**Payment Controller Documentation**

This code file contains functions related to payment handling in the software project.

**Variables**

timeNow

A variable that stores the current timestamp using the `Date.now()` function.

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

PaymentStatusMap

An object that maps payment statuses to their corresponding codes. The available payment statuses are:

*const* PaymentStatusMap = {

    PENDING: 0,

    SUCCESS: 1,

    CANCELLED: 2

}

PaymentMethodsMap

An object that maps payment methods to their corresponding codes. The available payment methods are:

*const* PaymentMethodsMap = {

    UPI\_GATEWAY: "upi\_gateway",

    UPI\_MANUAL: "upi\_manual",

    USDT\_MANUAL: "usdt\_manual",

    WOW\_PAY: "wow\_pay",

    USDT: "usdt",

}

**Functions**

**initiateUPIPayment**

A function that initiates a UPI payment.

**Parameters:** req (Request object), res (Response object)

**Returns:** JSON response with payment information and status.

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

*const* query = *req*.query

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

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

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

*let* bank\_recharge\_momo\_data

    if (bank\_recharge\_momo.length) {

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

    }

*const* momo = {

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

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

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

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

    }

*const* user\_admin\_invitor = await get\_user\_invitor(rows[0].phone);

*const* [bank\_recharge] = await connection.query("SELECT \* FROM bank\_recharge WHERE phone = ?", [user\_admin\_invitor]);

*var* upi\_address = '';

    if(bank\_recharge[0].colloborator\_action == "off")

    {

*const* [f\_admin] = await connection.query('SELECT \*  FROM users WHERE `level` = 1 ');

*var* invite\_phone = f\_admin[0].phone;

*const* [bank\_recharge12] = await connection.query("SELECT \* FROM bank\_recharge WHERE phone = ?", [invite\_phone]);

        upi\_address = bank\_recharge12[0].stk;

    }

    else{

        upi\_address = bank\_recharge[0].stk;

    }

    return *res*.render("wallet/manual\_payment.ejs", {

        Amount: query?.am,

        UpiId: momo.upi\_id,

        dynamic\_upi\_Addr:upi\_address,

    });

}

This code defines an asynchronous function initiateManualUPIPayment that takes req and res as input parameters. It performs several database queries to fetch user and payment information, then renders a specified EJS template with the retrieved data.

Here is a breakdown of the code:

1. Retrieves query parameters and cookie data from the request (req.query and req.cookies.auth).
2. Fetches user information from the database based on the authentication token (auth).
3. Queries the bank\_recharge table to get MOMO (Mobile Money) details associated with the user's phone number.
4. Checks if there is any MOMO data retrieved and extracts specific fields like bank name, username, UPI ID, and wallet address.
5. Calls the get\_user\_invitor function to get the invitor of the current user.
6. Queries the bank\_recharge table again based on the invitor's phone number to determine the UPI address for payment.
7. Based on a condition, it selects the UPI address either from the invitor's bank recharge data or the original user's bank recharge data.
8. Finally, it renders the manual\_payment.ejs template with the amount from the query, the UPI ID, and the dynamic UPI address obtained.

**Example:**

initiateUPIPayment({

body: {

money: 100

}

}, res);

**verifyUPIPayment**

A function that verifies the status of a UPI payment.

**Parameters:** req (Request object), res (Response object)

**Returns:** JSON response with payment verification status.

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

*const* type = PaymentMethodsMap.UPI\_GATEWAY

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

*let* orderId = *req*.query.client\_txn\_id;

    if (!auth || !orderId) {

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

            message: `orderId is Required!`,

            status: false,

            timeStamp: timeNow,

        })

    }

    try {

*const* user = await getUserDataByAuthToken(auth)

*const* recharge = await rechargeTable.getRechargeByOrderId({ orderId })

        if (!recharge) {

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

                message: `Unable to find recharge with this order id!`,

                status: false,

                timeStamp: timeNow,

            })

        }

*const* ekqrResponse = await axios.post('https://api.ekqr.in/api/check\_order\_status', {

            key: process.env.UPI\_GATEWAY\_PAYMENT\_KEY,

            client\_txn\_id: orderId,

            txn\_date: rechargeTable.getDMYDateOfTodayFiled(recharge.today),

        });

*const* ekqrData = ekqrResponse?.data

        if (ekqrData === undefined || ekqrData.status === false) {

            throw *Error*("Gateway error from ekqr!")

        }

        if (ekqrData.data.status === "created") {

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

                message: 'Your payment request is just created',

                status: false,

                timeStamp: timeNow,

            });

        }

        if (ekqrData.data.status === "scanning") {

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

                message: 'Waiting for confirmation',

                status: false,

                timeStamp: timeNow,

            });

        }

        if (ekqrData.data.status === 'success') {

            if (recharge.status === PaymentStatusMap.PENDING || recharge.status === PaymentStatusMap.CANCELLED) {

                await rechargeTable.setStatusToSuccessByIdAndOrderId({

                    id: recharge.id,

                    orderId: recharge.orderId

                })

                await addUserAccountBalance({

                    phone: user.phone,

                    money: recharge.money,

                    code: user.code,

                    invite: user.invite,

                })

            }

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

            //     status: true,

            //     message: "Payment verified",

            //     timestamp: timeNow

            // })

            return *res*.redirect("/wallet/rechargerecord")

        }

    } catch (error) {

        console.log(error)

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

            status: false,

            message: "Something went wrong!",

            timestamp: timeNow

        })

    }

}

The provided code is an asynchronous function named verifyUPIPayment that handles UPI payment verification. Here's a breakdown of the code:

1. const type = PaymentMethodsMap.UPI\_GATEWAY;: Defines a constant type with the value from PaymentMethodsMap.UPI\_GATEWAY.
2. let auth = req.cookies.auth; and let orderId = req.query.client\_txn\_id;: Retrieve auth token from cookies and orderId from the query parameters.
3. Validation check for auth and orderId. If either is missing, it returns a 400 status with an error message.
4. Attempt to fetch user data using the auth token and look up a recharge based on the orderId.
5. If the recharge is not found, it returns a 400 status with an appropriate error message.
6. It makes a POST request to an external API to check the order status using axios library.
7. Based on the response, different actions are taken like handling different status scenarios.
8. If the payment status is 'success', it updates the recharge status and adds money to the user's account.
9. Finally, it either redirects the user to a recharge record page or returns a generic error message if an exception occurs.

**Example:**

verifyUPIPayment(req, res);

**initiatePiPayment**

A function that initiates a payment using the Pi Pay system.

**Parameters:** req (Request object), res (Response object)

**Returns:** JSON response with payment information and status.

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

*const* type = PaymentMethodsMap.WOW\_PAY

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

*let* money = parseInt(*req*.query.money);

    // const minimumMoneyAllowed = parseInt(process.env.MINIMUM\_MONEY)

    // if (!money || !(money >= minimumMoneyAllowed)) {

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

    //         message: `Money is Required and it must be ₹${minimumMoneyAllowed} or above!`,

    //         status: false,

    //         timeStamp: timeNow,

    //     })

    // }

    try {

*const* user = await getUserDataByAuthToken(auth)

*const* query = *req*.query

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

*let* bank\_recharge\_momo\_data

        if (bank\_recharge\_momo.length) {

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

        }

*const* momo = {

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

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

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

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

        }

        return *res*.render("wallet/pipay.ejs", {

            Amount: query?.am,

            UsdtWalletAddress: momo.usdt\_wallet\_address,

        });

    } catch (error) {

        console.log(error)

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

            status: false,

            message: "Something went wrong!",

            timestamp: timeNow

        })

     }

}

The provided code defines an asynchronous function named 'initiatePiPayment' that handles a payment initiation process based on certain conditions and user data retrieved from the request object.

Here is the breakdown of the code:

1. It extracts the payment type from the constant 'PaymentMethodsMap' and initializes variables for authentication token ('auth') and money amount ('money').
2. Attempts to get user data based on the authentication token and extracts the query parameters from the request.
3. Performs a database query to fetch bank recharge data for a specific type and phone number.
4. Constructs a 'momo' object containing relevant bank recharge information like bank name, username, UPI ID, and USDT wallet address.
5. Renders a template 'pipay.ejs' passing the amount from the query and the USDT wallet address from the 'momo' object.
6. If any error occurs during the process, it logs the error and responds with a 500 status along with an error message.

**Example:**

initiatePiPayment({

query: {

money: 100

}

}, res);

**verifyPiPayment**

A function that verifies the status of a payment made using the Pi Pay system.

**Parameters:** req (Request object), res (Response object)

**Returns:** JSON response with payment verification status.

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

    try {

*const* type = PaymentMethodsMap.WOW\_PAY

*let* data = *req*.body;

        if (!*req*.body) {

            data = *req*.query;

        }

        console.log(data)

*let* merchant\_key = process.env.WOWPAY\_MERCHANT\_KEY;

*const* params = {

            mchId: process.env.WOWPAY\_MERCHANT\_ID,

            amount: data.amount || '',

            mchOrderNo: data.mchOrderNo || '',

            merRetMsg: data.merRetMsg || '',

            orderDate: data.orderDate || '',

            orderNo: data.orderNo || '',

            oriAmount: data.oriAmount || '',

            tradeResult: data.tradeResult || '',

            signType: data.signType || '',

            sign: data.sign || '',

        };

*let* signStr = "";

        signStr += "amount=" + params.amount + "&";

        signStr += "mchId=" + params.mchId + "&";

        signStr += "mchOrderNo=" + params.mchOrderNo + "&";

        signStr += "merRetMsg=" + params.merRetMsg + "&";

        signStr += "orderDate=" + params.orderDate + "&";

        signStr += "orderNo=" + params.orderNo + "&";

        signStr += "oriAmount=" + params.oriAmount + "&";

        signStr += "tradeResult=" + params.tradeResult;

*let* flag = wowpay.validateSignByKey(signStr, merchant\_key, params.sign);

        if (!flag) {

            console.log({

                status: false,

                message: "Something went wrong!",

                flag,

                timestamp: timeNow

            })

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

                status: false,

                message: "Something went wrong!",

                flag,

                timestamp: timeNow

            })

        }

*const* newRechargeParams = {

            orderId: params.mchOrderNo,

            transactionId: 'NULL',

            utr: null,

            phone: params.merRetMsg,

            money: params.amount,

            type: type,

            status: PaymentStatusMap.SUCCESS,

            today: rechargeTable.getCurrentTimeForTodayField(),

            url: 'NULL',

            time: timeNow,

        }

*const* recharge = await rechargeTable.getRechargeByOrderId({ orderId: newRechargeParams.orderId })

        if (!!recharge) {

            console.log({

                message: `Recharge already verified!`,

                status: true,

                timeStamp: timeNow,

            })

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

                message: `Recharge already verified!`,

                status: true,

                timeStamp: timeNow,

            })

        }

*const* newRecharge = await rechargeTable.create(newRechargeParams)

        await addUserAccountBalance({

            phone: user.phone,

            money: recharge.money,

            code: user.code,

            invite: user.invite,

        })

        return *res*.redirect("/wallet/rechargerecord")

    } catch (error) {

        console.log({

            status: false,

            message: "Something went wrong!",

            timestamp: timeNow

        })

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

            status: false,

            message: "Something went wrong!",

            timestamp: timeNow

        })

    }

}

The code provided is an asynchronous function verifyPiPayment that handles payment verification logic. Here is a breakdown of the key points:

1. const type = PaymentMethodsMap.WOW\_PAY: Declares a constant type with the value WOW\_PAY from PaymentMethodsMap.
2. let data = req.body;: Initializes data with the request body.
3. Checks if the request body is empty and assigns data to req.query if true.
4. Constructs a params object with various properties extracted from data or set to an empty string.
5. Generates a signature string signStr from the properties of params.
6. Validates the signature using wowpay.validateSignByKey with the merchant key and the sign from params.
7. If the validation fails, it logs an error message and returns a 400 status response with the error details.
8. Creates new recharge parameters for processing payment verification.
9. Checks if a recharge with the same order ID already exists and returns an appropriate response.
10. If no existing recharge found, it creates a new recharge entry and updates the user account balance.
11. Handles errors by logging and returning a 500 status response with an error message.

**Example:**

verifyPiPayment(req, res);

**initiateManualUPIPayment**

A function that initiates a manual UPI payment.

**Parameters:** req (Request object), res (Response object)

**Returns:** HTML rendering of the manual payment page.

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

*const* query = *req*.query

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

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

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

*let* bank\_recharge\_momo\_data

    if (bank\_recharge\_momo.length) {

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

    }

*const* momo = {

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

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

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

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

    }

*const* user\_admin\_invitor = await get\_user\_invitor(rows[0].phone);

*const* [bank\_recharge] = await connection.query("SELECT \* FROM bank\_recharge WHERE phone = ?", [user\_admin\_invitor]);

*var* upi\_address = '';

    if(bank\_recharge[0].colloborator\_action == "off")

    {

*const* [f\_admin] = await connection.query('SELECT \*  FROM users WHERE `level` = 1 ');

*var* invite\_phone = f\_admin[0].phone;

*const* [bank\_recharge12] = await connection.query("SELECT \* FROM bank\_recharge WHERE phone = ?", [invite\_phone]);

        upi\_address = bank\_recharge12[0].stk;

    }

    else{

        upi\_address = bank\_recharge[0].stk;

    }

    return *res*.render("wallet/manual\_payment.ejs", {

        Amount: query?.am,

        UpiId: momo.upi\_id,

        dynamic\_upi\_Addr:upi\_address,

    });

}

The provided code defines an asynchronous function initiateManualUPIPayment that takes two parameters req and res. It performs the following actions:

1. Extracts query parameters and authentication data from req.
2. Executes a SQL SELECT query to fetch data from the database table users based on the provided token and verification status.
3. Queries the bank\_recharge table to fetch details based on the user's phone number.
4. Constructs an object momo with bank details retrieved from bank\_recharge.
5. Invokes a function get\_user\_invitor to get the inviter of the user.
6. Queries the bank\_recharge table again for the inviter's details.
7. Based on a condition check, assigns a UPI address to upi\_address from different sources.
8. Renders a template manual\_payment.ejs passing data like Amount, UpiId, and dynamic UPI address to be displayed.

**Example:**

initiateManualUPIPayment(req, res);

**addManualUPIPaymentRequest**

A function that adds a manual UPI payment request to the system.

**Parameters:** req (Request object), res (Response object)

**Returns:** JSON response with the status of the payment request.

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

    try {

*const* data = *req*.body

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

*let* money = parseInt(data.money);

*let* utr = parseInt(data.utr);

*const* minimumMoneyAllowed = parseInt(process.env.MINIMUM\_MONEY)

        if (!money || !(money >= minimumMoneyAllowed)) {

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

                message: `Money is Required and it should be ₹${minimumMoneyAllowed} or above!`,

                status: false,

                timeStamp: timeNow,

            })

        }

        if (!utr && utr?.length != 12) {

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

                message: `UPI Ref No. or UTR is Required And it should be 12 digit long`,

                status: false,

                timeStamp: timeNow,

            })

        }

*const* user = await getUserDataByAuthToken(auth)

*const* pendingRechargeList = await rechargeTable.getRecordByPhoneAndStatus({ phone: user.phone, status: PaymentStatusMap.PENDING, type: PaymentMethodsMap.UPI\_GATEWAY })

        if (pendingRechargeList.length !== 0) {

*const* deleteRechargeQueries = pendingRechargeList.map(*recharge* *=>* {

                return rechargeTable.cancelById(*recharge*.id)

            });

            await *Promise*.all(deleteRechargeQueries)

        }

*const* orderId = getRechargeOrderId()

*const* newRecharge = {

            orderId: orderId,

            transactionId: 'NULL',

            utr: utr,

            phone: user.phone,

            money: money,

            type: PaymentMethodsMap.UPI\_MANUAL,

            status: 0,

            today: rechargeTable.getCurrentTimeForTodayField(),

            url: "NULL",

            time: timeNow,

        }

*const* recharge = await rechargeTable.create(newRecharge)

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

            message: 'Payment Requested successfully Your Balance will update shortly!',

            recharge: recharge,

            status: true,

            timeStamp: timeNow,

        });

    } catch (error) {

        console.log(error)

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

            status: false,

            message: "Something went wrong!",

            timestamp: timeNow

        })

    }

}

This code defines an asynchronous function addManualUPIPaymentRequest that takes req and res as parameters to handle a manual UPI payment request.

1. Within a try block, the function extracts data from the request body, parses integers for money and utr, and retrieves the minimum allowed money from an environment variable.
2. If the money is missing or less than the minimum allowed, a JSON response with an error message is sent.
3. If the utr is missing or not 12 digits long, another JSON response with an appropriate error message is sent.
4. It then fetches user data by authentication token, retrieves pending recharge records based on the user's phone number and specific payment method.
5. If there are any pending recharges, it iterates over them, cancels each recharge, and waits for all cancellation promises to complete.
6. Next, it generates an order ID, creates a new recharge object with provided details, and stores it in the database.
7. Finally, a success JSON response is sent with a message and the created recharge details.
8. If any error occurs during the process, an error message is logged, and a generic error response is sent back with a status code of 500.

**Example:**

addManualUPIPaymentRequest({

body: {

money: 100,

utr: 123456789012

}

}, res);

**addManualUSDTPaymentRequest**

A function that adds a manual USDT payment request to the system.

**Parameters:** req (Request object), res (Response object)

**Returns:** JSON response with the status of the payment request.

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

    try {

*const* data = *req*.body

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

*let* money\_usdt = parseInt(data.money);

*let* money = money\_usdt \* 92;

*let* utr = parseInt(data.utr);

*const* minimumMoneyAllowed = parseInt(process.env.MINIMUM\_MONEY)

        if (!money || !(money >= minimumMoneyAllowed)) {

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

                message: `Money is Required and it should be ₹${minimumMoneyAllowed} or ${(minimumMoneyAllowed / 92).toFixed(2)} or above!`,

                status: false,

                timeStamp: timeNow,

            })

        }

        if (!utr) {

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

                message: `Ref No. or UTR is Required`,

                status: false,

                timeStamp: timeNow,

            })

        }

*const* user = await getUserDataByAuthToken(auth)

*const* pendingRechargeList = await rechargeTable.getRecordByPhoneAndStatus({ phone: user.phone, status: PaymentStatusMap.PENDING, type: PaymentMethodsMap.UPI\_GATEWAY })

        if (pendingRechargeList.length !== 0) {

*const* deleteRechargeQueries = pendingRechargeList.map(*recharge* *=>* {

                return rechargeTable.cancelById(*recharge*.id)

            });

            await *Promise*.all(deleteRechargeQueries)

        }

*const* orderId = getRechargeOrderId()

*const* newRecharge = {

            orderId: orderId,

            transactionId: 'NULL',

            utr: utr,

            phone: user.phone,

            money: money,

            type: PaymentMethodsMap.USDT\_MANUAL,

            status: 0,

            today: rechargeTable.getCurrentTimeForTodayField(),

            url: "NULL",

            time: timeNow,

        }

*const* recharge = await rechargeTable.create(newRecharge)

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

            message: 'Payment Requested successfully Your Balance will update shortly!',

            recharge: recharge,

            status: true,

            timeStamp: timeNow,

        });

    } catch (error) {

        console.log(error)

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

            status: false,

            message: "Something went wrong!",

            timestamp: timeNow

        })

    }

}

The provided code is a function called addManualUSDTPaymentRequest that handles a POST request to add a manual USD payment. Here is a breakdown of the code:

1. It is an asynchronous function using async and awaits response using await.
2. It extracts data, auth, money, and utr from the request body and cookies.
3. It validates the money and utr against a minimum value retrieved from environment variables.
4. It fetches user data by auth token and checks for pending recharge requests.
5. If there are pending recharge requests, it cancels them before proceeding.
6. Generates a new order ID for the recharge and creates a new recharge object.
7. The final response sends a success message with the recharge details if successful, otherwise, it handles errors and sends an error response.

**Example:**

addManualUSDTPaymentRequest({

body: {

money: 10,

utr: 123456789012

}

}, res);

**initiateManualUSDTPayment**

A function that initiates a manual USDT payment.

**Parameters:** req (Request object), res (Response object)

**Returns:** HTML rendering of the manual USDT payment page.

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

*const* query = *req*.query

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

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

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

*let* bank\_recharge\_momo\_data

    if (bank\_recharge\_momo.length) {

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

    }

*const* momo = {

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

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

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

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

    }

    return *res*.render("wallet/usdt\_manual\_payment.ejs", {

        Amount: query?.am,

        UsdtWalletAddress: momo.usdt\_wallet\_address,

    });

}

The given code is an asynchronous function named initiateManualUSDTPayment. It takes req and res as parameters.

Here is a breakdown of the code:

1. It extracts query and auth from req and req.cookies.auth respectively.
2. It then executes a SELECT query to fetch tokens, level, and status from the users table based on the provided token and verification status.
3. Another SELECT query is executed to fetch data from the bank\_recharge table where the type is 'momo' and phone matches a value from the previously fetched data.
4. If there is data in bank\_recharge\_momo array, it assigns the first element to bank\_recharge\_momo\_data.
5. Creates an object momo with properties populated from bank\_recharge\_momo\_data.
6. Returns the result of rendering an ejs file along with the Amount and UsdtWalletAddress.

## get\_user\_invitor

This function retrieves the inviter's phone number based on the provided phone number. It takes the phone number as a parameter and returns the inviter's phone number as a result. The function performs multiple database queries to find the inviter's phone number by traversing through multiple levels of invitations. If the provided phone number is "8895203112", it returns the username as the inviter's phone number.

*const* get\_user\_invitor = *async* (*phone\_num*) *=>* {

*let* phone = *phone\_num*;

*let* invite\_phone = "";

*let* invite\_role = "";

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

    if(phone != "8895203112"){

*const* [f1s\_inv] = await connection.query('SELECT \* FROM users WHERE `code` = ? ', [f1s[0].invite]);

      if(parseInt(f1s\_inv[0].level) != 2 && parseInt(f1s\_inv[0].level) != 1)

      {

*const* [f2s\_inv] = await connection.query('SELECT \* FROM users WHERE `code` = ? ', [f1s\_inv[0].invite]);

        if(parseInt(f2s\_inv[0].level) != 2 && parseInt(f2s\_inv[0].level) != 1)

        {

*const* [f3s\_inv] = await connection.query('SELECT \* FROM users WHERE `code` = ? ', [f2s\_inv[0].invite]);

          if(parseInt(f3s\_inv[0].level) != 2 && parseInt(f3s\_inv[0].level) != 1)

          {

*const* [f4s\_inv] = await connection.query('SELECT \* FROM users WHERE `code` = ? ', [f3s\_inv[0].invite]);

            if(parseInt(f4s\_inv[0].level) != 2 && parseInt(f4s\_inv[0].level) != 1)

            {

*const* [f5s\_inv] = await connection.query('SELECT \* FROM users WHERE `code` = ? ', [f4s\_inv[0].invite]);

              if(parseInt(f5s\_inv[0].level) != 2 && parseInt(f5s\_inv[0].level) != 1)

              {

*const* [f6s\_inv] = await connection.query('SELECT \* FROM users WHERE `code` = ? ', [f5s\_inv[0].invite]);

                if(parseInt(f6s\_inv[0].level) != 2 && parseInt(f6s\_inv[0].level) != 1)

                {

*const* [f\_admin] = await connection.query('SELECT \*  FROM users WHERE `level` = 1 ');

                  invite\_role = 'admin';

                  invite\_phone = f\_admin[0].phone;

                }

                else{

                  invite\_role = f6s\_inv[0].level == 2 ? "colloborator" : "admin";

                  invite\_phone = f6s\_inv[0].phone ;

                }

              }

              else{

                invite\_role = f5s\_inv[0].level == 2 ? "colloborator" : "admin";

                invite\_phone = f5s\_inv[0].phone ;

              }

            }

            else{

              invite\_role = f4s\_inv[0].level == 2 ? "colloborator" : "admin";

              invite\_phone = f4s\_inv[0].phone ;

            }

          }

          else{

            invite\_role = f3s\_inv[0].level == 2 ? "colloborator" : "admin";

            invite\_phone = f3s\_inv[0].phone ;

          }

        }

        else{

          invite\_role = f2s\_inv[0].level == 2 ? "colloborator" : "admin";

          invite\_phone = f2s\_inv[0].phone ;

        }

      }

      else{

        invite\_role = f1s\_inv[0].level == 2 ? "colloborator" : "admin";

        invite\_phone = f1s\_inv[0].phone ;

      }

    }

    else{

      invite\_role = "admin";

      invite\_phone =  username;

    }

    return invite\_phone;

  }

The provided code defines an asynchronous function named get\_user\_invitor that takes a phone\_num parameter.

1. It queries a database table named users using the provided phone number and retrieves related information.
2. The function then checks if the phone number is not equal to a specific value '8895203112'. Based on this condition, it iterates through a series of nested queries to find the inviter user and their role until a certain level (2 or 1) is reached.
3. If the condition is met, it sets the invite\_role and invite\_phone accordingly. If the condition is not met or if the phone number matches '8895203112', it sets the role as admin and assigns username value to invite\_phone.
4. The optimized code refactors the nested if-else statements into a while loop to simplify the logic and reduce repetitive code. This improvement makes the function more concise and maintainable.

Example usage:

const inviterPhone = await get\_user\_invitor("9876543210");

console.log(inviterPhone); // Output: "8895203112"

## getUserDataByAuthToken

This function retrieves user data based on the provided authentication token. It takes the authentication token as a parameter and returns an object containing the user's phone number, code, username, and invitation code. The function performs a database query to fetch the user data using the authentication token. If the user data is not found, it throws an error.

*const* getUserDataByAuthToken = *async* (*authToken*) *=>* {

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

*const* user = users?.[0]

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

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

    }

    return {

        phone: user.phone,

        code: user.code,

        username: user.name\_user,

        invite: user.invite,

    }

}

In the provided code:

1. An asynchronous function getUserDataByAuthToken is defined that takes authToken as a parameter.
2. It utilizes await to asynchronously query the database connection to retrieve user data based on the provided authToken.
3. It then extracts the first element from the retrieved users array, if available, using optional chaining (users?.[0]).
4. If no user data is found (i.e., user is undefined or null), an error is thrown.
5. Finally, it returns an object containing the user data fields like phone, code, username, and invite based on the retrieved user information.

Example usage:

const authToken = "abc123";

const userData = await getUserDataByAuthToken(authToken);

console.log(userData); // Output: { phone: "9876543210", code: "USER123", username: "John Doe", invite: "INVITE123" }

## addUserAccountBalance

This function adds account balance to a user's account. It takes an object as a parameter containing the amount of money to be added, the user's phone number, and the inviter's code. The function calculates the user's money and inviter's money using the given amounts. It then updates the user's and inviter's account balances in the database. If the inviter's code is found, the function adds the inviter's money to their account as well. This function is executed asynchronously.

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

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

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

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

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

    if (inviter.length) {

        console.log(inviter)

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

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

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

    }

}

In this code snippet, there is an asynchronous function addUserAccountBalance that takes an object as a parameter containing money, phone, and invite.

Inside the function:

1. user\_money is calculated by adding 5% of the money input to itself.
2. inviter\_money is calculated as 5% of the money input.
3. An update query is executed on the database to increment the user's money and total money in the users table based on the provided phone.
4. A select query retrieves the phone number of the inviter based on the given invite code.
5. If an inviter is found, their money is incremented based on the inviter\_money amount.
6. The code also contains some console.log statements for logging purposes.

Example usage:

const accountBalanceData = {

money: 100,

phone: "9876543210",

invite: "INVITE123"

};

await addUserAccountBalance(accountBalanceData);

console.log("Successfully added account balance!");

## getRechargeOrderId

This function generates a recharge order ID using the current date and a randomly generated number. The function does not require any parameters and returns the generated recharge order ID. The order ID is a combination of the year, month, day, and a randomly generated number.

*const* getRechargeOrderId = () *=>* {

*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;

    return id\_time + id\_order

}

The given code defines a function getRechargeOrderId that generates a unique recharge order ID based on the current UTC date and a random number.

1. It first creates a new Date object to get the current date and time.
2. It then concatenates the year, month + 1 (months start from 0), and date to create a part of the order ID (id\_time).
3. Next, it generates a random order number between 10000000000000 and 99999999999999 (id\_order).
4. Finally, it returns the combination of id\_time and id\_order as the recharge order ID.

Example usage:

const orderId = getRechargeOrderId();

console.log(orderId); // Output: "20221113123456789"

## rechargeTable

This constant represents a collection of functions related to recharge records. It provides functions to get recharge records based on phone number and status, get a recharge record by order ID, cancel a recharge record by ID, set the status of a recharge record to success, get the current time for the "today" field, get the formatted date for the "today" field, and create a new recharge record. The functions are executed asynchronously.

### getRecordByPhoneAndStatus

This function retrieves recharge records based on the provided phone number, status, and type. It takes an object as a parameter containing phone number, status, and optional type. If the type is provided, the function retrieves records matching all three criteria. If the type is not provided, the function retrieves records matching the phone number and status only. The function performs a database query to fetch the recharge records and returns an array of objects containing various details of each record.

Example usage:

const recordData = {

phone: "9876543210",

status: "success",

type: "online"

};

### getRechargeByOrderId

This function retrieves a recharge record based on the provided order ID. It takes the order ID as a parameter and returns an object containing the details of the recharge record. The function performs a database query to fetch the recharge record using the order ID. If the record is not found, it returns null.

Example usage:

const orderId = "20221113123456789";

const record = await rechargeTable.getRechargeByOrderId({ orderId });

console.log(record);

/\* Output:

{

id: 1,

orderId: "20221113123456789",

transactionId: "TRANS123",

utr: "UTR123",

phone: "9876543210",

money: 100,

type: "online",

status: "success",

today: "2022-11-13",

url: "https://example.com",

time: "13:45:00"

}

\*/

### cancelById

This function cancels a recharge record based on the provided ID. It takes the ID as a parameter and updates the status of the recharge record to cancelled in the database. The function throws an error if the provided ID is not a number.

Example usage:

const recordId = 1;

await rechargeTable.cancelById(recordId);

console.log("Recharge record cancelled successfully!");

### setStatusToSuccessByIdAndOrderId

This function sets the status of a recharge record to success based on the provided ID and order ID. It takes an object as a parameter containing the ID and order ID. The function updates the status of the recharge record to success in the database. The function throws an error if the provided ID is not a number.

Example usage:

const recordData = {

id: 1,

orderId: "20221113123456789"

};

await rechargeTable.setStatusToSuccessByIdAndOrderId(recordData);

console.log("Recharge record status updated to success!");

### getCurrentTimeForTodayField

This function returns the current time in the format "YYYY-DD-MM h:mm:ss A" for the "today" field of a recharge record. The function does not require any parameters.

Example usage:

const currentTime = rechargeTable.getCurrentTimeForTodayField();

console.log(currentTime); // Output: "2022-11-13 13:45:00 PM"

### getDMYDateOfTodayFiled

This function formats the provided "today" field of a recharge record to the "DD-MM-YYYY" format. It takes the "today" field as a parameter and returns the formatted date.

Example usage:

const today = "2022-11-13 13:45:00";

const formattedDate = rechargeTable.getDMYDateOfTodayFiled(today);

console.log(formattedDate); // Output: "13-11-2022"

### create

This function creates a new recharge record in the database. It takes an object as a parameter containing the details of the new record. The function inserts the record into the database and returns the created recharge record. If the URL is not provided, it defaults to "0". The function throws an error if the record is not created successfully.

Example usage:

const newRecharge = {

orderId: "20221113123456789",

*const* rechargeTable = {

    getRecordByPhoneAndStatus: *async* ({ *phone*, *status*, *type* }) *=>* {

        if (![PaymentStatusMap.SUCCESS, PaymentStatusMap.CANCELLED, PaymentStatusMap.PENDING].includes(*status*)) {

            throw *Error*("Invalid Payment Status!")

        }

*let* recharge

        if (*type*) {

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

        } else {

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

        }

        return recharge.map((*item*) *=>* ({

            id: *item*.id,

            orderId: *item*.id\_order,

            transactionId: *item*.transaction\_id,

            utr: *item*.utr,

            phone: *item*.phone,

            money: *item*.money,

            type: *item*.type,

            status: *item*.status,

            today: *item*.today,

            url: *item*.url,

            time: *item*.time,

        }))

    },

    getRechargeByOrderId: *async* ({ *orderId* }) *=>* {

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

        if (recharge.length === 0) {

            return null

        }

        return recharge.map((*item*) *=>* ({

            id: *item*.id,

            orderId: *item*.id\_order,

            transactionId: *item*.transaction\_id,

            utr: *item*.utr,

            phone: *item*.phone,

            money: *item*.money,

            type: *item*.type,

            status: *item*.status,

            today: *item*.today,

            url: *item*.url,

            time: *item*.time,

        }))?.[0]

    },

    cancelById: *async* (*id*) *=>* {

        if (typeof *id* !== "number") {

            throw *Error*("Invalid Recharge 'id' expected a number!")

        }

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

    },

    setStatusToSuccessByIdAndOrderId: *async* ({ *id*, *orderId* }) *=>* {

        if (typeof *id* !== "number") {

            throw *Error*("Invalid Recharge 'id' expected a number!")

        }

        console.log(*id*, *orderId*)

*const* [re] = await connection.query('UPDATE recharge SET status = 1 WHERE id = ? AND id\_order = ?', [*id*, *orderId*]);

        console.log(re)

    },

    getCurrentTimeForTodayField: () *=>* {

        return moment().format("YYYY-DD-MM h:mm:ss A")

    },

    getDMYDateOfTodayFiled: (*today*) *=>* {

        return moment(*today*, "YYYY-DD-MM h:mm:ss A").format("DD-MM-YYYY")

    },

    create: *async* (*newRecharge*) *=>* {

        if (*newRecharge*.url === undefined || *newRecharge*.url === null) {

*newRecharge*.url = "0"

        }

        await connection.query(

            `INSERT INTO recharge SET id\_order = ?, transaction\_id = ?, phone = ?, money = ?, type = ?, status = ?, today = ?, url = ?, time = ?, utr = ?`,

            [*newRecharge*.orderId, *newRecharge*.transactionId, *newRecharge*.phone, *newRecharge*.money, *newRecharge*.type, *newRecharge*.status, *newRecharge*.today, *newRecharge*.url, *newRecharge*.time, *newRecharge*?.utr || "NULL"]

        );

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

        if (recharge.length === 0) {

            throw *Error*("Unable to create recharge!")

        }

        return recharge[0]

    }

}

The provided code is defining a JavaScript object called 'rechargeTable' which contains several async functions to interact with a 'recharge' table in a database. Here is a breakdown of the key functions:

1. **getRechargeRecordByPhoneAndStatus:** Accepts 'phone', 'status', and 'type' parameters, checks for valid 'status', and queries the database for records based on the provided parameters.
2. **getRechargeByOrderId:** Accepts 'orderId', queries the database for records matching the order ID, and returns the first item found.
3. **cancelById:** Accepts an 'id', checks if it's a number, and updates the status of the corresponding record in the database.
4. **setStatusToSuccessByIdAndOrderId:** Accepts 'id' and 'orderId', validates 'id', and updates the status of the corresponding record in the database.
5. **getCurrentTimeForTodayField:** Returns the current date and time formatted as 'YYYY-DD-MM h:mm:ss A'.
6. **getDMYDateOfTodayField:** Accepts a 'today' parameter, converts it to a different date format, and returns it.
7. **create:** Accepts a 'newRecharge' object, inserts a new record into the database, and returns the created record.
8. The code is structured to handle database operations related to recharges with error handling and proper data manipulation for insertion and retrieval.