**Documentation for userController.js**

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

**Functions Overview**

The userController.js file contains several functions related to user authentication and information retrieval. It uses an external API to send verification codes via SMS, interacts with a MySQL database, and handles user-related operations like changing the username. The code is written in JavaScript and designed to work in the context of a web server. Also this code file contains several functions related to user authentication and account management, as well as functions to check and manage referrals. It is primarily used in a software project to handle password changes, referral calculations, and information retrieval regarding a user's team and promotions.

This code file consists of several functions related to payment, recharge, bank management, withdrawal, balance transfer, and transaction history. Let's go through each function in detail.

**Function: randomNumber(min, max)**

This function generates a random number within a specified range.

*const* randomNumber = (*min*, *max*) *=>* {

    return *String*(Math.floor(Math.random() \* (*max* - *min* + 1)) + *min*);

}

**Parameters:**

* min: The minimum value of the range (inclusive).
* max: The maximum value of the range (inclusive).

**Return Value:**

A string representation of the generated random number.

**Example Usage:**

const otp = randomNumber(100000, 999999);

console.log(otp); // Output: "512345"

**Function: verifyCode(req, res)**

This function verifies the authenticity of a user's account by checking the provided authentication token and sending a verification code via SMS.

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

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

*let* now = new *Date*().getTime();

*let* timeEnd = (+new *Date*) + 1000 \* (60 \* 2 + 0) + 500;

*let* otp = randomNumber(100000, 999999);

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

    if (!rows) {

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

            message: 'Account does not exist',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* user = rows[0];

    if (user.time\_otp - now <= 0) {

        request(`http://47.243.168.18:9090/sms/batch/v2?appkey=NFJKdK&appsecret=brwkTw&phone=84${user.phone}&msg=Your verification code is ${otp}&extend=${now}`, *async* (*error*, *response*, *body*) *=>* {

*let* data = JSON.parse(*body*);

            if (data.code == '00000') {

                await connection.execute("UPDATE users SET otp = ?, time\_otp = ? WHERE phone = ? ", [otp, timeEnd, user.phone]);

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

                    message: 'Submitted successfully',

                    status: true,

                    timeStamp: timeNow,

                    timeEnd: timeEnd,

                });

            }

        });

    } else {

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

            message: 'Send SMS regularly.',

            status: false,

            timeStamp: timeNow,

        });

    }

}

The provided code is an asynchronous function named verifyCode that takes req and res as parameters. It performs the following tasks:

1. Extracts the auth value from request cookies and sets the current timestamp and future timestamp.
2. Generates a random OTP in the range of 100000 to 999999.
3. Queries the database to find a user with the specified token (auth).
4. If the user does not exist, it returns a JSON response with an error message.
5. If the user exists and the OTP expiration time has not passed, it sends an SMS with the OTP to the user's phone number.
6. If the SMS is sent successfully, it updates the user's OTP and expiration time in the database and returns a success JSON response.
7. Otherwise, it returns a JSON response indicating that SMS sending should be done regularly.
8. The code contains a potential syntax error in the line where it sets the rows variable. The correct keyword should be const, not conswit.

**Parameters:**

* req: The request object containing user information.
* res: The response object for sending the server's response back to the client.

**Return Value:**

The function returns a JSON object as the server's response to the client.

**Example Usage:**

// Example Express route

app.post('/verify', verifyCode);

**Function: aviator(req, res)**

This function redirects the user to a URL based on their authentication token.

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

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

*res*.redirect(`https://247cashwin.cloud/theninja/src/api/userapi.php?action=loginandregisterbyauth&token=${auth}`);

    //res.redirect(`https://jetx.asia/#/jet/loginbyauth/${auth}`);

}

In this code snippet, a function aviator is defined using an arrow function syntax.

1. This function is asynchronous (declared with async) which means it can work with promises and allows the use of await inside it.
2. The function takes req and res as parameters, assuming it is from an Express route handler. It accesses the auth cookie from the req object using req.cookies.auth.
3. Then, it redirects the user to a specific URL constructed using the obtained token from the auth cookie. The final redirection URL is generated dynamically with the value of auth interpolated within it.

**Parameters:**

* req: The request object containing user information.
* res: The response object for sending the server's response back to the client.

**Return Value:**

None

**Example Usage:**

// Example Express route

app.get('/redirect', aviator);

**Function: userInfo(req, res)**

This function retrieves user information based on their authentication token.

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

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

    if (!rows) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*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,

            user\_avatar: others.avatar,

        },

        totalRecharge: totalRecharge,

        totalWithdraw: totalWithdraw,

        timeStamp: timeNow,

    });

}

The code defines an asynchronous function named userInfo that receives req (request) and res (response) as parameters.

1. It checks if the auth cookie is present in the request. If not, it returns a JSON response with a 'Failed' message, status as false, and a timestamp.
2. It then queries the database to fetch data about a user based on the token stored in the auth cookie. If no user data is found, it returns a similar 'Failed' JSON response.
3. Subsequent queries retrieve data about recharges and withdrawals for the user, calculate total recharge and withdrawal amounts, and extract specific user data from the fetched row.
4. Finally, it constructs a JSON response with a 'Success' message, status as true, user-specific data, total recharge and withdrawal amounts, and a timestamp.

**Parameters:**

* req: The request object containing user information.
* res: The response object for sending the server's response back to the client.

**Return Value:**

The function returns a JSON object as the server's response to the client.

**Example Usage:**

// Example Express route

app.get('/user', userInfo);

**Function: changeUser(req, res)**

This function allows the user to change their username based on their authentication token.

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

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

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

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

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

    if (!rows || !type || !name) return *res*.status(200).json({

        message: 'Failed',

        status: false,

        timeStamp: timeNow,

    });;

    switch (type) {

        case 'editname':

            await connection.query('UPDATE users SET name\_user = ? WHERE `token` = ? ', [name, auth]);

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

                message: 'Username modification successful',

                status: true,

                timeStamp: timeNow,

            });

            break;

        default:

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

                message: 'Failed',

                status: false,

                timeStamp: timeNow,

            });

            break;

    }

}

This code defines an asynchronous function changeUser that takes the req and res objects. It extracts auth, name, and type from the request body.

1. It then queries the database to find a user with the given token (auth).
2. If the query is unsuccessful or either name or type are missing, it returns a JSON response with a failure message, status, and a timestamp.
3. The code uses a switch statement to handle different values of type. If type is 'editname', it updates the name of the user in the database with the provided name for the matching token. It then returns a success JSON response with a corresponding message, status, and timestamp.
4. If the type does not match any case (default), it returns a failed JSON response with a message, status, and timestamp.

**Parameters:**

* req: The request object containing user information.
* res: The response object for sending the server's response back to the client.

**Return Value:**

The function returns a JSON object as the server's response to the client.

**Example Usage:**

// Example Express route

app.post('/change', changeUser);

## Function: changePassword

This function allows users to change their password by accepting their current password, new password, and authentication token. It performs the following steps:

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

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

*let* password = *req*.body.password;

*let* newPassWord = *req*.body.newPassWord;

    // let otp = req.body.otp;

    if (!password || !newPassWord) return *res*.status(200).json({

        message: 'Failed',

        status: false,

        timeStamp: timeNow,

    });;

*const* [rows] = await connection.query('SELECT \* FROM users WHERE `token` = ? AND `password` = ? ', [auth, md5(password)]);

    if (rows.length == 0) return *res*.status(200).json({

        message: 'Incorrect password',

        status: false,

        timeStamp: timeNow,

    });;

    // let getTimeEnd = Number(rows[0].time\_otp);

    // let tet = new Date(getTimeEnd).getTime();

    // var now = new Date().getTime();

    // var timeRest = tet - now;

    // if (timeRest <= 0) {

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

    //         message: 'Mã OTP đã hết hiệu lực',

    //         status: false,

    //         timeStamp: timeNow,

    //     });

    // }

    // const [check\_otp] = await connection.query('SELECT \* FROM users WHERE `token` = ? AND `password` = ? AND otp = ? ', [auth, md5(password), otp]);

    // if(check\_otp.length == 0) return res.status(200).json({

    //     message: 'Mã OTP không chính xác',

    //     status: false,

    //     timeStamp: timeNow,

    // });;

    await connection.query('UPDATE users SET otp = ?, password = ?, plain\_password = ? WHERE `token` = ? ', [randomNumber(100000, 999999), md5(newPassWord), newPassWord, auth]);

    return *res*.status(200).json({

        message: 'Password modification successful',

        status: true,

        timeStamp: timeNow,

    });

}

1. Checks if the current password and new password are provided. If either is missing, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Queries the database to find a user with the provided authentication token and the current password. If no user is found, the function returns a response with the message "Incorrect password", status "false", and a timestamp. Example:

* {
* "message": "Incorrect password",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Updates the user's password in the database with the new password and generates a new OTP (One-Time Password). Returns a response with the message "Password modification successful", status "true", and a timestamp. Example:

* {
* "message": "Password modification successful",
* "status": true,
* "timeStamp": [current-timestamp]
* }

## Function: checkInHandling

This function handles the check-in process for users by accepting the authentication token and data. It performs the following steps:

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

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

*let* data = *req*.body.data;

    if (!auth) return *res*.status(200).json({

        message: 'Failed',

        status: false,

        timeStamp: timeNow,

    });;

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

    if (!rows) return *res*.status(200).json({

        message: 'Failed',

        status: false,

        timeStamp: timeNow,

    });;

    if (!data) {

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

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

            message: 'No More Data',

            datas: point\_list,

            status: true,

            timeStamp: timeNow,

        });

    }

    if (data) {

        if (data == 1) {

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

*let* check = rows[0].money;

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

*let* get = 300;

            if (check >= data && point\_list.total1 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total1, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total1 = ? WHERE phone = ? ', [0, rows[0].phone]);

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

                    message: `You just received ₹ ${point\_list.total1}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total1 != 0) {

                return res.status(200).json({

                    message: 'Please Recharge ₹ 300 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total1 == 0) {

                return res.status(200).json({

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

        if (data == 2) {

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

*let* check = rows[0].money;

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

*let* get = 3000;

            if (check >= get && point\_list.total2 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total2, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total2 = ? WHERE phone = ? ', [0, rows[0].phone]);

                return res.status(200).json({

                    message: `You just received ₹ ${point\_list.total2}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total2 != 0) {

                return res.status(200).json({

                    message: 'Please Recharge ₹ 3000 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total2 == 0) {

                return res.status(200).json({

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

        if (data == 3) {

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

*let* check = rows[0].money;

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

*let* get = 6000;

            if (check >= get && point\_list.total3 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total3, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total3 = ? WHERE phone = ? ', [0, rows[0].phone]);

                return res.status(200).json({

                    message: `You just received ₹ ${point\_list.total3}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total3 != 0) {

                return res.status(200).json({

                    message: 'Please Recharge ₹ 6000 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total3 == 0) {

                return res.status(200).json({

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

        if (data == 4) {

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

*let* check = rows[0].money;

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

*let* get = 12000;

            if (check >= get && point\_list.total4 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total4, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total4 = ? WHERE phone = ? ', [0, rows[0].phone]);

                return res.status(200).json({

                    message: `You just received ₹ ${point\_list.total4}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total4 != 0) {

                return res.status(200).json({

                    message: 'Please Recharge ₹ 12000 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total4 == 0) {

                return res.status(200).json({

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

        if (data == 5) {

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

*let* check = rows[0].money;

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

*let* get = 28000;

            if (check >= get && point\_list.total5 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total5, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total5 = ? WHERE phone = ? ', [0, rows[0].phone]);

                return res.status(200).json({

                    message: `You just received ₹ ${point\_list.total5}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total5 != 0) {

                return res.status(200).json({

                    message: 'Please Recharge ₹ 28000 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total5 == 0) {

                return res.status(200).json({

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

        if (data == 6) {

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

*let* check = rows[0].money;

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

*let* get = 100000;

            if (check >= get && point\_list.total6 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total6, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total6 = ? WHERE phone = ? ', [0, rows[0].phone]);

                return res.status(200).json({

                    message: `You just received ₹ ${point\_list.total6}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total6 != 0) {

                return res.status(200).json({

                    message: 'Please Recharge ₹ 100000 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total6 == 0) {

                return res.status(200).json({

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

        if (data == 7) {

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

*let* check = rows[0].money;

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

*let* get = 200000;

            if (check >= get && point\_list.total7 != 0) {

                await connection.query('UPDATE users SET money = money + ? WHERE phone = ? ', [point\_list.total7, rows[0].phone]);

                await connection.query('UPDATE point\_list SET total7 = ? WHERE phone = ? ', [0, rows[0].phone]);

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

                    message: `You just received ₹ ${point\_list.total7}.00`,

                    status: true,

                    timeStamp: timeNow,

                });

            } else if (check < get && point\_list.total7 != 0) {

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

                    message: 'Please Recharge ₹200000 to claim gift.',

                    status: false,

                    timeStamp: timeNow,

                });

            } else if (point\_list.total7 == 0) {

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

                    message: 'You have already received this gift',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        };

    }

}

1. Checks if the authentication token is missing. If so, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Queries the database to find the user with the provided authentication token. If no user is found, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Checks if the data parameter is missing. If so, queries the database to fetch point\_list data associated with the user's phone number and returns a response with the message "No More Data", the fetched point\_list data, status "true", and a timestamp. Example:

* {
* "message": "No More Data",
* "datas": [point-list-data],
* "status": true,
* "timeStamp": [current-timestamp]
* }

1. Checks if the data parameter is provided. Based on the data value, the function performs different actions. The following cases are handled:
   * If data is equal to 1, checks the user's available money and the total1 value from the point\_list data. If the user has enough money and the total1 value is nonzero, updates the user's money and point\_list data, and returns a response with the message "You just received ₹ [point\_list.total1].00", status "true", and a timestamp. If the user does not have enough money, returns a response with the message "Please Recharge ₹ 300 to claim gift.", status "false", and a timestamp. If the total1 value is zero, returns a response with the message "You have already received this gift", status "false", and a timestamp. Example positive response:
   * {
     + "message": "You just received ₹ [point\_list.total1].00",
     + "status": true,
     + "timeStamp": [current-timestamp]
     + }
     + Example negative response (insufficient money):
     + {
       - "message": "Please Recharge ₹ 300 to claim gift.",
       - "status": false,
       - "timeStamp": [current-timestamp]
     + }
     + Example negative response (already received gift):
     + {
       - "message": "You have already received this gift",
       - "status": false,
       - "timeStamp": [current-timestamp]
     + }
   * If data is equal to 2, performs similar checks and updates for total2 value.
   * If data is equal to 3, performs similar checks and updates for total3 value.
   * If data is equal to 4, performs similar checks and updates for total4 value.
   * If data is equal to 5, performs similar checks and updates for total5 value.
   * If data is equal to 6, performs similar checks and updates for total6 value.
   * If data is equal to 7, performs similar checks and updates for total7 value.

## Function: formateT

This function formats a parameter by adding a leading zero if it is less than 10. The formatted parameter is returned as a string.

*function* formateT(*params*) {

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

    return result;

}

Example:

formateT(2); // Returns "02"

## Function: timerJoin

This function formats a timestamp parameter and adds a certain number of hours to it. The formatted timestamp is returned as a string in the format "yyyy-mm-dd hh:mm:ss AM/PM". If the parameter is not provided, the function uses the current timestamp. The additional hours can be specified using the addHours parameter, which defaults to 0 if not provided.

*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;

}

This JavaScript function timerJoin takes two parameters: params (default value '') and addHours (default value 0). It calculates a new date based on the provided parameters and adds hours to it. Then, it formats the date components (year, month, day, hour, minute, second) and returns a formatted date string.

Explanation of the code:

1. If params is provided, it converts it to a number and creates a new date, otherwise, it creates a new date with the current time.
2. Adds addHours to the hours of the date.
3. Formats the year, month, day, hour (in 12-hour format), minute, and second using the formatT function to ensure leading zeros.
4. Determines whether the time is in AM or PM based on the hour component.
5. Returns the formatted date string in the format: YYYY-MM-DD HH:MM:SS AM/PM.
6. The code includes a helper function formatT to add leading zeros to single-digit values for proper formatting.

Example:

timerJoin(1626450000000, 2); // Returns "2021-07-16 08:40:00 AM"

## Function: timerJoin1

This function is similar to the timerJoin function but returns only the date portion of the formatted timestamp in the format "yyyy-mm-dd".

*function* timerJoin1(*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;

}

The given code defines a function timerJoin1 that takes two parameters: params (optional, default value is an empty string) and addHours (optional, default value is 0).

Inside the function:

1. If params is provided, it converts params to a Number and creates a new Date object with that value. Otherwise, it creates a Date object with the current date and time.
2. Adds addHours to the hours of the date object.
3. Formats the year, month, and day of the date into two digits using formateT function (which must be defined elsewhere).
4. Calculates the hours in 12-hour format along with the AM/PM designation.
5. Formats the minutes and seconds into two digits each.
6. Returns the formatted date in the format 'YYYY-MM-DD'.

Example:

timerJoin1(1626450000000, 2); // Returns "2021-07-16"

## Function: promotion

This function retrieves and calculates various promotion-related data for a user based on the authentication token. It performs the following steps:

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*const* [user] = await connection.query('SELECT `phone`, `code`,`invite`, `roses\_f`, `roses\_f1`, `roses\_today`,`user\_level` FROM users WHERE `token` = ? ', [auth]);

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

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

*let* userInfo = user[0];

    // Directly referred level-1 users

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

    // Directly referred users today

*let* f1\_today = 0;

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

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

*let* check = (timerJoin1(f1\_time) == timerJoin1()) ? true : false;

        if (check) {

            f1\_today += 1;

        }

    }

    // All direct referrals today

*let* f\_all\_today = 0;

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

*const* f1\_code = f1s[i].code;

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

*let* check\_f1 = (timerJoin1(f1\_time) == timerJoin1()) ? true : false;

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

        // Total level-2 referrals today

*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;

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

*let* check\_f2 = (timerJoin1(f2\_time) == timerJoin1()) ? true : false;

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

            // Total level-3 referrals today

*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;

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

*let* check\_f3 = (timerJoin1(f3\_time) == timerJoin1()) ? true : false;

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

                // Total level-4 referrals today

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

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

*const* f4\_code = f4s[i].code;

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

*let* check\_f4 = (timerJoin1(f4\_time) == timerJoin1()) ? true : false;

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

                }

            }

        }

    }

    // Total level-2 referrals

*let* f2 = 0;

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

*const* f1\_code = f1s[i].code;

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

        f2 += f2s.length;

    }

    // Total level-3 referrals

*let* f3 = 0;

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

*const* f1\_code = f1s[i].code;

*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;

        }

    }

    // Total level-4 referrals

*let* f4 = 0;

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

*const* f1\_code = f1s[i].code;

*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;

            }

        }

    }

*let* selectedData = [];

*async* *function* fetchInvitesByCode(*code*, *depth* = 1) {

        if (*depth* > 6) {

            return;

        }

*const* [inviteData] = await connection.query('SELECT `id\_user`,`name\_user`,`phone`, `code`, `invite`, `rank`, `user\_level`, `total\_money` FROM users WHERE `invite` = ?', [*code*]);

        if (inviteData.length > 0) {

            for (*const* invite of inviteData) {

                selectedData.push(invite);

                await fetchInvitesByCode(invite.code, *depth* + 1);

            }

        }

    }

    if (f1s.length > 0) {

        for (*const* initialInfoF1 of f1s) {

            selectedData.push(initialInfoF1);

            await fetchInvitesByCode(initialInfoF1.code);

        }

    }

*const* rosesF1 = parseFloat(userInfo.roses\_f);

*const* rosesAll = parseFloat(userInfo.roses\_f1);

*let* rosesAdd = rosesF1 + rosesAll;

    return *res*.status(200).json({

        message: 'Receive success',

        level: level,

        info: user,

        status: true,

        invite: {

            f1: f1s.length,

            total\_f: selectedData.length,

            f1\_today: f1\_today,

            f\_all\_today: f\_all\_today,

            roses\_f1: userInfo.roses\_f1,

            roses\_f: userInfo.roses\_f,

            roses\_all: rosesAdd,

            roses\_today: userInfo.roses\_today,

        },

        timeStamp: timeNow,

    });

}

1. Checks if the authentication token is missing. If so, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Queries the database to find the user with the provided authentication token. If no user is found, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Queries the database to fetch the level data for promotions.
2. Calculates and retrieves various promotion-related data, such as the user's direct referrals (f1s), total direct referrals (total\_f), direct referrals today (f1\_today), total referrals today (f\_all\_today), and rose-related data (roses\_f1, roses\_f, roses\_all, roses\_today).
3. Returns a response with the fetched data and calculated values. Example:

* {
* "message": "Receive success",
* "level": [level-data],
* "info": [user-data],
* "status": true,
* "invite": {
* "f1": [f1-data],
* "total\_f": [selectedData-length],
* "f1\_today": [f1\_today],
* "f\_all\_today": [f\_all\_today],
* "roses\_f1": [user-roses\_f1],
* "roses\_f": [user-roses\_f],
* "roses\_all": [rosesAdd],
* "roses\_today": [user-roses\_today]
* },
* "timeStamp": [current-timestamp]
* }

## Function: myTeam

This function retrieves information about the user's team based on the authentication token. It performs the following steps:

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

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

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

    return *res*.status(200).json({

        message: 'Receive success',

        level: level,

        info: user,

        status: true,

        timeStamp: timeNow,

    });

}

1. Checks if the authentication token is missing. If so, the function returns a response with the message "Failed", status "false", and a timestamp. Example:
2. {
3. "message": "Failed",
4. "status": false,
5. "timeStamp": [current-timestamp]

}

1. Queries the database to find the user with the provided authentication token. If no user is found, the function returns a response with the message "Failed", status "false", and a timestamp. Example:
2. {
3. "message": "Failed",
4. "status": false,
5. "timeStamp": [current-timestamp]

}

1. Queries the database to fetch the level data for promotions.
2. Returns a response with the fetched data and calculated values. Example:
3. {
4. "message": "Receive success",
5. "level": [level-data],
6. "info": [user-data],
7. "status": true,
8. "timeStamp": [current-timestamp]

}

## Function: listMyTeam

This function retrieves a list of users in the user's team based on the authentication token. It performs the following steps:

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

*let* userInfo = user[0];

*const* [f1] = await connection.query('SELECT `id\_user`, `phone`, `code`, `invite`,`roses\_f`, `rank`, `name\_user`,`status`,`total\_money`, `time` FROM users WHERE `invite` = ? ORDER BY id DESC', [userInfo.code]);

*const* [mem] = await connection.query('SELECT `id\_user`, `phone`, `time` FROM users WHERE `invite` = ? ORDER BY id DESC LIMIT 100', [userInfo.code]);

*const* [total\_roses] = await connection.query('SELECT `f1`,`invite`, `code`,`phone`,`time` FROM roses WHERE `invite` = ? ORDER BY id DESC LIMIT 100', [userInfo.code]);

*const* selectedData = [];

*async* *function* fetchUserDataByCode(*code*, *depth* = 1) {

        if (*depth* > 6) {

            return;

        }

*const* [userData] = await connection.query('SELECT `id\_user`, `name\_user`, `phone`, `code`, `invite`, `rank`, `total\_money` FROM users WHERE `invite` = ?', [*code*]);

        if (userData.length > 0) {

            for (*const* user of userData) {

*const* [turnoverData] = await connection.query('SELECT `phone`, `daily\_turn\_over`, `total\_turn\_over` FROM turn\_over WHERE `phone` = ?', [user.phone]);

*const* [inviteCountData] = await connection.query('SELECT COUNT(\*) as invite\_count FROM users WHERE `invite` = ?', [user.code]);

*const* inviteCount = inviteCountData[0].invite\_count;

*const* userObject = {

                    ...user,

                    invite\_count: inviteCount,

                    user\_level: *depth*,

                    daily\_turn\_over: turnoverData[0]?.daily\_turn\_over || 0,

                    total\_turn\_over: turnoverData[0]?.total\_turn\_over || 0,

                };

                selectedData.push(userObject);

                await fetchUserDataByCode(user.code, *depth* + 1);

            }

        }

    }

    await fetchUserDataByCode(userInfo.code);

*let* newMem = [];

    mem.map((*data*) *=>* {

*let* objectMem = {

            id\_user: *data*.id\_user,

            phone: '91' + *data*.phone.slice(0, 1) + '\*\*\*\*' + *String*(*data*.phone.slice(-4)),

            time: *data*.time,

        };

        return newMem.push(objectMem);

    });

    return *res*.status(200).json({

        message: 'Receive success',

        f1: selectedData,

        f1\_direct: f1,

        mem: newMem,

        total\_roses: total\_roses,

        status: true,

        timeStamp: timeNow,

    });

}

1. Checks if the authentication token is missing. If so, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Queries the database to find the user with the provided authentication token. If no user is found, the function returns a response with the message "Failed", status "false", and a timestamp. Example:

* {
* "message": "Failed",
* "status": false,
* "timeStamp": [current-timestamp]
* }

1. Queries the database to fetch the users in the user's level-1 team (f1), the recently joined members (mem), and the total roses received (total\_roses). Also, defines an empty array selectedData.
2. Returns a response with the fetched data and the selectedData array. The selectedData array contains information about users in the user's team in a hierarchical structure. Example:

* {
* "message": "Receive success",
* "f1": [selectedData-for-level-1],
* "f1\_direct": [f1-data],
* "mem": [newMem],
* "total\_roses": [total\_roses],
* "status": true,
* "timeStamp": [current-timestamp]
* }

## wowpay()

This function handles the payment process. It retrieves the authorization token and money amount from the request. Currently, the code for fetching the user's mobile number from the database using the auth token is missing.

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

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

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

    // Fetching the user's mobile number from the database using auth token

    // Your existing controller code here

};

After fetching the user's mobile number, there is existing controller code that will handle the payment process.

Example usage:

wowpay(req, res);

## recharge()

This function handles the recharge process. It retrieves the authorization token, money amount, type, and typeid from the request. It also retrieves the minimum money value from the environment variables.

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

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

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

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

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

*const* minimumMoney = process.env.MINIMUM\_MONEY

    if (type != 'cancel') {

        if (!auth || !money || money < minimumMoney - 1) {

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

                message: 'Failed',

                status: false,

                timeStamp: timeNow,

            })

        }

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

    if (type == 'cancel') {

        await connection.query('UPDATE recharge SET status = 2 WHERE phone = ? AND id\_order = ? AND status = ? ', [userInfo.phone, typeid, 0]);

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

            message: 'Cancelled order successfully',

            status: true,

            timeStamp: timeNow,

        });

    }

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

    if (recharge.length == 0) {

*let* time = new Date().getTime();

*const* date = new Date();

*function* formateT(params) {

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

            return result;

        }

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

*let* date = '';

            if (params) {

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

            } else {

                date = new Date();

            }

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

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

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

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

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

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

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

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

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

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

        }

*let* checkTime = timerJoin(time);

*let* id\_time = date.getUTCFullYear() + '' + date.getUTCMonth() + 1 + '' + date.getUTCDate();

*let* id\_order = *Math*.floor(*Math*.random() \* (99999999999999 - 10000000000000 + 1)) + 10000000000000;

        // let vat = Math.floor(Math.random() \* (2000 - 0 + 1) ) + 0;

        money = *Number*(money);

*let* client\_transaction\_id = id\_time + id\_order;

*const* formData = {

            username: *process*.*env*.accountBank,

            secret\_key: *process*.*env*.secret\_key,

            client\_transaction: client\_transaction\_id,

            amount: money,

        }

        if (type == 'momo') {

*const* sql = `INSERT INTO recharge SET

            id\_order = ?,

            transaction\_id = ?,

            phone = ?,

            money = ?,

            type = ?,

            status = ?,

            today = ?,

            url = ?,

            time = ?`;

            await connection.execute(sql, [client\_transaction\_id, 'NULL', userInfo.phone, money, type, 0, checkTime, 'NULL', time]);

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

            return res.status(200).json({

                message: 'Received successfully',

                datas: recharge[0],

                status: true,

                timeStamp: timeNow,

            });

        }

*const* moneyString = money.toString();

*const* apiData = {

            key: "0c79da69-fdc1-4a07-a8b4-7135a0168385",

            client\_txn\_id: client\_transaction\_id,

            amount: moneyString,

            p\_info: 'WINGO PAYMENT',

            customer\_name: userInfo.name\_user,

            customer\_email: 'manas.xdr@gmail.com',

            customer\_mobile: userInfo.phone,

            redirect\_url: `https://247cashwin.cloud/wallet/verify/upi`,

            udf1: 'TIRANGA',

        };

        try {

*const* apiResponse = await axios.post('https://api.ekqr.in/api/create\_order', apiData);

            if (apiResponse.data.status == true) {

*const* sql = `INSERT INTO recharge SET

                id\_order = ?,

                transaction\_id = ?,

                phone = ?,

                money = ?,

                type = ?,

                status = ?,

                today = ?,

                url = ?,

                time = ?`;

                await connection.execute(sql, [client\_transaction\_id, '0', userInfo.phone, money, type, 0, checkTime, '0', timeNow]);

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

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

                    message: 'Received successfully',

                    datas: recharge[0],

                    payment\_url: apiResponse.data.data.payment\_url,

                    status: true,

                    timeStamp: timeNow,

                });

            } else {

                return *res*.status(500).json({ message: 'Failed to create order', status: false });

            }

        } catch (error) {

            return *res*.status(500).json({ message: 'API request failed', status: false });

        }

    } else {

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

            message: 'Received successfully',

            datas: recharge[0],

            status: true,

            timeStamp: timeNow,

        });

    }

}

The function performs several checks to ensure the validity of the request:

* Checks if the type is not equal to 'cancel' and the auth or money value is missing or less than the minimumMoney value, it returns a failed response.
* Checks if the type is 'cancel'. If true, it updates the status of the recharge record with the given typeid.

Next, the function checks if the recharge record exists for the user. If not, it generates necessary values for id\_time, id\_order, checkTime, and client\_transaction\_id using various helper functions.

If the type is 'momo', it inserts a new record into the recharge table and returns a success response along with the newly created recharge record.

For other types, it prepares the necessary data for API request and sends a POST request to the specified URL using axios. If the API response is successful, it inserts a new record into the recharge table and returns a success response along with the newly created recharge record and the payment URL from the API response. Otherwise, it returns a failed response.

Example usage:

recharge(req, res);

## cancelRecharge()

This function cancels a recharge order. It retrieves the authorization token from the request. If the token is missing, it returns a failed response.

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

    try {

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

        if (!auth) {

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

                message: 'Authorization is required to access this API!',

                status: false,

                timeStamp: timeNow,

            })

        }

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

        if (!user) {

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

                message: 'Authorization is required to access this API!',

                status: false,

                timeStamp: timeNow,

            })

        }

*let* userInfo = user[0];

*const* result = await connection.query('DELETE FROM recharge WHERE phone = ? AND status = ?', [userInfo.phone, 0]);

        if (result.affectedRows > 0) {

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

                message: 'All the pending recharges has been deleted successfully!',

                status: true,

                timeStamp: timeNow,

            })

        } else {

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

                message: 'There was no pending recharges for this user or delete operation has been failed!',

                status: true,

                timeStamp: timeNow,

            })

        }

    } catch (error) {

        console.error("API error: ", error)

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

            message: 'API Request failed!',

            status: false,

            timeStamp: timeNow,

        })

    }

}

The provided code defines an asynchronous function named cancelRecharge that handles canceling pending recharges for a user based on authorization information.

Here's a breakdown of the code:

1. The function checks for the presence of an auth cookie in the request object.
2. If the auth cookie is missing, it responds with an error message related to authorization.
3. It then queries the database to fetch user information based on the provided authentication token.
4. If no user is found for the token, it returns an authorization error response.
5. The function attempts to delete pending recharges for the user from the database.
6. Based on the deletion result, success or failure messages are sent back in the response.
7. If an error occurs during the process, it catches the error, logs it, and returns a general API request failure message.

Example usage:

cancelRecharge(req, res);

## addBank()

This function handles the addition of a bank account. It retrieves the authorization token, bank account details, and tinh from the request.

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

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

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

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

*let* stk = *req*.body.stk;

*let* email = *req*.body.email;

*let* sdt = *req*.body.sdt;

*let* tinh = *req*.body.tinh;

*let* time = new *Date*().getTime();

    if (!auth || !name\_bank || !name\_user || !stk || !email || !stk || !tinh ) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

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

*const* [user\_bank2] = await connection.query('SELECT \* FROM user\_bank WHERE phone = ? ', [userInfo.phone]);

    if (user\_bank.length == 0 && user\_bank2.length == 0) {

*const* sql = `INSERT INTO user\_bank SET

        phone = ?,

        name\_bank = ?,

        name\_user = ?,

        stk = ?,

        email = ?,

        sdt = ?,

        tinh = ?,

        time = ?`;

        await connection.execute(sql, [userInfo.phone, name\_bank, name\_user, stk, email, sdt, tinh, time]);

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

            message: 'Successfully added bank',

            status: true,

            timeStamp: timeNow,

        });

    } else if (user\_bank.length == 0) {

        await connection.query('UPDATE user\_bank SET tinh = ? WHERE phone = ? ', [tinh, userInfo.phone]);

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

            message: 'KYC Already Done',

            status: false,

            timeStamp: timeNow,

        });

    } else if (user\_bank2.length > 0) {

        await connection.query('UPDATE user\_bank SET name\_bank = ?, name\_user = ?, stk = ?, email = ?, sdt = ?, tinh = ?, time = ? WHERE phone = ?', [name\_bank, name\_user, stk, email, sdt, tinh, time, userInfo.phone]);

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

            message: 'your account is updated',

            status: false,

            timeStamp: timeNow,

        });

    }

}

The function performs basic validations to ensure all required fields are provided. If any required field is missing, it returns a failed response.

This code defines an asynchronous function addBank that handles a POST request to add bank details to a database. It retrieves necessary data from the request object, validates the input, performs database operations, and sends back appropriate responses based on the database queries' results.

Here's a breakdown of the code:

1. It extracts data from the request object such as authentication token, bank details, contact details, and location.
2. A timestamp timeNow is generated using new Date().getTime().
3. It checks for the presence of required data and returns a failure response if any of the mandatory fields are missing.
4. Queries the database to fetch user details based on the provided token (auth).
5. Handles different scenarios based on the database query results:

* If no user is found, it returns a failure response.
* Performs queries to check existing bank entries for the provided location and phone number.
* Inserts new bank details if no existing entries are found for both location and phone number.
* Updates the bank details if an existing entry matches the phone number.

Example usage:

addBank(req, res);

## infoUserBank()

This function retrieves the user's bank account information. It retrieves the authorization token and bank account type (e.g., "Bank" or "Pi") from the request.

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

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

*let* b\_type = "Bank";

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

*function* formateT(*params*) {

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

        return result;

    }

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

*let* date = '';

        if (*params*) {

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

        } else {

            date = new *Date*();

        }

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

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

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

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

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

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

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

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

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

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

    }

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

*let* checkTime = timerJoin(date);

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

*const* [minutes\_1] = await connection.query('SELECT \* FROM minutes\_1 WHERE phone = ?', [userInfo.phone]);

*const* [d5\_minute] = await connection.query('SELECT \* FROM result\_5d WHERE phone = ?', [userInfo.phone]);

*const* [k3\_minute] = await connection.query('SELECT \* FROM result\_k3 WHERE phone = ?', [userInfo.phone]);

*const* [trx\_minute] = await connection.query('SELECT \* FROM trx\_wingo\_bets WHERE phone = ?', [userInfo.phone]);

*let* total = 0;

    recharge.forEach((data) *=>* {

        total += parseFloat(data.money);

    });

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

*let* total\_w\_fee = 0;

*let* total\_k3\_fee = 0;

*let* total\_5d\_fee = 0;

*let* total\_trx\_fee = 0;

*let* fee = 0;

    minutes\_1.forEach((data) *=>* {

        total\_w += parseFloat(data.money);

        total\_w\_fee += parseFloat(data.fee);

    });

    k3\_minute.forEach((data) *=>* {

        total\_k3 += parseFloat(data.money);

        total\_k3\_fee += parseFloat(data.fee);

    });

    d5\_minute.forEach((data) *=>* {

        total\_5d += parseFloat(*data*.money);

        total\_5d\_fee += parseFloat(*data*.fee);

    });

    trx\_minute.forEach((*data*) *=>* {

        total\_trx += parseFloat(*data*.money);

        total\_trx\_fee += parseFloat(*data*.fee);

    });

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d) + parseInt(total\_trx);

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

*let* result = 0;

    if (total - total2 > 0) result = total - total2 - fee;

    result = Math.max(result, 0);

*var* [userBank] = '';

    if(b\_type == "Bank")

    {

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

    }

    else if(b\_type == "Pi")

    {

        [userBank] = await connection.query('SELECT \* FROM user\_bank WHERE phone = ? AND `name\_bank` = ?', [userInfo.phone,'Pi\_pay']);

    }

    return *res*.status(200).json({

        message: 'Received successfully',

        datas: userBank,

        userInfo: user,

        result: result,

        status: true,

        timeStamp: timeNow,

    });

}

The function fetches user data from the database using the auth token and performs necessary calculations on the user's recharge and betting data.

This code snippet defines an asynchronous function infoUserBank that retrieves and processes user bank information based on the provided request and response objects.

Here's a brief explanation of the code:

1. Checks for authentication using a cookie named auth.
2. Retrieves user information from a database table based on the token stored in auth.
3. Defines helper functions formateT and timerJoin to format timestamp and date/time values.
4. Fetches data from various database tables related to user transactions.
5. Calculates total amounts and fees for different transaction types.
6. Determines the final result based on the difference between total transactions and total fees.
7. Based on the b\_type variable, queries user bank information accordingly.
8. Returns a JSON response with the fetched user bank data, user information, result, and status.

Example usage:

infoUserBank(req, res);

## withdrawal3()

This function handles the withdrawal process. It retrieves the authorization token, withdrawal amount, and password from the request.

*const* withdrawal3 = *async* (*req*, *res*) *=>* {

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

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

*let* password = *req*.body.password;

    if (!auth || !money || !password || money < 299) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

*const* [user] = await connection.query('SELECT `phone`, `code`,`invite`, `money` FROM users WHERE `token` = ? AND password = ?', [auth, md5(password)]);

    if (user.length == 0) {

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

            message: 'incorrect password',

            status: false,

            timeStamp: timeNow,

        });

    };

*let* userInfo = user[0];

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

*let* id\_time = date.getUTCFullYear() + '' + date.getUTCMonth() + 1 + '' + date.getUTCDate();

*let* id\_order = Math.floor(Math.random() \* (99999999999999 - 10000000000000 + 1)) + 10000000000000;

*function* formateT(*params*) {

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

        return result;

    }

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

*let* date = '';

        if (params) {

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

        } else {

            date = new Date();

        }

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

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

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

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

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

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

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

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

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

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

    }

*let* dates = new Date().getTime();

*let* checkTime = timerJoin(dates);

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

*const* [minutes\_1] = await connection.query('SELECT \* FROM minutes\_1 WHERE phone = ?', [userInfo.phone]);

*const* [k3\_bet\_money] = await connection.query('SELECT \* FROM result\_k3 WHERE phone = ?', [userInfo.phone]);

*const* [d5\_bet\_money] = await connection.query('SELECT \* FROM result\_5d WHERE phone = ?', [userInfo.phone]);

*const* [trx\_bet\_money] = await connection.query('SELECT \* FROM trx\_wingo\_bets WHERE phone = ?', [userInfo.phone]);

*let* total = 0;

    recharge.forEach((data) *=>* {

        total += parseFloat(data.money);

    });

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

    minutes\_1.forEach((data) *=>* {

        total\_w += parseFloat(data.money);

    });

    k3\_bet\_money.forEach((data) *=>* {

        total\_k3 += parseFloat(data.money);

    });

    d5\_bet\_money.forEach((data) *=>* {

        total\_5d += parseFloat(data.money);

    });

    trx\_bet\_money.forEach((data) *=>* {

        total\_trx += parseFloat(data.money);

    });

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d)+ parseInt(total\_trx);

*let* result = 0;

    if (total - total2 > 0) result = total - total2;

    result = *Math*.max(result, 0);

*let* result2 = parseInt ((parseInt(total) \* 80)/100);

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

*const* [withdraw] = await connection.query('SELECT \* FROM withdraw WHERE `phone` = ? AND today = ?', [userInfo.phone, checkTime]);

    if (user\_bank.length != 0) {

        if (withdraw.length < 3) {

            if (userInfo.money - money >= 0) {

                if (total2 >= result2) {

                    if (total - total2 >= 0) {

                        if (total2 >= result2) {

                            return res.status(200).json({

                                message: 'The total bet is not enough to fulfill the request',

                                status: false,

                                timeStamp: timeNow,

                            });

                        }

                    } else {

*let* infoBank = user\_bank[0];

*const* sql = `INSERT INTO withdraw SET

                    id\_order = ?,

                    phone = ?,

                    money = ?,

                    stk = ?,

                    name\_bank = ?,

                    ifsc = ?,

                    name\_user = ?,

                    status = ?,

                    today = ?,

                    time = ?,

                    type = ?,

                    with\_type = ?`;

                        await connection.execute(sql, [id\_time + '' + id\_order, userInfo.phone, money, infoBank.stk, infoBank.name\_bank, infoBank.email, infoBank.name\_user, 0, checkTime, dates,'manual','bank']);

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

                        return res.status(200).json({

                            message: 'Withdrawal successful',

                            status: true,

                            money: userInfo.money - money,

                            timeStamp: timeNow,

                        });

                    }

                } else {

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

                        message: 'The total bet is not enough to fulfill the request',

                        status: false,

                        timeStamp: timeNow,

                    });

                }

            } else {

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

                    message: 'The balance is not enough to fulfill the request',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        } else {

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

                message: 'You can only make 2 withdrawals per day',

                status: false,

                timeStamp: timeNow,

            });

        }

    } else {

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

            message: 'Please link your bank first',

            status: false,

            timeStamp: timeNow,

        });

    }

}

The function performs several checks to ensure the validity of the request:

* Checks if the auth, money, or password values are missing, or if the money is less than 299, it returns a failed response.
* Fetches user data from the database using the auth token and checks if the user exists. If not, it returns a failed response.

It calculates the remaining balance by subtracting the withdrawal amount from the user's money. If the remaining balance is greater than or equal to 0, it updates the user's money value and inserts a new record into the withdraw table. It then returns a success response along with the updated money value.

Example usage:

withdrawal3(req, res);

## transfer()

This function handles the balance transfer process. It retrieves the authorization token, transfer amount, and receiver's phone number from the request.

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

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

*let* amount = *req*.body.amount;

*let* receiver\_phone = *req*.body.phone;

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

    // let id\_time = date.getUTCFullYear() + '' + (date.getUTCMonth() + 1) + '' + date.getUTCDate();

*let* id\_order = Math.floor(Math.random() \* (99999999999999 - 10000000000000 + 1)) + 10000000000000;

*let* time = new *Date*().getTime();

*let* client\_transaction\_id = id\_order;

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

*let* userInfo = user[0];

*let* sender\_phone = userInfo.phone;

*let* sender\_money = parseInt(userInfo.money);

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

*function* formateT(*params*) {

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

        return result;

    }

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

*let* date = '';

        if (*params*) {

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

        } else {

            date = new *Date*();

        }

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

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

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

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

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

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

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

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

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

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

    }

*let* dates = new *Date*().getTime();

*let* checkTime = timerJoin(dates);

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

*const* [minutes\_1] = await connection.query('SELECT \* FROM minutes\_1 WHERE phone = ? ', [userInfo.phone]);

*const* [k3\_bet\_money] = await connection.query('SELECT \* FROM result\_k3 WHERE phone = ?', [userInfo.phone]);

*const* [d5\_bet\_money] = await connection.query('SELECT \* FROM result\_5d WHERE phone = ?', [userInfo.phone]);

*const* [trx\_bet\_money] = await connection.query('SELECT \* FROM trx\_wingo\_bets WHERE phone = ?', [userInfo.phone]);

*let* total = 0;

    recharge.forEach((*data*) *=>* {

        total += parseFloat(*data*.money);

    });

*let* total2 = 0;

*let* total\_w = 0;

*let* total\_k3 = 0;

*let* total\_5d = 0;

*let* total\_trx = 0;

    minutes\_1.forEach((data) *=>* {

        total\_w += parseFloat(data.money);

    });

    k3\_bet\_money.forEach((data) *=>* {

        total\_k3 += parseFloat(data.money);

    });

    d5\_bet\_money.forEach((data) *=>* {

        total\_5d += parseFloat(data.money);

    });

    trx\_bet\_money.forEach((data) *=>* {

        total\_trx += parseFloat(data.money);

    });

    total2 += parseInt(total\_w) + parseInt(total\_k3) + parseInt(total\_5d)+ parseInt(total\_trx);

*let* result = 0;

    if (total - total2 > 0) result = total - total2;

*let* result2 = parseInt ((parseInt(total) \* 80)/100)

    if (total2 >= result2) {

        if (sender\_money >= amount) {

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

            if (receiver.length === 1 && sender\_phone !== receiver\_phone) {

*let* money = sender\_money - amount;

*let* total\_money = amount + receiver[0].total\_money;

*let* trans\_mode = '';

*const* [admin\_user] = await connection.query('SELECT \* FROM users WHERE level = ? ', [1]);

*let* adminInfo = admin\_user[0];

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

                trans\_mode = ad\_rows[0].transfer\_mode;

                if(trans\_mode == 'instant')

                {

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

                    await connection.query(`UPDATE users SET money = money + ? WHERE phone = ?`, [amount, receiver\_phone]);

*const* sql = "INSERT INTO balance\_transfer (sender\_phone, receiver\_phone, amount) VALUES (?, ?, ?)";

                    await connection.execute(sql, [sender\_phone, receiver\_phone, amount]);

*const* sql\_recharge = "INSERT INTO recharge (id\_order, transaction\_id, phone, money, type, status, today, url, time) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";

                    await connection.execute(sql\_recharge, [client\_transaction\_id, 0, receiver\_phone, amount, 'wallet', 1, checkTime, 0, time]);

*const* sql\_recharge\_with = "INSERT INTO withdraw (id\_order, phone, money, stk, name\_bank, name\_user, ifsc, sdt, tp, status, today, time, type,with\_type) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?,?,?,?,?,?)";

                    await connection.execute(sql\_recharge\_with, [client\_transaction\_id, sender\_phone, amount,0,0,0,0,0,0, 1, checkTime, time,trans\_mode,'transfer']);

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

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

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

*let* withdrdesc = "Amount of "+ amount + " have been transferred successfully.";

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

                    return res.status(200).json({

                        message: `Requested ${amount} sent successfully`,

                        curr\_user\_m:money,

                        transfer\_mode:trans\_mode,

                        status: true,

                        timeStamp: timeNow,

                    });

                }

                else{

*const* sql\_recharge\_with = "INSERT INTO withdraw (id\_order, phone, money, stk, name\_bank, name\_user, ifsc, sdt, tp, status, today, time, type,with\_type) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?,?,?,?,?,?)";

                    await connection.execute(sql\_recharge\_with, [client\_transaction\_id, sender\_phone, amount,0,0,0,0,0,0, 0, checkTime, time,trans\_mode,'transfer']);

*const* sql = "INSERT INTO balance\_transfer (sender\_phone, receiver\_phone, amount) VALUES (?, ?, ?)";

                    await connection.execute(sql, [sender\_phone, receiver\_phone, amount]);

*const* sql\_recharge = "INSERT INTO recharge (id\_order, transaction\_id, phone, money, type, status, today, url, time) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";

                    await connection.execute(sql\_recharge, [client\_transaction\_id, 0, receiver\_phone, amount, 'wallet', 0, checkTime, 0, time]);

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

                    await connection.query(sql\_noti, [userInfo.id, "Balance Transfer of Amount "+amount+" Initiated Successfully.", '0', "Transfer"]);

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

                        message: `Waiting for admin approval`,

                        curr\_user\_m:money,

                        transfer\_mode:trans\_mode,

                        status: true,

                        timeStamp: timeNow,

                    });

                }

            } else {

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

                    message: `${receiver\_phone} is not a valid user mobile number`,

                    status: false,

                    timeStamp: timeNow,

                });

            }

        } else {

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

                message: 'Your balance is not enough',

                status: false,

                timeStamp: timeNow,

            });

        }

    }

    else {

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

            message: 'The total bet is not enough to fulfill the request',

            status: false,

            timeStamp: timeNow,

        });

    }

}

The function performs validation checks to ensure that the sender has sufficient balance and the receiver's phone number is valid.

If the checks pass, it subtracts the transfer amount from the sender's balance and adds it to the receiver's balance. It also inserts records into the balance\_transfer, recharge, and withdraw tables, depending on the transfer type. If the transfer type requires admin approval, it returns a response indicating that the transfer is waiting for admin approval. Otherwise, it returns a success response.

Example usage:

transfer(req, res);

## transferHistory()

This function retrieves the balance transfer history for the user. It retrieves the authorization token from the request.

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

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

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

*const* [history] = await connection.query('SELECT \* FROM balance\_transfer WHERE sender\_phone = ?', [userInfo.phone]);

*const* [receive] = await connection.query('SELECT \* FROM balance\_transfer WHERE receiver\_phone = ?', [userInfo.phone]);

    if (receive.length > 0 || history.length > 0) {

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

            message: 'Success',

            receive: receive,

            datas: history,

            status: true,

            timeStamp: timeNow,

        });

    }

}

The function fetches user data from the database using the auth token and retrieves both sent and received transfer records from the balance\_transfer table. If there are any records, it returns a success response along with the transfer history data. Otherwise, it returns a response indicating that there is no available transfer history.

Example usage:

transferHistory(req, res);

## Function: recharge2

This function handles the recharge process. It takes a request (req) and response (res) object as parameters. It first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's phone, code, and invite information. It then checks if there is any pending recharge for the user. If there is a pending recharge, it returns a JSON response with the recharge details. Otherwise, it returns a 'Failed' message in a JSON response.

*const* recharge2 = *async* (*req*, *res*) *=>* {

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

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

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

    if (recharge.length != 0) {

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

            message: 'Received successfully',

            datas: recharge[0],

            infoBank: bank\_recharge,

            status: true,

            timeStamp: timeNow,

        });

    } else {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

}

This code defines an asynchronous function recharge2 that handles a POST request. It first gets the authentication token and money amount from the request object.

1. If there is no authentication token, it returns a failure response.
2. Then it queries the database to retrieve user information based on the token. If no user is found, it returns a failure response.
3. It further queries the database for any pending recharges and bank recharge information. If a recharge exists, it returns a success response with the recharge details and bank recharge information. Otherwise, it returns a failure response.
4. There are some issues in the code such as a syntax error in the SQL query for bank\_recharge where there is a typo in the query string. It should be SELECT \* FROM bank\_recharge WHERE phone = ?.

## Function: listRecharge

This function retrieves the recharge history of a user. It takes a request (req) and response (res) object as parameters. It first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's phone, code, and invite information. It then queries the database to retrieve the recharge history of the user and returns it in a JSON response.

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

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

    return *res*.status(200).json({

        message: 'Receive success',

        datas: recharge,

        status: true,

        timeStamp: timeNow,

    });

}

The provided code is an asynchronous function in JavaScript. It checks for authentication using a cookie named 'auth' from the request object.

1. If authentication fails (no 'auth' cookie), it returns a JSON response with a 'Failed' message, status 'false', and a timestamp.
2. Then it queries the database to retrieve user data based on the received 'auth' token and extracts phone, code, and invite information. If the user data is not found, it returns a similar 'Failed' JSON response.
3. Next, it retrieves recharge data from a 'recharge' table in the database for the user's phone number, orders it by id in descending order, and returns a JSON response with a 'Receive success' message, the fetched recharge data, status 'true' and a timestamp.
4. In the explanation, 'timeNow' variable is used but not defined in the provided code snippet. You may need to define or import it from another module for the code to work correctly.

## Function: search

This function searches for a user based on their phone number. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's phone, code, invite, and level information. Based on the user's level, it either retrieves the user's information or returns a 'Failed' message in a JSON response.

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

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

*const* [user] = await connection.query('SELECT `phone`, `code`,`invite`, `level` 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 (userInfo.level == 1) {

*const* [users] = await connection.query(`SELECT \* FROM users WHERE phone = ? ORDER BY id DESC `, [phone]);

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

            message: 'Receive success',

            datas: users,

            status: true,

            timeStamp: timeNow,

        });

    } else if (userInfo.level == 2) {

*const* [users] = await connection.query(`SELECT \* FROM users WHERE phone = ? ORDER BY id DESC `, [phone]);

        if (users.length == 0) {

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

                message: 'Receive success',

                datas: [],

                status: true,

                timeStamp: timeNow,

            });

        } else {

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

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

                    message: 'Receive success',

                    datas: users,

                    status: true,

                    timeStamp: timeNow,

                });

            } else {

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

                    message: 'Failed',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        }

    } else {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

}

The provided code is an asynchronous function (search) that handles a request and response objects. Here's a breakdown of the code:

1. It extracts 'auth' from cookies in the request and 'phone' from the request body.
2. If 'auth' does not exist, it returns a JSON response with a 'Failed' message, false status, and 'timeNow' timestamp.
3. It queries the database to find a user based on the 'auth' token.
4. If no user is found, it returns a similar failed JSON response.
5. It then checks the user's level. If level is 1, it queries the database for users with a specific phone number and returns a success JSON response with the retrieved users.
6. If the user level is 2, it again queries for users with a specific phone number. If no users are found, a success response with an empty data array is returned. Otherwise, it checks if the first user has a 'ctv' field matching the user's phone.
7. If the conditions are met, it returns a success response with the users; otherwise, it returns a failed response.
8. If the user level is neither 1 nor 2, it returns a generic failed response.

## Function: listWithdraw

This function retrieves the withdrawal history of a user. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's phone, code, and invite information. It then queries the database to retrieve the withdrawal history of the user and returns it in a JSON response.

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

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

    return *res*.status(200).json({

        message: 'Receive success',

        datas: recharge,

        status: true,

        timeStamp: timeNow,

    });

}

In this code snippet, a function listWithdraw is defined as an asynchronous function that takes two parameters req and res which represent the request and response objects respectively.

1. The function first extracts the auth value from the cookies of the request. If auth is falsy, a response with a failure message is sent back containing a status of false and a timestamp timeNow.
2. Then it queries the database to fetch user information based on the token received. If no user is found, it returns a similar failure response as before.
3. Next, it queries the database to get the recharge details for the specific user's phone number, ordered by id in descending order.
4. Finally, if all operations are successful, it sends a response with a message 'Receive success', the fetched recharge data in datas field, a status of true, and also includes the timestamp timeNow.

## Function: getnotificationCount

This function retrieves the count of unread notifications of a user. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's information. It then queries the database to retrieve the count of unread notifications of the user and returns it in a JSON response.

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

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

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

            N\_Count: 0,

        });

    }

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

*const* [notifications] = await connection.query(`SELECT COUNT(id) as total FROM notification WHERE recipient  = ? AND isread = ? `,[user[0]?.id, '0']);

*let* countNot = notifications[0].total;

    return *res*.status(200).json({

            message: 'Success',

            status: true,

            N\_Count: countNot,

        });

}

The given code defines an asynchronous function named getNotificationCount that takes req (request) and res (response) as parameters.

1. It first checks if the auth cookie exists in the request. If not, it returns a JSON response with a failure message, status 'false', current timestamp, and notification count '0'.
2. Then, it queries the database to get the user information based on the token extracted from the cookie. Subsequently, it queries the database to count the number of unread notifications for the user.
3. Finally, it constructs a success JSON response including a success message, status 'true', and the count of unread notifications retrieved from the database.
4. Make sure that the timeNow variable is defined and accessible within the scope of this function.

## Function: useRedenvelope

This function handles the usage of a redemption code (redenvelope). It takes a request (req) and response (res) object as parameters. It checks if the user is authenticated and the redemption code is provided in the request. If the user is not authenticated or the redemption code is not provided, it returns a JSON response with a 'Failed' message. If the user is authenticated and the redemption code is provided, it queries the database to retrieve the user's phone, code, and invite information. It then queries the database to retrieve the redemption code information. Based on the status of the redemption code, it updates the status, user's money, and inserts a record into the redenvelopes\_used table. It returns a JSON response with a success or failure message.

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

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

*let* code = *req*.body.code;

    if (!auth || !code) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

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

        'SELECT \* FROM redenvelopes WHERE id\_redenvelope = ?', [code]);

    if (redenvelopes.length == 0) {

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

            message: 'Redemption code error',

            status: false,

            timeStamp: timeNow,

        });

    } else {

*let* infoRe = redenvelopes[0];

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

*const* time = d.getTime();

        if (infoRe.status == 0) {

            await connection.query('UPDATE redenvelopes SET used = ?, status = ? WHERE `id\_redenvelope` = ? ', [0, 1, infoRe.id\_redenvelope]);

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

*let* sql = 'INSERT INTO redenvelopes\_used SET phone = ?, phone\_used = ?, id\_redenvelops = ?, money = ?, `time` = ? ';

            await connection.query(sql, [infoRe.phone, userInfo.phone, infoRe.id\_redenvelope, infoRe.money, time]);

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

                message: `Received successfully +${infoRe.money}`,

                status: true,

                timeStamp: timeNow,

            });

        } else {

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

                message: 'Gift code already used',

                status: false,

                timeStamp: timeNow,

            });

        }

    }

}

The code defines an asynchronous function useRedenvelope that takes req (request) and res (response) as parameters.

1. It checks for the presence of authentication and a code in the request. If either is missing, it returns a response with 'Failed', false status, and the current timestamp.
2. It then queries the database to fetch user information based on the authentication token.
3. If no user is found, it returns the failed response.
4. Further, it queries the redenvelopes table to find a red envelope based on the provided code.
5. If no red envelope is found, it returns an error response for 'Redemption code error'.
6. Otherwise, it proceeds to check the status of the red envelope. If the status is 0, it updates the red envelope status, adds money to the user's account, records the usage in another table, and returns a success response with the added money amount.
7. If the status is not 0, it returns a response stating that the gift code is already used.

## Function: callback\_bank

This function handles the callback from a bank transaction. It takes transaction details as parameters and updates the recharge status and user's money in the database accordingly. It returns a JSON response with a success or failure message.

*const* callback\_bank = *async* (*req*, *res*) *=>* {

*let* transaction\_id = *req*.body.transaction\_id;

*let* client\_transaction\_id = *req*.body.client\_transaction\_id;

*let* amount = *req*.body.amount;

*let* requested\_datetime = *req*.body.requested\_datetime;

*let* expired\_datetime = *req*.body.expired\_datetime;

*let* payment\_datetime = *req*.body.payment\_datetime;

*let* status = *req*.body.status;

    if (!transaction\_id) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

    if (status == 2) {

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

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

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

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

            message: 0,

            status: true,

        });

    } else {

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

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

            message: 'Cancellation successful',

            status: true,

            datas: recharge,

        });

    }

}

The provided code defines an asynchronous function callback\_bank that takes req and res as parameters.

1. It retrieves various data from the request body such as transaction\_id, client\_transaction\_id, amount, requested\_datetime, expired\_datetime, payment\_datetime, and status.
2. If transaction\_id is falsy, it returns a JSON response with a message 'Failed', status as false, and a timeStamp value.
3. If the status is equal to 2, it performs a series of database operations by updating the 'recharge' table, selecting data based on client\_transaction\_id, and updating the 'users' table accordingly. It then returns a JSON response with message 0 and a status of true.
4. Otherwise, if the status is not equal to 2, it updates the 'recharge' table with status 2 where id is undefined (should likely be corrected to client\_transaction\_id). It then returns a JSON response indicating the 'Cancellation successful', status as true, and includes some recharge data.

## Function: confirmRecharge

This function handles the confirmation of a recharge. It takes a request (req) and response (res) object as parameters. It checks if the user is authenticated and the required parameters (client\_txn\_id) are provided in the request. If the user is not authenticated or the required parameters are not provided, it returns a JSON response with a 'Failed' message. If the user is authenticated and the required parameters are provided, it queries the database to retrieve the user's phone, code, and invite information. It then queries an external API to check the status of the recharge. Based on the status of the recharge, it updates the recharge status, user's money, and returns a JSON response with a success or failure message.

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

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

    //let money = req.body.money;

    //let paymentUrl = req.body.payment\_url;

*let* client\_txn\_id = *req*.body?.client\_txn\_id;

    if (!client\_txn\_id) {

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

            message: 'client\_txn\_id is required',

            status: false,

            timeStamp: timeNow,

        })

    }

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

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

    if (recharge.length != 0) {

*const* rechargeData = recharge[0];

*const* date = new *Date*(rechargeData.today);

*const* dd = *String*(date.getDate()).padStart(2, '0');

*const* mm = *String*(date.getMonth() + 1).padStart(2, '0');

*const* yyyy = date.getFullYear();

*const* formattedDate = `${dd}-${mm}-${yyyy}`;

*const* apiData = {

            key: "0c79da69-fdc1-4a07-a8b4-7135a0168385",

            client\_txn\_id: client\_txn\_id,

            txn\_date: formattedDate,

        };

        try {

*const* apiResponse = await axios.post('https://api.ekqr.in/api/check\_order\_status', apiData);

            console.log(apiResponse.data)

*const* apiRecord = apiResponse.data.data;

            if (apiRecord.status === "scanning") {

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

                    message: 'Waiting for confirmation',

                    status: false,

                    timeStamp: timeNow,

                });

            }

            if (apiRecord.client\_txn\_id === rechargeData.id\_order && apiRecord.customer\_mobile === rechargeData.phone && apiRecord.amount === rechargeData.money) {

                if (apiRecord.status === 'success') {

                    await connection.query(`UPDATE recharge SET status = 1 WHERE id = ? AND id\_order = ? AND phone = ? AND money = ?`, [rechargeData.id, apiRecord.client\_txn\_id, apiRecord.customer\_mobile, apiRecord.amount]);

                    // const [code] = await connection.query(`SELECT invite, total\_money from users WHERE phone = ?`, [apiRecord.customer\_mobile]);

                    // const [data] = await connection.query('SELECT recharge\_bonus\_2, recharge\_bonus FROM admin WHERE id = 1');

                    // let selfBonus = info[0].money \* (data[0].recharge\_bonus\_2 / 100);

                    // let money = info[0].money + selfBonus;

*let* money = apiRecord.amount;

                    await connection.query('UPDATE users SET money = money + ?, total\_money = total\_money + ? WHERE phone = ? ', [money, money, apiRecord.customer\_mobile]);

                    // let rechargeBonus;

                    // if (code[0].total\_money <= 0) {

                    //     rechargeBonus = apiRecord.customer\_mobile \* (data[0].recharge\_bonus / 100);

                    // }

                    // else {

                    //     rechargeBonus = apiRecord.customer\_mobile \* (data[0].recharge\_bonus\_2 / 100);

                    // }

                    // const percent = rechargeBonus;

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

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

                        message: 'Successful application confirmation',

                        status: true,

                        datas: recharge,

                    });

                } else if (apiRecord.status === 'failure' || apiRecord.status === 'close') {

                    console.log(apiRecord.status)

                    await connection.query(`UPDATE recharge SET status = 2 WHERE id = ? AND id\_order = ? AND phone = ? AND money = ?`, [rechargeData.id, apiRecord.client\_txn\_id, apiRecord.customer\_mobile, apiRecord.amount]);

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

                        message: 'Payment failure',

                        status: true,

                        timeStamp: timeNow,

                    });

                }

            } else {

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

                    message: 'Mismtach data',

                    status: true,

                    timeStamp: timeNow,

                });

            }

        } catch (error) {

            console.error(error);

        }

    } else {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    }

}

The provided code defines an asynchronous function confirmRecharge that handles the confirmation of a recharge process. Here is a breakdown of the code:

1. The function extracts the auth and client\_txn\_id from the request object.
2. It then checks if the client\_txn\_id is provided and returns an error response if it's missing.
3. Similarly, it checks if the auth token is missing and returns an error response.
4. Queries the database to retrieve user information based on the token.
5. If the user information is not found, it returns a failure response.
6. Queries the database to fetch recharge data based on the user's phone number and status.
7. If recharge data is found, it proceeds to validate and process it:
   * Extracts relevant details from the recharge data and formats the date.
   * Makes an API call to check the order status.
   * If the status is 'scanning', it returns a waiting response.
   * Checks if the API data matches the recharge data and processes accordingly:
     + If the status is 'success', updates the recharge status in the database and adjusts the user's money.
     + If the status is 'failure' or 'close', updates the recharge status and returns a payment failure response.
   * If the data does not match, it returns a mismatch response.
8. If there is no recharge data found, it returns a failure response.

## Function: confirmUSDTRecharge

This function is commented out and not used in the code. It seems to be a work-in-progress or a future implementation for confirming USDT recharges. It is similar to the confirmRecharge function but appears to handle USDT-specific data.

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

    console.log(*res*?.body)

    console.log(*res*?.query)

    console.log(*res*?.cookies)

}

## Function: updateRecharge

This function handles the updating of a recharge. It takes a request (req) and response (res) object as parameters. It checks if the user is authenticated, the required parameters (money and order\_id) are provided, and the input data is valid. If the user is not authenticated, the required parameters are not provided, or the input data is not valid, it returns a JSON response with a 'Failed' message. If all the conditions are met, it queries the database to update the UTR (Unique Transaction Reference) of the recharge. It returns a JSON response with a success message.

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

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

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

*let* order\_id = *req*.body.id\_order;

*let* data = *req*.body.inputData;

    // if (type != 'upi') {

    //     if (!auth || !money || money < 300) {

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

    //             message: 'upi failed',

    //             status: false,

    //             timeStamp: timeNow,

    //         })

    //     }

    // }

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

*let* userInfo = user[0];

    if (!user) {

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

            message: 'user not found',

            status: false,

            timeStamp: timeNow,

        });

    };

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

*let* utrInfo = utr[0];

    if (!utrInfo) {

        await connection.query('UPDATE recharge SET utr = ? WHERE phone = ? AND id\_order = ?', [data, userInfo.phone, order\_id]);

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

            message: 'UTR updated',

            status: true,

            timeStamp: timeNow,

        });

    } else {

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

            message: 'UTR is already in use',

            status: false,

            timeStamp: timeNow,

        });

    }

}

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

1. It extracts values from req like auth, money, order\_id, and data.
2. It then queries the database to fetch user information based on the given auth. If the user is not found, it returns a JSON response indicating 'user not found'.
3. Next, it queries the database to check if the given data (UTR) exists in the recharge table. If it doesn't exist, it updates the recharge table with the new UTR and returns a success message. If the UTR already exists, it returns a message stating that it is already in use.

## Function: getnotifications

This function retrieves the notifications of a user. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's information. It then queries the database to retrieve the user's notifications and returns them in a JSON response.

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

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

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

*const* [rows] = await connection.query('SELECT \* FROM notification WHERE `recipient` = ? ORDER BY id DESC',[user[0]?.id]);

    if (!rows) {

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

            message: 'Failed',

            status: false,

        });

    }

    return *res*.status(200).json({

        message: 'Success',

        status: true,

        data: {

        },

        rows: rows

    })

};

The given code defines an asynchronous function getnotifications that takes two parameters req and res, presumably representing request and response objects.

1. It retrieves an authentication token from the request cookies and queries the database table users to find a user corresponding to that token.
2. Then, it queries the notification table to fetch notifications for the user found, ordered by id in descending order.
3. If no rows are returned, it responds with a JSON object containing a message 'Failed' and a status of false.
4. Otherwise, it responds with a JSON object containing a message 'Success', a status of true, an empty data object, and the retrieved rows from the query.

## Function: updatenotifications

This function updates the read status of notifications for a user. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it updates the read status of the notifications in the database and returns a JSON response with a success message.

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

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

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

    await connection.execute("UPDATE notification SET isread = ?  WHERE `recipient` = ? AND  isread = ? ", [1,user[0]?.id,'0'] );

    return *res*.status(200).json({

        message: 'Updated',

        status: true,

        });

};

In this code snippet, an async function updatenotifications is defined with two parameters req and res, which represent the request and response objects respectively.

1. The function first extracts the auth token from the request cookies. Then it queries the database to fetch a user based on the provided token using a prepared statement. The user data is stored in the user variable.
2. Subsequently, the code executes an SQL UPDATE statement to set the isread field to 1 for notifications where the recipient matches the user's ID and the isread field is currently 0.
3. Finally, a JSON response with a message 'Updated' and a status true is sent back with a 200 status code.

## Function: xpgain\_value

This function calculates the XP (Experience Points) gain of a user. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it queries the database to retrieve the user's information. It then calculates the XP gain value based on the user's staking activities and betting activities (minutes\_1, result\_k3, trx\_wingo\_bets). It returns a JSON response with the XP value, user's level, and status.

*const* xpgain\_value = *async* (*req*, *res*) *=>* {

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

*let* xp\_gain\_val = 0;

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

*const* [user\_stoke] = await connection.query('SELECT SUM(stake\_amnt) AS `sum` FROM claimed\_rewards WHERE status = 2 AND `reward\_id` = 136 AND `phone` = ?', [user[0].phone]);

*const* stakeamount = user\_stoke[0].sum || 0;

*const* [trx\_xp] = await connection.query(`SELECT SUM(money) as total FROM trx\_wingo\_bets WHERE phone = ? `, user[0].phone);

*const* [wingo\_xp] = await connection.query(`SELECT SUM(money) as total FROM minutes\_1 WHERE phone = ? `, user[0].phone);

*const* [k3\_xp] = await connection.query(`SELECT SUM(money) as total FROM result\_k3 WHERE phone = ? `, user[0].phone);

*const* [d5\_xp] = await connection.query(`SELECT SUM(money) as total FROM result\_5d WHERE phone = ? `, user[0].phone);

    if(wingo\_xp[0].total == null)

    {

        wingo\_xp[0].total = 0;

    }

    if(k3\_xp[0].total == null)

    {

        k3\_xp[0].total = 0;

    }

    if(d5\_xp[0].total == null)

    {

        d5\_xp[0].total = 0;

    }

    if(trx\_xp[0].total == null)

    {

        trx\_xp[0].total = 0;

    }

*let* level = user[0].level;

    xp\_gain\_val = parseInt(wingo\_xp[0].total) + parseInt(k3\_xp[0].total) + parseInt(d5\_xp[0].total) + parseInt(trx\_xp[0].total);

    return *res*.status(200).json({

        message: 'Successful',//Register Sucess

        xp\_value: xp\_gain\_val,

        level: level,

        status: true,

        language:user[0].lang\_code,

        user\_stake:stakeamount,

    });

};

The provided code is an asynchronous function in JavaScript using the `async` keyword. It calculates the total experience points gained by a user based on various queries to databases and then returns a JSON response containing information such as experience value, level, status, language, and user stake.

Here is a breakdown of the code:

1. It first retrieves the authentication token from the request cookies.
2. Performs several SQL queries to calculate the total stakes and XP values for the user.
3. Checks if any of the XP totals are null and sets them to 0 if they are.
4. Calculates the total XP gained by summing up the individual XP values.
5. Returns a JSON response with the calculated values and other user information.

## Function: getlang\_datacall

This function retrieves language data based on the language code provided in the request. It takes a request (req) and response (res) object as parameters. It retrieves the language code from the cookie and uses it to retrieve the corresponding language data. It returns a JSON response with the language data.

*const* getlang\_datacall = *async* (*req*, *res*) *=>* {

*let* lang\_code = *req*.cookies.lang;

*var* lang\_data = getlang\_data(lang\_code);

    return *res*.status(200).json({

        message: 'Successful',//Register Sucess

        status: true,

        data:lang\_data,

    });

}

The provided code is a JavaScript function that uses the async/await syntax to handle asynchronous operations. Here's a breakdown of the code:

1. const getlang\_datacall = async (req, res) => {: Declares an async function getlang\_datacall that takes req and res as parameters.
2. let lang\_code = req.cookies.lang;: Retrieves the value of lang from cookies and assigns it to lang\_code.
3. var lang\_data = getlang\_data(lang\_code);: Calls a function getlang\_data with lang\_code as an argument and stores the result in lang\_data.
4. return res.status(200).json({ message: 'Successful', status: true, data: lang\_data });: Constructs a JSON response with a message, status, and data extracted using lang\_data and sends it back with a 200 status.

## Function: getmybets

This function retrieves the betting details of a user. It takes a request (req) and response (res) object as parameters. Similar to other functions, it first checks if the user is authenticated by checking the auth cookie in the request. If the user is not authenticated, it returns a JSON response with a 'Failed' message. If the user is authenticated, it retrieves the user's phone number from the database. It then queries the database to retrieve the betting details (money, count, win amount) for different bet types (minutes\_1, result\_k3, result\_5d, trx\_wingo\_bets) based on the specified data type (day, yesterday, week, month). It calculates the total bet amount, total win amount, and win/loss label. It returns a JSON response with the betting details.

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

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

*let* type = *req*.body.data\_type;

    if (!auth) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        })

    }

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

*let* phone = user[0].phone;

    if (!user) {

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

            message: 'Failed',

            status: false,

            timeStamp: timeNow,

        });

    };

*var* wing\_betting\_list = '';

*var* k3\_betting\_list = '';

*var* d5\_betting\_list = '';

*var* trx\_betting\_list = '';

*var* total\_wingo\_bet = 0;

*var* total\_k3\_bet = 0;

*var* total\_trx\_bet = 0;

*var* total\_d5\_bet = 0;

*var* total\_wingo\_win\_amt = 0;

*var* total\_k3\_win\_amt = 0;

*var* total\_d5\_win\_amt = 0;

*var* total\_trx\_win\_amt = 0;

*var* total\_bet\_amt = 0;

*var* total\_win\_loss = 0 ;

*var* total\_win\_amt = 0;

*var* win\_loss\_label = "";

    if( type == 1)

    {

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

        [wing\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM minutes\_1 WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [k3\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_k3 WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [d5\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_5d WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [trx\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM trx\_wingo\_bets WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [[total\_wingo\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND `time`>= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_k3\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_k3 WHERE `phone` = ? AND `time` >= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_d5\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_5d WHERE `phone` = ? AND `time` >= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_trx\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND `time` >= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_wingo\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

        [[total\_k3\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_k3 WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

        [[total\_d5\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_5d WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

        [[total\_trx\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

*var* tt\_wingo\_bets = total\_wingo\_bet.sum || 0 ;

*var* tt\_k3\_bets = total\_k3\_bet.sum || 0 ;

*var* tt\_d5\_bets = total\_d5\_bet.sum || 0 ;

*var* tt\_trx\_bets = total\_trx\_bet.sum || 0 ;

*var* ttw\_wingo\_bets = total\_wingo\_win\_amt.sum || 0 ;

*var* ttw\_k3\_bets = total\_k3\_win\_amt.sum || 0 ;

*var* ttw\_d5\_bets = total\_d5\_win\_amt.sum || 0 ;

*var* ttw\_trx\_bets = total\_trx\_win\_amt.sum || 0 ;

        total\_bet\_amt +=  parseInt(tt\_wingo\_bets) + parseInt(tt\_k3\_bets) + parseInt(tt\_d5\_bets) + parseInt(tt\_trx\_bets);

        total\_win\_amt +=  parseInt(ttw\_wingo\_bets) + parseInt(ttw\_k3\_bets) + parseInt(ttw\_d5\_bets) + parseInt(ttw\_trx\_bets);

        total\_win\_loss = parseInt(total\_win\_amt) - parseInt(total\_bet\_amt);

        if(total\_win\_loss < 0)

        {

            win\_loss\_label = "LOSS";

        }

        else{

            win\_loss\_label = "WIN";

        }

    }

    else if(type == 2)

    {

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

        [wing\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM minutes\_1 WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [k3\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_k3 WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [d5\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_5d WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [trx\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM trx\_wingo\_bets WHERE `phone` = ? AND status NOT IN ( 0 ) AND `time` >= ? GROUP BY `stage` DESC;', [phone, time]);

        [[total\_wingo\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND `time`>= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_k3\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_k3 WHERE `phone` = ? AND `time` >= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_d5\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_5d WHERE `phone` = ? AND `time` >= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_trx\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND `time` >= ?  ORDER BY `id` DESC', [phone, time]);

        [[total\_wingo\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

        [[total\_k3\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_k3 WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

        [[total\_d5\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_5d WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

        [[total\_trx\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND `time` >= ? AND `status` = 1  ORDER BY `id` DESC', [phone, time]);

*var* tt\_wingo\_bets = total\_wingo\_bet.sum || 0 ;

*var* tt\_k3\_bets = total\_k3\_bet.sum || 0 ;

*var* tt\_d5\_bets = total\_d5\_bet.sum || 0 ;

*var* tt\_trx\_bets = total\_trx\_bet.sum || 0 ;

*var* ttw\_wingo\_bets = total\_wingo\_win\_amt.sum || 0 ;

*var* ttw\_k3\_bets = total\_k3\_win\_amt.sum || 0 ;

*var* ttw\_d5\_bets = total\_d5\_win\_amt.sum || 0 ;

*var* ttw\_trx\_bets = total\_trx\_win\_amt.sum || 0 ;

        total\_bet\_amt +=  parseInt(tt\_wingo\_bets) + parseInt(tt\_k3\_bets) + parseInt(tt\_d5\_bets) + parseInt(tt\_trx\_bets);

        total\_win\_amt +=  parseInt(ttw\_wingo\_bets) + parseInt(ttw\_k3\_bets) + parseInt(ttw\_d5\_bets) + parseInt(ttw\_trx\_bets);

        total\_win\_loss = parseInt(total\_win\_amt) - parseInt(total\_bet\_amt);

        if(total\_win\_loss < 0)

        {

            win\_loss\_label = "LOSS";

        }

        else{

            win\_loss\_label = "WIN";

        }

    }

    else if(type == 3)

    {

        [wing\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM minutes\_1 WHERE `phone` = ? AND status NOT IN ( 0 ) AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) GROUP BY `stage` DESC;', [phone]);

        [k3\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_k3 WHERE `phone` = ? AND status NOT IN ( 0 ) AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) GROUP BY `stage` DESC;', [phone]);

        [d5\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_5d WHERE `phone` = ? AND status NOT IN ( 0 ) AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) GROUP BY `stage` DESC;', [phone]);

        [trx\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM trx\_wingo\_bets WHERE `phone` = ? AND status NOT IN ( 0 ) AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) GROUP BY `stage` DESC;', [phone]);

        [[total\_wingo\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1)  ORDER BY `id` DESC', [phone]);

        [[total\_k3\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_k3 WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1)  ORDER BY `id` DESC', [phone]);

        [[total\_d5\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_5d WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1)  ORDER BY `id` DESC', [phone]);

        [[total\_trx\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1)  ORDER BY `id` DESC', [phone]);

        [[total\_wingo\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) AND `status` = 1  ORDER BY `id` DESC', [phone]);

        [[total\_k3\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_k3 WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) AND `status` = 1  ORDER BY `id` DESC', [phone]);

        [[total\_d5\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_5d WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) AND `status` = 1  ORDER BY `id` DESC', [phone]);

        [[total\_trx\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND YEARWEEK(`today`, 1) = YEARWEEK(CURDATE(), 1) AND `status` = 1  ORDER BY `id` DESC', [phone]);

*var* tt\_wingo\_bets = total\_wingo\_bet.sum || 0 ;

*var* tt\_k3\_bets = total\_k3\_bet.sum || 0 ;

*var* tt\_d5\_bets = total\_d5\_bet.sum || 0 ;

*var* tt\_trx\_bets = total\_trx\_bet.sum || 0 ;

*var* ttw\_wingo\_bets = total\_wingo\_win\_amt.sum || 0 ;

*var* ttw\_k3\_bets = total\_k3\_win\_amt.sum || 0 ;

*var* ttw\_d5\_bets = total\_d5\_win\_amt.sum || 0 ;

*var* ttw\_trx\_bets = total\_trx\_win\_amt.sum || 0 ;

        total\_bet\_amt +=  parseInt(tt\_wingo\_bets) + parseInt(tt\_k3\_bets) + parseInt(tt\_d5\_bets) + parseInt(tt\_trx\_bets);

        total\_win\_amt +=  parseInt(ttw\_wingo\_bets) + parseInt(ttw\_k3\_bets) + parseInt(ttw\_d5\_bets) + parseInt(ttw\_trx\_bets);

        total\_win\_loss = parseInt(total\_win\_amt) - parseInt(total\_bet\_amt);

        if(total\_win\_loss < 0)

        {

            win\_loss\_label = "LOSS";

        }

        else{

            win\_loss\_label = "WIN";

        }

    }

    else if(type == 4)

    {

        [wing\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM minutes\_1 WHERE `phone` = ? AND status NOT IN ( 0 ) AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) GROUP BY `stage` DESC;', [phone]);

        [k3\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_k3 WHERE `phone` = ? AND status NOT IN ( 0 ) AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) GROUP BY `stage` DESC;', [phone]);

        [d5\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM result\_5d WHERE `phone` = ? AND status NOT IN ( 0 ) AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) GROUP BY `stage` DESC;', [phone]);

        [trx\_betting\_list] = await connection.query('SELECT sum(money) as sum , count(\*) as count , sum(CASE WHEN status = "1" THEN get ELSE NULL END) AS win\_amt FROM trx\_wingo\_bets WHERE `phone` = ? AND status NOT IN ( 0 ) AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) GROUP BY `stage` DESC;', [phone]);

        [[total\_wingo\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE())  ORDER BY `id` DESC', [phone]);

        [[total\_k3\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_k3 WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE())  ORDER BY `id` DESC', [phone]);

        [[total\_d5\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM result\_5d WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE())  ORDER BY `id` DESC', [phone]);

        [[total\_trx\_bet]] = await connection.query('SELECT SUM(money) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE())  ORDER BY `id` DESC', [phone]);

        [[total\_wingo\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM minutes\_1 WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) AND `status` = 1  ORDER BY `id` DESC', [phone]);

        [[total\_k3\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_k3 WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) AND `status` = 1  ORDER BY `id` DESC', [phone]);

        [[total\_d5\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM result\_5d WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) AND `status` = 1  ORDER BY `id` DESC', [phone]);

        [[total\_trx\_win\_amt]] = await connection.query('SELECT SUM(get) AS `sum` FROM trx\_wingo\_bets WHERE `phone` = ? AND MONTH(`today`) = MONTH(CURRENT\_DATE()) AND YEAR(`today`) = YEAR(CURRENT\_DATE()) AND `status` = 1  ORDER BY `id` DESC', [phone]);

*var* tt\_wingo\_bets = total\_wingo\_bet.sum || 0 ;

*var* tt\_k3\_bets = total\_k3\_bet.sum || 0 ;

*var* tt\_d5\_bets = total\_d5\_bet.sum || 0 ;

*var* tt\_trx\_bets = total\_trx\_bet.sum || 0 ;

*var* ttw\_wingo\_bets = total\_wingo\_win\_amt.sum || 0 ;

*var* ttw\_k3\_bets = total\_k3\_win\_amt.sum || 0 ;

*var* ttw\_d5\_bets = total\_d5\_win\_amt.sum || 0 ;

*var* ttw\_trx\_bets = total\_trx\_win\_amt.sum || 0 ;

        total\_bet\_amt +=  parseInt(tt\_wingo\_bets) + parseInt(tt\_k3\_bets) + parseInt(tt\_d5\_bets) + parseInt(tt\_trx\_bets);

        total\_win\_amt +=  parseInt(ttw\_wingo\_bets) + parseInt(ttw\_k3\_bets) + parseInt(ttw\_d5\_bets) + parseInt(ttw\_trx\_bets);

        total\_win\_loss = parseInt(total\_win\_amt) - parseInt(total\_bet\_amt);

        if(total\_win\_loss < 0)

        {

            win\_loss\_label = "LOSS";

        }

        else{

            win\_loss\_label = "WIN";

        }

    }

    return *res*.status(200).json({

        message: 'Successful',//Register Sucess

        status: true,

        wingo\_betting: wing\_betting\_list,

        k3\_betting: k3\_betting\_list,

        trx\_betting: trx\_betting\_list,

        d5\_betting: d5\_betting\_list,

        total\_bet: total\_bet\_amt,

        winning\_amount:total\_win\_loss,

        win\_loss\_label:win\_loss\_label,

        no\_of\_bets:wing\_betting\_list.length + k3\_betting\_list.length + d5\_betting\_list.length + trx\_betting\_list.length

    });

}

This code defines an asynchronous function getmybets that handles a GET request to retrieve betting information based on different data types.

1. It first extracts the authorization token and data type from the request object. If there is no authorization token, it returns a 'Failed' message as JSON with a status of false.
2. The function then queries a database based on the data type provided to fetch betting information such as bet amounts, win amounts, and counts. The time is calculated based on the data type (today, yesterday, week, or month).
3. After processing the queries, it calculates the total bet amount, total win amount, total win/loss, and determines if the user has won or lost overall. It distinguishes a win or loss based on the total win/loss amount.
4. Finally, it constructs a JSON response with the retrieved betting information and relevant totals, and returns it with a 'Successful' message and a status of true.