extracted from the user object.

### addUserAccountBalance

Adds a specified amount to the user's account balance and also awards a bonus to the inviter associated with the user.

*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")

    }

}

This code defines an asynchronous function called addUserAccountBalance that takes an object with money, phone, and invite as parameters. The function calculates the user's money and inviter's money based on the input money. It then updates the user's money and total money in the database using the provided phone number.

Next, it queries the database to find an inviter based on the invite code. If an inviter is found, it updates the inviter's money and total money in the database.

The code uses async/await to handle asynchronous operations like querying the database. The connection object is assumed to be defined elsewhere and provides the query method to interact with the database.

### updateLevel

Updates user levels based on provided hierarchy inputs such as f1, f2, f3, and f4.

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

    try {

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

*let* f1 = *req*.body.f1;

*let* f2 = *req*.body.f2;

*let* f3 = *req*.body.f3;

*let* f4 = *req*.body.f4;

        console.log("level : " + id, f1, f2, f3, f4);

        await connection.query(

            'UPDATE `level` SET `f1`= ? ,`f2`= ? ,`f3`= ? ,`f4`= ?  WHERE `id` = ?',

            [f1, f2, f3, f4, id]

        );

        // Send a success response to the client

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

            message: 'Update successful',

            status: true,

        });

    } catch (error) {

        console.error('Error updating level:', error);

        // Send an error response to the client

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

            message: 'Update failed',

            status: false,

            error: error.message,

        });

    }

};

The given code defines an asynchronous function updateLevel that updates records in a database table named 'level' based on the request body parameters.

* req and res are objects representing the request and response respectively.
* The function retrieves id, f1, f2, f3, f4 from the request body.
* It then logs the values of these parameters to the console.
* Subsequently, an SQL UPDATE query is executed using the connection.query method to update the 'level' table with new values of f1, f2, f3, f4 based on the provided id.
* If the update is successful, a JSON response with a success message and status is sent back with a 200 status code.
* If an error occurs during the update process, an error response containing the error message is sent back with a 500 status code.

### handlWithdraw

Handles withdrawal requests by updating the status based on confirmation or deletion actions.

*const* handlWithdraw = *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 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: recharge,

        });

    }

    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: recharge,

        });

    }

}

This code defines an asynchronous function handleWithdraw that handles withdrawal requests.

It extracts authentication data, ID, and type from the request body. If any of these values are missing, it returns a failure response.

If the type is 'confirm', it updates the withdrawal status, fetches related information, and processes either a transfer or a simple withdrawal. It then sends notifications based on the transaction type.

If the type is 'delete', it updates the withdrawal status to '2', returns the withdrawn money back to the user, and sends a cancellation success message.

The code uses SQL queries to interact with a database and contains conditional flows based on the withdrawal type.

### createBonus

Creates bonuses for users based on selected criteria, allowing for bonus amounts to be adjusted based on different selections.

*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* 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;

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

    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];

    if (type == 'all') {

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

        if (select == '1') {

            await connection.query(`UPDATE point\_list SET money = money + ? WHERE level = 2`, [money]);

        } else {

            await connection.query(`UPDATE point\_list SET money = money - ? WHERE level = 2`, [money]);

        }

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

            message: 'successful change',

            status: true,

        });

    }

    if (type == 'two') {

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

        if (select == '1') {

            await connection.query(`UPDATE point\_list SET money\_us = money\_us + ? WHERE level = 2`, [money]);

        } else {

            await connection.query(`UPDATE point\_list SET money\_us = money\_us - ? WHERE level = 2`, [money]);

        }

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

            message: 'successful change',

            status: true,

        });

    }

    if (type == 'one') {

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

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

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

        if (user.length == 0) {

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

                message: 'Failed',

                status: false,

                timeStamp: timeNow,

            });

        }

        if (select == '1') {

            await connection.query(`UPDATE point\_list SET money = money + ? WHERE level = 2 and phone = ?`, [money, phone]);

        } else {

            await connection.query(`UPDATE point\_list SET money = money - ? WHERE level = 2 and phone = ?`, [money, phone]);

        }

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

            message: 'successful change',

            status: true,

        });

    }

    if (type == 'three') {

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

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

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

        if (user.length == 0) {

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

                message: 'account does not exist',

                status: false,

                timeStamp: timeNow,

            });

        }

        if (select == '1') {

            await connection.query(`UPDATE point\_list SET money\_us = money\_us + ? WHERE level = 2 and phone = ?`, [money, phone]);

        } else {

            await connection.query(`UPDATE point\_list SET money\_us = money\_us - ? WHERE level = 2 and phone = ?`, [money, phone]);

        }

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

            message: 'successful change',

            status: true,

        });

    }

    if (!type) {

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

*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]);

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

            message: 'Successful change',

            status: true,

            id: id\_redenvelops,

        });

    }

}

The code defines an asynchronous function createBonus that handles various operations based on the input type. Here's a breakdown of the code:

1. A helper function randomString generates a random alphanumeric string of a specified length.
2. A function timerJoin formats a given timestamp to a specific date-time format.
3. Retrieves authentication token, money, and type from the request body.
4. Validates the presence of money and authentication token; returns an error response if not present.
5. Queries the database to find a user based on the provided token.
6. Based on the 'type' received, different actions are performed like updating the 'money' field in the database for different scenarios.
7. If 'type' is not provided, it generates a unique id\_redenvelops and inserts a new record in the 'redenvelopes' table.
8. Returns a JSON response with the corresponding message and status for each scenario.

### register

This function registers a new user by validating input data and storing it in the database.

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

*let* { username, password,invitecode } = *req*.body;

*let* id\_user = randomNumber(10000, 99999);

*let* name\_user = "Member" + randomNumber(10000, 99999);

*let* code = randomString(5) + randomNumber(10000, 99999);

*let* ip = ipAddress(*req*);

*let* time = timeCreate();

    invitecode = '2cOCs36373';

    if (!username || !password || !invitecode) {

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

            message: 'ERROR!!!',

            status: false

        });

    }

    if (!username) {

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

            message: 'phone error',

            status: false

        });

    }

    try {

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

        if (check\_u.length == 1) {

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

                message: 'register account', //Số điện thoại đã được đăng ký

                status: false

            });

        } else {

*const* sql = `INSERT INTO users SET

            id\_user = ?,

            phone = ?,

            name\_user = ?,

            password = ?,

            money = ?,

            level = ?,

            code = ?,

            invite = ?,

            veri = ?,

            ip\_address = ?,

            status = ?,

            time = ?`;

            await connection.execute(sql, [id\_user, username, name\_user, md5(password), 0, 2, code, invitecode, 1, ip, 1, time]);

            await connection.execute('INSERT INTO point\_list SET phone = ?, level = 2', [username]);

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

                message: 'registration success',//Register Sucess

                status: true

            });

        }

    } catch (error) {

        if (error) console.log(error);

    }

}

The code defines an asynchronous function register that handles user registration. Here is a breakdown of the code:

* It extracts username, password, and invitecode from the req.body.
* Generates random id\_user, name\_user, and code.
* Gets the user's IP address and current time.
* Hardcodes the invitecode value to '2cOCs36373'.
* Checks for missing fields and returns an error response if any field is missing.
* Performs a database query to check if the user already exists. If so, it returns an error response.
* If the user does not exist, it constructs an SQL query to insert user details into the database.
* Executes the SQL query and responds with a success message if the registration is successful.
* Catches any errors that occur during the database operations and logs them to the console.

### profileUser

Gets the profile information of a user identified by their phone number. It returns financial transaction history along with the User’s profile details.

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

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

    if (!phone) {

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

            message: 'Phone Error',

            status: false,

            timeStamp: timeNow,

        });

    }

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

    if (user.length == 0) {

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

            message: 'Phone Error',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* [recharge] = await connection.query(`SELECT \* FROM recharge WHERE phone = ? ORDER BY id DESC LIMIT 10`, [phone]);

*let* [withdraw] = await connection.query(`SELECT \* FROM withdraw WHERE phone = ? ORDER BY id DESC LIMIT 10`, [phone]);

    return *res*.status(200).json({

        message: 'Get success',

        status: true,

        recharge: recharge,

        withdraw: withdraw,

    });

}

In the provided code snippet, an asynchronous function profileUser is defined that takes req (request) and res (response) as parameters. It accesses the phone number from the request body and performs database queries based on that phone number.

If the phone number is not provided, it sends a JSON response with a 'Phone Error' message and status set to false along with the current timestamp. If no user is found for the provided phone number, it returns a similar error response.

Otherwise, it queries the 'recharge' and 'withdraw' tables for the given phone number, orders the results by ID in descending order, and limits the results to the latest 10 entries for each table. Finally, it sends a success response with status set to true along with the fetched 'recharge' and 'withdraw' data.

**infoMember**

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

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

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

}

Displays the information of a specified member based on the phone ID provided in the URL.

**listMember**

Fetches a paginated list of members from the database.

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

*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

        });

    }

*const* [users] = await connection.query(`SELECT \* FROM users WHERE veri = 1  ORDER BY id DESC LIMIT ${pageno}, ${limit}; `);

*const* [total\_users] = await connection.query(`SELECT \* FROM users WHERE veri = 1 ; `);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: users,

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

    });

}

The given code defines an asynchronous function listMember that handles a request and response object. It retrieves pageno and limit from the request body.

If either pageno or limit is missing or less than zero, it returns a JSON response indicating 'No more data' with an empty gameslist array and status false.

It then executes SQL queries to fetch users based on the verification status and calculates the total number of users meeting the verification requirement. Finally, it returns a JSON response with the retrieved users, success message, status true, and the total number of pages based on the supplied limit.

## settingBank

This function handles the request for setting bank-related information. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

*const* settingBank = *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: recharge,

            });

        }

        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: recharge,

            });

        }

    } catch (error) {

        console.log(error)

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

            message: 'Something went wrong!',

            status: false,

        });

    }

}

## deleteBankRechargeById(id)

This function deletes a bank recharge entry from the database based on the provided ID. It expects the ID of the bank recharge entry as a parameter. The function performs the following steps:

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

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

    return recharge

}

## tranfermode

This function handles the request for updating the transfer mode for a user. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

1. Extracts the authentication token from the request cookies and the transfer mode from the request body.
2. Queries the database to retrieve information about the user associated with the authentication token.
3. Updates the transfer mode for the user in the 'bank\_recharge' table in the database.
4. Returns a JSON response with a 'Submitted successfully' message and sets the status to true.
5. *const* tranfermode = *async* (*req*, *res*) *=>* {
6. *let* auth = *req*.cookies.auth;
7. *let* tran\_mode = *req*.body.mode\_tran;
8. *const* [rows] = await connection.query('SELECT \* FROM users WHERE `token` = ? ', [auth]);
9. *let* user = rows[0];
10. await connection.execute("UPDATE bank\_recharge SET transfer\_mode = ?  WHERE `phone` = ? ", [tran\_mode,user.phone] );
11. return *res*.status(200).json({
12. message: 'Submitted successfully',
13. status: true,
14. });
15. }

## settingCskh

This function handles the request for setting customer support information. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

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

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

*let* telegram = *req*.body.telegram;

*let* cskh = *req*.body.cskh;

*let* myapp\_web = *req*.body.myapp\_web;

    if (!auth || !cskh || !telegram) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

    await connection.query(`UPDATE admin SET telegram = ?, cskh = ?, app = ?`, [telegram, cskh, myapp\_web]);

    return *res*.status(200).json({

        message: 'Successful change',

        status: true,

    });

}

## levelSetting

This function renders the 'manage/levelSetting.ejs' view. It does not expect any parameters.

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

    return *res*.render("manage/levelSetting.ejs");

}

## settings

This function renders the 'manage/settings.ejs' view. It does not expect any parameters.

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

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

}

## editResult2

This function handles the request for editing a result. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

1. Extracts the game and list parameters from the request body.
2. Verifies the presence of the list and game parameters in the request. If any of them is missing, it returns a JSON response with an 'ERROR!!!' message and sets the status to false.
3. Determines the appropriate variable ('join') based on the provided game parameter.
4. Executes an SQL update query to update the corresponding variable with the provided list value in the 'admin' table.
5. Returns a JSON response with an 'Editing is successful' message and sets the status to true.

*const* editResult2 = *async* (*req*, *res*) *=>* {

*let* { game, list } = *req*.body;

    if (!list || !game) {

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

            message: 'ERROR!!!',

            status: false

        });

    }

*let* join = '';

    if (game == 1) join = 'k3d';

    if (game == 3) join = 'k3d3';

    if (game == 5) join = 'k3d5';

    if (game == 10) join = 'k3d10';

*const* sql = `UPDATE admin SET ${join} = ?`;

    await connection.execute(sql, [list]);

    return *res*.status(200).json({

        message: 'Editing is successful',//Register Sucess

        status: true

    });

}

## CreatedSalary

This function handles the request for creating a salary record. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

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

    try {

*const* phone = *req*.body.phone;

*const* amount = *req*.body.amount;

*const* type = *req*.body.type;

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

*const* formattedTime = now.toLocaleString('en-US', {

            year: 'numeric',

            month: '2-digit',

            day: '2-digit',

            hour: '2-digit',

            minute: '2-digit',

            second: '2-digit',

            hour12: true

        });

        // Check if the phone number is a 10-digit number

        if (!/^\d{10}$/.test(phone)) {

            return *res*.status(400).json({

                message: 'ERROR!!! Invalid phone number. Please provide a 10-digit phone number.',

                status: false

            });

        }

        // Check if user with the given phone number exists

*const* checkUserQuery = 'SELECT \* FROM `users` WHERE phone = ?';

*const* [existingUser] = await connection.execute(checkUserQuery, [phone]);

        if (existingUser.length === 0) {

            // If user doesn't exist, return an error

            return *res*.status(400).json({

                message: 'ERROR!!! User with the provided phone number does not exist.',

                status: false

            });

        }

        // If user exists, update the 'users' table

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

        await connection.execute(updateUserQuery, [amount, phone]);

        // Insert record into 'salary' table

*const* insertSalaryQuery = 'INSERT INTO salary (phone, amount, type, time) VALUES (?, ?, ?, ?)';

        await connection.execute(insertSalaryQuery, [phone, amount, type, formattedTime]);

*res*.status(200).json({ message: 'Salary record created successfully' });

    } catch (error) {

        console.error(error);

*res*.status(500).json({ error: 'Internal server error' });

    }

};

1. Extracts the phone number, amount, and type parameters from the request body.
2. Validates the phone number to ensure it is a 10-digit number. If it is invalid, it returns a JSON response with an 'ERROR!!! Invalid phone number' message and sets the status to false.
3. Checks if a user with the provided phone number exists. If not, it returns a JSON response with an 'ERROR!!! User with the provided phone number does not exist' message and sets the status to false.
4. Updates the 'money' field for the user in the 'users' table with the provided amount.
5. Inserts a salary record into the 'salary' table with the phone number, amount, type, and current timestamp.
6. Returns a JSON response with a 'Salary record created successfully' message.

## getSalary

This function handles the request for retrieving all salary records. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

1. Queries the 'salary' table to retrieve all records, ordered by timestamp in descending order.
2. Returns a JSON response with a 'Success' message, sets the status to true, and includes the retrieved salary records.
3. *const* getSalary = *async* (*req*, *res*) *=>* {
4. *const* [rows] = await connection.query(`SELECT \* FROM salary ORDER BY time DESC`);
5. if (!rows) {
6. return *res*.status(200).json({
7. message: 'Failed',
8. status: false,
9. });
10. }
11. console.log("asdasdasd : " + rows)
12. return *res*.status(200).json({
13. message: 'Success',
14. status: true,
15. data: {
16. },
17. rows: rows
18. })
19. };

## gettranfermode

This function handles the request for retrieving the transfer mode for a user. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

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

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

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

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

    if (!rows) {

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

            message: 'Failed',

            status: false,

        });

    }

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        data: {

        },

        rows: rows

    })

};

## getdashboardInfo

This function handles the request for retrieving dashboard information. It expects a request object (req) and a response object (res) as parameters. The function performs the following steps:

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

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

*let* totaltodayUsers = 0;

*let* totaltodayRecharge = 0;

*let* totaltodayWithdrawal = 0;

*let* usersBalanace = 0;

*let* totalUsers = 0;

*let* peningRecharge = 0;

*let* sucessRecharge = 0;

*let* totalwithdrawal = 0;

*let* withdrawalRequest = 0;

*let* totalBet = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

*let* totalWinning = 0;

*let* win\_total\_w = 0;

*let* win\_total\_k3 = 0;

*let* win\_total\_5d = 0;

*let* win\_total\_trx = 0;

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

*const* [today\_minutes\_1] = await connection.query("SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE  `time` >= ?;", [today]);

*const* [today\_k3\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_k3 WHERE  `time` >= ?;", [today]);

*const* [today\_d5\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM result\_5d WHERE  `time` >= ?;", [today]);

*const* [today\_trx\_bet\_money] = await connection.query("SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE  `time` >= ?;", [today]);

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

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

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

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

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

*const* [win\_today\_minutes\_1] = await connection.query("SELECT SUM(get) AS `sum` FROM minutes\_1 WHERE  `time` >= ?;", [today]);

*const* [win\_today\_k3\_bet\_money] = await connection.query("SELECT SUM(get) AS `sum` FROM result\_k3 WHERE  `time` >= ?;", [today]);

*const* [win\_today\_d5\_bet\_money] = await connection.query("SELECT SUM(get) AS `sum` FROM result\_5d WHERE  `time` >= ?;", [today]);

*const* [win\_today\_trx\_bet\_money] = await connection.query("SELECT SUM(get) AS `sum` FROM trx\_wingo\_bets WHERE  `time` >= ?;", [today]);

    win\_total\_w = win\_today\_minutes\_1[0].sum || 0;

    win\_total\_k3 = win\_today\_k3\_bet\_money[0].sum || 0;

    win\_total\_5d = win\_today\_d5\_bet\_money[0].sum || 0;

    win\_total\_trx = win\_today\_trx\_bet\_money[0].sum || 0;

    totalWinning += parseInt(win\_total\_w) + parseInt(win\_total\_k3) + parseInt(win\_total\_5d) + parseInt(win\_total\_trx);

*const* [users\_join\_today] = await connection.query("SELECT COUNT(\*) AS `count` FROM users WHERE  `time` >= ?;", [today]);

    totaltodayUsers = users\_join\_today[0].count || 0;

*const* [users\_recharge\_today] = await connection.query("SELECT SUM(money) AS `sum` FROM recharge WHERE  `time` >= ? AND `status` = ?;", [today, 1]);

    totaltodayRecharge = users\_recharge\_today[0].sum || 0;

*const* [users\_withdraw\_today] = await connection.query("SELECT SUM(money) AS `sum` FROM withdraw WHERE  `time` >= ? AND `status` = ?;", [today, 1]);

    totaltodayWithdrawal = users\_withdraw\_today[0].sum || 0;

*const* [users\_total] = await connection.query("SELECT COUNT(\*) AS `count` FROM users;", []);

    totalUsers = users\_total[0].count || 0;

*const* [users\_balance] = await connection.query("SELECT SUM(money) AS `sum` FROM users;", []);

    usersBalanace = users\_balance[0].sum || 0;

*const* [list\_pending\_recharge] = await connection.query("SELECT COUNT(\*) AS `count` FROM recharge WHERE  `status` = ?;", [0]);

    peningRecharge = list\_pending\_recharge[0].count || 0;

*const* [list\_success\_recharge] = await connection.query("SELECT COUNT(\*) AS `count` FROM recharge WHERE  `status` = ?;", [1]);

    sucessRecharge = list\_success\_recharge[0].count || 0;

*const* [list\_pending\_withdraw] = await connection.query("SELECT COUNT(\*) AS `count` FROM withdraw WHERE  `status` = ?;", [0]);

    withdrawalRequest = list\_pending\_withdraw[0].count || 0;

*const* [list\_success\_withdraw] = await connection.query("SELECT COUNT(\*) AS `count` FROM withdraw WHERE  `status` = ?;", [1]);

    totalwithdrawal = list\_success\_withdraw[0].count || 0;

*let* monthkyturnover = 0;

*const* [list\_month\_turn\_over] = await connection.query("SELECT SUM(daily\_turn\_over) AS `sum` FROM turn\_over WHERE MONTH(`date\_time`) = MONTH(CURRENT\_DATE()) AND YEAR(`date\_time`) = YEAR(CURRENT\_DATE()) ORDER BY `id` DESC;");

    monthkyturnover = list\_month\_turn\_over[0].sum || 0;

*let* monthrechagebonus = 0;

*const* [list\_month\_recharge] = await connection.query("SELECT SUM(`money`) AS `sum` FROM recharge WHERE `status`=1 AND MONTH(STR\_TO\_DATE(`today`, '%Y-%d-%m %h:%i:%s %p')) = MONTH(CURRENT\_DATE()) AND YEAR(STR\_TO\_DATE(`today`, '%Y-%d-%m %h:%i:%s %p')) = YEAR(CURRENT\_DATE()) ORDER BY `id` DESC;");

*const* monthrecharge = list\_month\_recharge[0].sum || 0;

    monthrechagebonus = parseInt((monthrecharge/100) \* 5);

*let* monthstakingamount = 0;

*const* [list\_month\_stakes] = await connection.query("SELECT SUM(`amount`) AS `sum` FROM claimed\_rewards WHERE `reward\_id`=136 AND MONTH(`to\_date`) = MONTH(CURRENT\_DATE()) AND YEAR(`to\_date`) = YEAR(CURRENT\_DATE()) ORDER BY `id` DESC;");

    monthstakingamount = list\_month\_stakes[0].sum || 0;

*let* monthstakingcount = 0;

*const* [list\_month\_stakes\_c] = await connection.query("SELECT SUM(`stake\_amnt`) AS `sum` FROM claimed\_rewards WHERE `reward\_id`=136 AND MONTH(`to\_date`) = MONTH(CURRENT\_DATE()) AND YEAR(`to\_date`) = YEAR(CURRENT\_DATE()) ORDER BY `id` DESC;");

    monthstakingcount = list\_month\_stakes\_c[0].sum || 0;

*let* active\_stakes\_count = 0;

*const* [list\_active\_stakes] = await connection.query("SELECT COUNT(\*) AS `count` FROM claimed\_rewards WHERE `reward\_id`=136 AND `status`= 2 ORDER BY `id` DESC;");

    active\_stakes\_count = list\_active\_stakes[0].count || 0;

*let* active\_stakes\_amt = 0;

*const* [list\_active\_stakes\_amt] = await connection.query("SELECT SUM(`amount`) AS `sum` FROM claimed\_rewards WHERE `reward\_id`=136 AND `status`= 2 ORDER BY `id` DESC;");

    active\_stakes\_amt = list\_active\_stakes\_amt[0].sum || 0;

*let* giftcodevalue = 0;

*const* [list\_redenvelopes\_used] = await connection.query("SELECT \* FROM redenvelopes\_used;");

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

*const* re\_dev\_used\_time = list\_redenvelopes\_used[i].time;

*let* check = timerJoin1(re\_dev\_used\_time);

        if(check == "match")

        {

            giftcodevalue +=  parseInt(list\_redenvelopes\_used[i].money);

        }

    }

*const* colloboratordata =  await getcolloboratorData();

*let* stakeROI = parseFloat(monthstakingamount).toFixed(2) - parseFloat(monthstakingcount).toFixed(2);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        data: {

            a\_usersJoin:totaltodayUsers,

            a\_TodaysRecharge:totaltodayRecharge,

            a\_TodaysWithdrawal:totaltodayWithdrawal,

            a\_UserBalance:usersBalanace,

            a\_TotalUsers:totalUsers - 1,

            a\_PendingRecharge:peningRecharge,

            a\_SuccessRecharge:sucessRecharge,

            a\_TotalWithdrawal:totalwithdrawal,

            a\_WithdrawalRequests:withdrawalRequest,

            a\_TodaysTotalBets:totalBet,

            a\_TodaysTotalWin:totalWinning,

            a\_TodaysProfit:totalBet -totalWinning,

            a\_month\_colloborator:colloboratordata.result\_val,

            a\_month\_turnover:monthkyturnover,

            a\_month\_gift\_redeem:giftcodevalue,

            a\_month\_recharge\_bonus:monthrechagebonus,

            a\_month\_staking\_amt: parseFloat(monthstakingamount).toFixed(2),

            a\_month\_staking\_rewards:stakeROI,

            a\_active\_stakes:active\_stakes\_count,

            a\_roi\_active\_stakes:parseFloat(active\_stakes\_amt).toFixed(2)

        },

    })

}

1. Extracts the authentication token from the request cookies.
2. Performs various calculations and queries to retrieve information such as the number of users that joined today, today's total recharge and withdrawal amounts, total user balance, total number of users, pending recharge and withdrawal counts, today's total bets and winnings, monthly turnover, and more.
3. Returns a JSON response with a 'Success' message, sets the status to true, and includes the retrieved dashboard information.
4. Extracts the authentication token from the request cookies.
5. Queries the 'users' table to retrieve the phone number associated with the authentication token.
6. Queries the 'bank\_recharge' table to retrieve the transfer mode for the user.
7. Returns a JSON response with a 'Success' message, sets the status to true, and includes the retrieved transfer mode.

### settingbuff

This function is responsible for updating the money account balance for a user. It is triggered when a request is made to the "/settingbuff" endpoint.

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

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

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

*let* buff\_acc = *req*.body.buff\_acc;

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

    if (!id\_user || !buff\_acc || !money\_value) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user\_id] = await connection.query(`SELECT \* FROM users WHERE id\_user = ?`, [id\_user]);

    if (user\_id.length > 0) {

        if (buff\_acc == '1') {

            await connection.query(`UPDATE users SET money = money + ? WHERE id\_user = ?`, [money\_value, id\_user]);

        }

        if (buff\_acc == '2') {

            await connection.query(`UPDATE users SET money = money - ? WHERE id\_user = ?`, [money\_value, id\_user]);

        }

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

            message: 'Successful change',

            status: true,

        });

    } else {

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

            message: 'Successful change',

            status: false,

        });

    }

}

The function retrieves the user's authentication token from the request cookies and the necessary user and transaction data from the request body. It then verifies the availability of the required data. If any of the fields (id\_user, buff\_acc, or money\_value) are missing, it returns a "Failed" response indicating the missing fields.

If all the fields are present, the function queries the database to fetch the user with the provided id\_user. If the user is found, it updates the user's money account based on the buff\_acc value:

* If buff\_acc is equal to 1, the money balance is increased by money\_value.
* If buff\_acc is equal to 2, the money balance is decreased by money\_value.

The function then returns a "Successful change" response indicating the success of the money account update.

### listCTV

This function retrieves a list of users (CTVs) based on the provided pagination parameters (pageno and pageto). It is triggered when a request is made to the "/listCTV" endpoint.

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

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

    if (!pageno || !pageto) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (pageno < 0 || pageto < 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

*const* [wingo] = await connection.query(`SELECT \* FROM users WHERE veri = 1 AND level = 2 ORDER BY id DESC LIMIT ${pageno}, ${pageto} `);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: wingo,

    });

}

### infoCtv

This function retrieves various information about a CTV (user) based on the provided phone number. It is triggered when a request is made to the "/infoCtv" endpoint.

The function retrieves the phone number from the request body and queries the database to fetch information about the user with the provided phone number.

If the user is found, the function performs several additional queries to calculate various statistics related to the user's performance:

1. f1: The number of directly referred users (level = 1).
2. f2: The number of users referred by f1 (level = 2).
3. f3: The number of users referred by f2 (level = 3).
4. f4: The number of users referred by f3 (level = 4).
5. list\_mems: The list of users directly referred by the CTV.
6. total\_recharge: The total amount of money recharge by all users referred by the CTV.
7. total\_withdraw: The total amount of money withdrawn by all users referred by the CTV.
8. total\_recharge\_today: The total amount of money recharge today by all users referred by the CTV.
9. total\_withdraw\_today: The total amount of money withdrawn today by all users referred by the CTV.
10. list\_mem\_baned: The number of users referred by the CTV who are currently banned.
11. win: The total amount of money won by the CTV and their referred users today.
12. loss: The total amount of money lost by the CTV and their referred users today.
13. list\_recharge\_news: The list of recharge transactions made by all users referred by the CTV today.
14. list\_withdraw\_news: The list of withdrawal transactions made by all users referred by the CTV today.
15. moneyCTV: The amount of money earned by the CTV as commission.
16. redenvelopes\_used: The list of redenvelopes used by the CTV today.
17. financial\_details\_today: The list of financial transactions made by the CTV today.

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

*const* phone = *req*.body.phone;

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

    if (user.length == 0) {

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

            message: 'Phone Error',

            status: false,

        });

    }

*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* [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) {

            list\_mems.push(list\_mem\_today[i]);

        }

    }

*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 function finally returns a "Success" response with the retrieved information.

### infoCtv2

This function retrieves various information about a CTV (user) based on the provided phone number and time and date. It is triggered when a request is made to the "/infoCtv2" endpoint.

The function retrieves the phone number and timeDate from the request body and queries the database to fetch information about the user and other related data.

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

*const* phone = *req*.body.phone;

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

*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 phone = ? ', [phone]);

    if (user.length == 0) {

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

            message: 'Phone Error',

            status: false,

        });

    }

*let* userInfo = user[0];

*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) {

            list\_mems.push(list\_mem\_today[i]);

        }

    }

*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 function performs similar operations as the infoCtv function but with the additional parameter timeDate. It calculates the statistics based on the specified time and date instead of the current time.

The calculated statistics and the retrieved information are then returned in the response.

### listRedenvelope

This function retrieves a list of redenvelopes used by a specific user (CTV). It is triggered when a request is made to the "/listRedenvelope/:phone" endpoint.

*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];

*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)

    });

}

The function retrieves the user's authentication token, phone number, and pagination parameters from the request. It verifies their availability and fetches the corresponding user from the database.

If the user is found, the function queries the database to fetch a paginated list of redenvelopes used by the user. The result is limited based on the provided pagination parameters.

The function returns a "Success" response with the list of redenvelopes and the total number of users using redenvelopes

### listBet

This function retrieves a list of bets made by a specific user. It is triggered when a request is made to the "/listBet/:phone" endpoint.

*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];

*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)

    });

}

The function retrieves the user's authentication token, phone number, and pagination parameters from the request. It verifies their availability and fetches the corresponding user from the database.

If the user is found, the function queries the database to fetch a paginated list of bets made by the user. The result is limited based on the provided pagination parameters.

The function returns a "Success" response with the list of bets and the total number of bets made by the user.

### listOrderOld

This function retrieves a list of game results for a specific game (K5D). It is triggered when a request is made to the "/listOrderOld" endpoint.

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

*let* { gameJoin } = *req*.body;

*let* internet\_bet = *req*.body.join\_al;

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

    if (!checkGame) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

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

*let* join = '';

    if (game == 1) join = 'k5d';

    if (game == 3) join = 'k5d3';

    if (game == 5) join = 'k5d5';

    if (game == 10) join = 'k5d10';

*const* [k5d] = await connection.query(`SELECT \* FROM d5 WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT 10 `);

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

*const* [waiting] = await connection.query(`SELECT phone, money, price, amount, bet, join\_bet FROM result\_5d WHERE status = 0 AND level = 0 AND game = '${game}' AND join\_bet = '${internet\_bet}' ORDER BY id ASC `);

*const* [settings] = await connection.query(`SELECT ${join} FROM admin`);

    if (k5d.length == 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!k5d[0] || !period[0]) {

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

            message: 'Error!',

            status: false

        });

    }

    return *res*.status(200).json({

        code: 0,

        msg: "Get success",

        data: {

            gameslist: k5d,

        },

        bet: waiting,

        settings: settings,

        join: join,

        period: period[0].period,

        status: true

    });

}

The function retrieves the gameJoin and internet\_bet parameters from the request body and verifies their availability. It then fetches the game results and related waiting bets from the database.

If the game results are available, the function returns a "Get success" response with the game results, waiting bets, and relevant settings.

### listOrderOldK3

This function retrieves a list of game results for a specific game (K3D). It is triggered when a request is made to the "/listOrderOldK3" endpoint.

*const* listOrderOldK3 = *async* (*req*, *res*) *=>* {

*let* { gameJoin } = *req*.body;

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

    if (!checkGame) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

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

*let* join = '';

    if (game == 1) join = 'k3d';

    if (game == 3) join = 'k3d3';

    if (game == 5) join = 'k3d5';

    if (game == 10) join = 'k3d10';

*const* [k5d] = await connection.query(`SELECT \* FROM k3 WHERE status != 0 AND game = '${game}' ORDER BY id DESC LIMIT 10 `);

*const* [period] = await connection.query(`SELECT period FROM k3 WHERE status = 0 AND game = '${game}' ORDER BY id DESC LIMIT 1 `);

*const* [waiting] = await connection.query(`SELECT phone, money, price, typeGame, amount, bet FROM result\_k3 WHERE status = 0 AND level = 0 AND game = '${game}' ORDER BY id ASC `);

*const* [settings] = await connection.query(`SELECT ${join} FROM admin`);

    if (k5d.length == 0) {

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

            code: 0,

            msg: "No more data",

            data: {

                gameslist: [],

            },

            status: false

        });

    }

    if (!k5d[0] || !period[0]) {

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

            message: 'Error!',

            status: false

        });

    }

    return *res*.status(200).json({

        code: 0,

        msg: "Get Success",

        data: {

            gameslist: k5d,

        },

        bet: waiting,

        settings: settings,

        join: join,

        period: period[0].period,

        status: true

    });

}

The function retrieves the gameJoin parameter from the request body and verifies its availability. It then fetches the game results and related waiting bets from the database.

If the game results are available, the function returns a "Get success" response with the game results, waiting bets, and relevant settings.

### editResult

This function edits the game result list for a specific game. It is triggered when a request is made to the "/editResult" endpoint.

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

*let* { game, list } = *req*.body;

    if (!list || !game) {

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

            message: 'ERROR!!!',

            status: false

        });

    }

*let* join = '';

    if (game == 1) join = 'k5d';

    if (game == 3) join = 'k5d3';

    if (game == 5) join = 'k5d5';

    if (game == 10) join = 'k5d10';

*const* sql = `UPDATE admin SET ${join} = ?`;

    await connection.execute(sql, [list]);

    return *res*.status(200).json({

        message: 'Editing is successful',//Register Sucess

        status: true

    });

}

The function retrieves the game and list parameters from the request body and verifies their availability. It then updates the corresponding game results in the database with the provided list.

The function returns an "Editing is successful" response indicating the success of the operation.

### CreatedSalaryRecord

This function renders the "CreatedSalaryRecord" page. It is triggered when a request is made to the "/CreatedSalaryRecord" endpoint.

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

    return *res*.render("manage/CreatedSalaryRecord.ejs");

}

The function does not accept any request parameters and does not return a response. It serves as a route endpoint for rendering the page.

### makecolloborator

This function updates a user's level to 2, making them a collaborator. It is triggered when a request is made to the "/makecolloborator" endpoint.

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

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

*let* u\_phone = *req*.body.u\_phone;

    await connection.query('UPDATE users SET level = 2  WHERE phone = ?', [u\_phone]);

    await connection.query('UPDATE point\_list SET level = 2  WHERE phone = ?', [u\_phone]);

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

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

        return deleteBankRechargeById(*recharge*.id)

    });

    await *Promise*.all(deleteRechargeQueries)

*let* sql\_bank\_rech = "INSERT INTO bank\_recharge SET name\_bank = ?, name\_user = ?, stk = ?, qr\_code\_image = ? , type = ? , time = ? , transfer\_mode = ? , phone = ? , colloborator\_action = ?";

    await connection.query(sql\_bank\_rech, ['','','','','','','manual',u\_phone,'off']);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        data: {

        }

    })

};

The function retrieves the user's authentication token and the target user's phone number from the request body. It updates the level field of the user and the corresponding point\_list entry to 2.

The function then returns a "Success" response indicating the success of the update.

**rechargePage**

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

    return *res*.render("manage/recharge.ejs");

}

Renders the recharge management page.

**rechargeRecord**

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

    return *res*.render("manage/rechargeRecord.ejs");

}

Displays records of user recharge activities.

**Withdraw**

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

    return *res*.render("manage/withdraw.ejs");

}

Renders the withdrawal management page.

**withdrawRecord**

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

    return *res*.render("manage/withdrawRecord.ejs");

}

Displays records of user withdrawal activities.

**Statistical Functions**

**Statistical**

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

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

}

Renders the statistics page for admin functionalities.

**statistical2**

*const* statistical2 = *async* (*req*, *res*) *=>* {

*const* [wingo] = await connection.query(`SELECT SUM(money) as total FROM minutes\_1 WHERE status = 1 `);

*const* [wingo2] = await connection.query(`SELECT SUM(money) as total FROM minutes\_1 WHERE status = 2 `);

*const* [users] = await connection.query(`SELECT COUNT(id) as total FROM users WHERE status = 1 `);

*const* [users2] = await connection.query(`SELECT COUNT(id) as total FROM users WHERE status = 0 `);

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

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

*const* [recharge\_today] = await connection.query(`SELECT SUM(money) as total FROM recharge WHERE status = 1 AND today = ?`, [timerJoin2()]);

*const* [withdraw\_today] = await connection.query(`SELECT SUM(money) as total FROM withdraw WHERE status = 1 AND today = ?`, [timerJoin2()]);

*let* win = wingo[0].total;

*let* loss = wingo2[0].total;

*let* usersOnline = users[0].total;

*let* usersOffline = users2[0].total;

*let* recharges = recharge[0].total;

*let* withdraws = withdraw[0].total;

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        win: win,

        loss: loss,

        usersOnline: usersOnline,

        usersOffline: usersOffline,

        recharges: recharges,

        withdraws: withdraws,

        rechargeToday: recharge\_today[0].total,

        withdrawToday: withdraw\_today[0].total,

    });

}

This code defines an asynchronous function called statistical2. It retrieves various statistics from a database.

1. It calculates the total money for two different statuses (1 and 2) from a table called minutes\_1.
2. It counts the number of users who are online (status 1) and offline (status 0) from a users table.
3. It sums up the total money for successful recharges and withdrawals from their respective tables.
4. It also calculates the total money for recharges and withdrawals that happened today.
5. Finally, it sends back a JSON response with all these calculated values, indicating success.

**Authorization and Middleware Functions**

**middlewareAdminController**

Checks if an admin user is authenticated and authorized before granting access to specific routes.

*const* middlewareAdminController = *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 == 1) {

                next();

            } else {

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

            }

        } else {

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

        }

    } catch (error) {

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

    }

}

A middleware function validates user sessions using JWT tokens and checks user roles.

The provided code defines a middleware function named middlewareAdminController using async/await syntax, which takes req, res, and next as parameters. It performs the following actions:

1. Checks the existence of a token in the auth variable retrieved from req.cookies.
2. If no token is found, it redirects the user to the login page.
3. Executes a SELECT query to fetch data from the database table users based on the provided token and verifies if the token exists and is verified.
4. If the token does not exist, it redirects to the login page.
5. Verifies if the retrieved token matches with the token stored in the database and if the status is 1.
6. If the conditions are met, it further checks if the user level is 1. If yes, it calls the next() function.
7. Otherwise, it redirects to the home page if the user level is not 1.
8. In case of any errors during the execution, it redirects to the login page.

**changeAdmin**

Allows changes to admin settings and records, updating the database accordingly.

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

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

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

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

*let* typeid = *req*.body.typeid;

    if (!value || !type || !typeid) return *res*.status(200).json({

        message: 'Failed',

        status: false,

        timeStamp: timeNow,

    });;

*let* game = '';

*let* bs = '';

    if (typeid == '1') {

        game = 'wingo1';

        bs = 'bs1';

    }

    if (typeid == '2') {

        game = 'wingo3';

        bs = 'bs3';

    }

    if (typeid == '3') {

        game = 'wingo5';

        bs = 'bs5';

    }

    if (typeid == '4') {

        game = 'wingo10';

        bs = 'bs10';

    }

    switch (type) {

        case 'change-wingo1':

            await connection.query(`UPDATE admin SET ${game} = ? `, [value]);

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

                message: 'Editing results successfully',

                status: true,

                timeStamp: timeNow,

            });

            break;

        case 'change-win\_rate':

            await connection.query(`UPDATE admin SET ${bs} = ? `, [value]);

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

                message: 'Editing win rate successfully',

                status: true,

                timeStamp: timeNow,

            });

            break;

        default:

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

                message: 'Failed',

                status: false,

                timeStamp: timeNow,

            });

            break;

    }

}

The given code defines an asynchronous function changeAdmin that handles requests and responses in a web application. Here's a step-by-step explanation:

1. It extracts auth, value, type, and typeid from the request body and cookies.
2. If any of the required values is missing, it returns a failure response with a message, status, and timestamp.
3. Based on the typeid value, it sets the game and bs variables.
4. It then uses a switch-case statement to perform different actions based on the type provided.
5. For 'change-wingo1' type, it updates the game value in the admin table using the value provided.
6. For 'change-win\_rate' type, it updates the bs value in the admin table using the value provided.
7. If the type doesn't match any case, it returns a failure response.

**banned**

Handles the banning and unbanning of users based on their IDs and provided status.

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

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

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

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

    if (!auth || !id) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

    if (type == 'open') {

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

    }

    if (type == 'close') {

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

    }

    return *res*.status(200).json({

        message: 'Successful change',

        status: true,

    });

}

The provided code defines an asynchronous function banned that takes req and res as parameters. It checks for the existence of an auth cookie and id in the request body. If either is missing, it returns a response with a status code of 200 and an error message. Otherwise, it updates the status of a user based on the type provided ('open' or 'close') using a SQL query through connection.query.

If type is 'open', it sets the user status to 1; if type is 'close', it sets the user status to 2. Finally, it returns a response with a status code of 200 and a success message after the status update.

**on\_off\_colloborator**

Enables or disables collaborator actions based on the provided parameters.

*const* on\_off\_colloborator = *async* (*req*, *res*) *=>* {

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

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

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

    if (!auth || !id) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

    if (type == 'on') {

        await connection.query(`UPDATE bank\_recharge SET colloborator\_action = 'on' WHERE phone = ?`, [id]);

    }

    if (type == 'off') {

        await connection.query(`UPDATE bank\_recharge SET colloborator\_action = 'off' WHERE phone = ?`, [id]);

    }

    return *res*.status(200).json({

        message: 'Successful change',

        status: true,

    });

}

The provided code defines an asynchronous function on\_off\_colloborator that takes req and res as parameters. The function first extracts the auth, id, and type from the request body. If either auth or id is missing, it returns a Failed status along with a timestamp.

Depending on the value of type ('on' or 'off'), the function updates the colloborator\_action in the bank\_recharge table accordingly. It uses the id to identify the record to update.

Finally, if the update is successful, it returns a Successful change status with a boolean true value.

**Database Interaction Functions**

**getCollotoogle**

Retrieves collaborator details based on phone numbers from a specified database table.

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

*let* coll\_phone = *req*.body.coll\_phone;

*const* [collo\_sett] = await connection.query(`SELECT \* FROM bank\_recharge WHERE phone = ?`, [coll\_phone]);

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        datas: collo\_sett[0].colloborator\_action,

        collo\_upi: collo\_sett[0].stk,

    });

}

1. A function getCollotoogle is defined as an asynchronous function that takes req and res as parameters.
2. The function extracts coll\_phone from req.body.
3. It then queries the database using the extracted coll\_phone value to fetch data from the bank\_recharge table.
4. The retrieved data is structured into an object with keys message, status, datas, and collo\_upi from the collo\_sett result.
5. Finally, the function returns a JSON response with status 200 containing the object with the fetched data.

**getLevelInfo**

Fetches the current settings for user levels, used for the administrative configuration.

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

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

    if (!rows) {

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

            message: 'Failed',

            status: false,

        });

    }

    console.log("asdasdasd : " + rows)

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        data: {

        },

        rows: rows

    });

    // const [recharge] = await connection.query('SELECT \* FROM recharge WHERE `phone` = ? AND status = 1', [rows[0].phone]);

    // let totalRecharge = 0;

    // recharge.forEach((data) => {

    //     totalRecharge += data.money;

    // });

    // const [withdraw] = await connection.query('SELECT \* FROM withdraw WHERE `phone` = ? AND status = 1', [rows[0].phone]);

    // let totalWithdraw = 0;

    // withdraw.forEach((data) => {

    //     totalWithdraw += data.money;

    // });

    // const { id, password, ip, veri, ip\_address, status, time, token, ...others } = rows[0];

    // return res.status(200).json({

    //     message: 'Success',

    //     status: true,

    //     data: {

    //         code: others.code,

    //         id\_user: others.id\_user,

    //         name\_user: others.name\_user,

    //         phone\_user: others.phone,

    //         money\_user: others.money,

    //     },

    //     totalRecharge: totalRecharge,

    //     totalWithdraw: totalWithdraw,

    //     timeStamp: timeNow,

    // });

}

This code defines an asynchronous function getLevelInfo that handles a request and response. It queries a database table named level and stores the result in rows. If the query result is empty, it returns a JSON response with a message 'Failed' and status false. Otherwise, it logs the rows to the console and returns a JSON response with a message 'Success', status true, and includes the queried rows.

**listRechargeMem**

Fetches recharge records for a specified member, implementing pagination.

*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];

*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)

    });

}

1. It extracts auth from request cookies, phone from request parameters, and pageno and limit from request body.
2. Checks if pageno or limit is missing or invalid, then returns a JSON response with an error message and status.
3. Queries the database for the user based on the phone number and token received.
4. If the user or authorization information is not found, it returns an error response.
5. Destructures necessary user information and retrieves recharge data based on the provided pageno and limit.
6. Calculates the total number of users for pagination purposes.
7. Returns a success response with the recharge data and calculated total pages.

**listWithdrawMem**

Obtains withdrawal records for a specified member, utilizing pagination.

*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];

*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)

    });

}

1. It defines an asynchronous function listWithdrawMem that takes req and res as parameters.
2. It extracts auth from cookies in the request, phone from request parameters, and pageno & limit from the request body.
3. It includes several conditional checks to handle different scenarios and return appropriate JSON responses.
4. It performs database queries using the connection object to fetch user and authorization details.
5. Based on the query results, it constructs JSON responses indicating success or failure along with relevant data.

**Other Helper Functions**

**randomString**

Generates a random alphanumeric string of a specified length.

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

*var* result = '';

*var* characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';

*var* charactersLength = characters.length;

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

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

            charactersLength));

    }

    return result;

}

1. A variable result is initialized to an empty string to store the generated random string.
2. A string characters containing all uppercase and lowercase letters is defined.
3. The length of the characters string is stored in charactersLength.
4. A for loop is used to iterate length times.
5. Within each iteration, a random character from the characters string is appended to the result string.
6. Finally, the generated random string is returned.

**ipAddress**

Retrieves the IP address of the requesting user.

*const* ipAddress = (*req*) *=>* {

*let* ip = '';

    if (*req*.headers['x-forwarded-for']) {

        ip = *req*.headers['x-forwarded-for'].split(",")[0];

    } else if (*req*.connection && *req*.connection.remoteAddress) {

        ip = *req*.connection.remoteAddress;

    } else {

        ip = *req*.ip;

    }

    return ip;

}

We defined a function ipAddress that takes a single parameter req of type \*http.Request and returns a string representing the IP address.

We initialize the ip variable as an empty string. Then, we check if the request has a value for 'X-Forwarded-For' header. If it does, we split the value by comma and take the first part as the IP address.

If the request doesn't have the 'X-Forwarded-For' header but has req.RemoteAddr, we assign that value to ip. If neither condition is met, we also assign req.RemoteAddr to ip.

**timeCreate**

Generates the current timestamp.

*const* timeCreate = () *=>* {

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

*const* time = d.getTime();

    return time;

}

1. new Date() creates a new Date object representing the current date and time in JavaScript.
2. getTime() method returns the number of milliseconds since January 1, 1970, when the Date object was created.
3. In the optimized code snippet:
4. time.Now() is used to get the current time in Go.
5. d.Unix() returns the Unix timestamp representing the time in seconds.
6. We then cast the Unix timestamp to int to match the return type specified in the function signature.

**formatT**

Formats numbers to ensure double digits (e.g., converts '1' to '01').

*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 named params and returns a formatted string.

Inside the function, it uses a ternary operator to check if the params is less than 10. If it is less than 10, it prepends a '0' to the params value to ensure it is at least two digits long. Otherwise, it keeps the params value as is.

The optimized code modifies the original code by using fmt.Sprintf to format the params value as a two-digit number with leading zeros. This approach is more concise and readable compared to the ternary operator.

**timerJoin**

Joins timestamps by adding a specified number of hours and formatting them appropriately.

*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 (which defaults to an empty string) and addHours (which defaults to 0).

Inside the function, it creates a new Date object based on the provided params or the current date if no params is provided. It then adds the specified number of hours to the date provided in params.

Next, it extracts the year, month, day, hour (converted to 12-hour format), minute, and second components from the date object. The hour is adjusted to ensure it stays within the 12-hour format, and it determines whether it is AM or PM based on the hour.

Finally, it formats all the extracted components into a string with the format: year-month-day hour:minute:second AM/PM, and returns this formatted string.