**Documentation for accountController.js**

This code file contains the implementation of the account controller for a software project.

**Import Statements**

The code file starts with import statements for various libraries and dependencies:

* connection - imported from "../config/connectDB"
* jwt - imported from 'jsonwebtoken'
* md5 - imported from "md5"
* request - imported from 'request'
* dotenv - imported from "dotenv/config"
* e - imported from "express"

**Helper Functions**

The code file contains several helper functions that are used throughout the implementation:

**randomString(length)**

This function generates a random string of the specified length.

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

*var* result = '';

*var* characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';

*var* charactersLength = characters.length;

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

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

            charactersLength));

    }

    return result;

}

The given code defines a function randomString that takes a parameter length and generates a random string of alphabets (both uppercase and lowercase) of the specified length.

Here's a breakdown of the code:

1. strings.Builder is used to efficiently build strings.
2. The characters pool consists of uppercase and lowercase alphabets.
3. The function loops length number of times and appends a random character from the pool to the result string.
4. Finally, the generated random string is returned.

**Example usage:**

var random = randomString(10);

console.log(random);

// Output: "IowxlUtndY"

**randomNumber(min, max)**

This function generates a random number between the specified minimum and maximum values.

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

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

}

The provided code defines a function randomNumber that takes two parameters min and max. Inside the function, it generates a random number between min (inclusive) and max (inclusive). The random number is obtained by multiplying a random floating-point number between 0 and 1 by the difference between max and min, adding min to it, flooring the result to get an integer, and finally converting it to a string before returning.

**Example usage:**

var number = randomNumber(1, 100);

console.log(number);

// Output: "42"

**isNumber(params)**

This function checks whether the given parameter is a number or not.

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

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

    return pattern.test(*params*);

}

The given code defines a function called isNumber which takes a parameter params and checks if the input is a number using a regular expression.

The regular expression /^[0-9]\*\d$/ ensures that the input contains only numeric characters.

The function then uses the test() method of the regular expression to check if the params string matches the defined pattern, and returns true if it is a number and false otherwise.

**Example usage:**

var result = isNumber("12345");

console.log(result);

// Output: true

**ipAddress(req)**

This function returns the IP address of the client making the request.

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

*let* ip = '';

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

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

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

        ip = *req*.connection.remoteAddress;

    } else {

        ip = *req*.ip;

    }

    return ip;

}

This code defines a function ipAddress that takes a req parameter, typically representing a request object in a server environment. It extracts the IP address of the client making the request by looking at different places depending on the structure of the request object.

1. If the x-forwarded-for header is present in the request's headers, it extracts the first IP address from it.
2. Otherwise, if the connection object exists in the request and it has a remoteAddress property, it uses that as the IP address.
3. If none of the above conditions are met, it falls back to using req.ip as the IP address.
4. The function then returns the extracted IP address.

**Example usage:**

var ip = ipAddress(req);

console.log(ip);

// Output: "192.168.0.1"

**timeCreate()**

This function returns the current time in milliseconds since the Unix epoch.

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

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

*const* time = d.getTime();

    return time;

}

In the provided code snippet, a function timeCreate is defined without any parameters. Inside the function, a new Date object d is created using the current date and time. The getTime() method is then called on the date object d to get the current time in milliseconds since the Unix epoch. Finally, the current time in milliseconds is returned from the function.

The optimized version of the code snippet replaces the usage of Date with time.Now() in Go, which provides a more idiomatic way to get the current time. The UnixNano() method is used to get the current time in nanoseconds since the Unix epoch, which is of type int64. The function signature specifies that the function returns an int64 value instead of a number.

**Example usage:**

var time = timeCreate();

console.log(time);

// Output: 1618274454321

**Controller Functions**

The code file contains several controller functions that handle different routes and actions:

**loginPage(req, res)**

This function renders the login page.

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

    return *res*.render("account/login.ejs");

}

**registerPage(req, res)**

This function renders the register page.

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

*var* whatsapp1 = process.env.WHATSAPP\_LOCAL\_KEY;

*var* whatsapp2 = process.env.WHATSAPP\_INTERNATIONAL\_KEY;

    return *res*.render("account/register.ejs", {whatsapp1, whatsapp2});

}

The provided code defines an asynchronous function named registerPage with two variables whatsapp1 and whatsapp2 that are assigned values from environment variables WHATSAPP\_LOCAL\_KEY and WHATSAPP\_INTERNATIONAL\_KEY respectively. It then renders an EJS template 'account/register.ejs' and passes an object containing whatsapp1 and whatsapp2 to the template for rendering.

**forgotPage(req, res)**

This function renders the forgot password page.

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

    return *res*.render("account/forgot.ejs");

}

**login(req, res)**

This function handles the login process by validating the username, password, and country code and returning the appropriate response.

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

*let* { username, pwd , countrycode} = *req*.body;

    if (!username || !pwd || !username) {//!isNumber(username)

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

            message: 'ERROR!!!'

        });

    }

    try {

*const* [rows] = await connection.query('SELECT \* FROM users WHERE phone = ? AND password = ? AND dial\_code = ? ', [username, md5(pwd), countrycode]);

        if (rows.length == 1) {

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

*const* { password, money, ip, veri, ip\_address, status, time, ...others } = rows[0];

*const* accessToken = jwt.sign({

                    user: { ...others },

                    timeNow: timeNow

                }, process.env.JWT\_ACCESS\_TOKEN, { expiresIn: "1d" });

                await connection.execute('UPDATE `users` SET `token` = ? WHERE `phone` = ? ', [md5(accessToken), username]);

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

                    message: 'Login Successfully!',

                    status: true,

                    token: accessToken,

                    value: md5(accessToken)

                });

            } else {

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

                    message: 'Account has been locked',

                    status: false

                });

            }

        } else {

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

                message: 'Incorrect Username or Password',

                status: false

            });

        }

    } catch (error) {

        if (error) console.log(error);

    }

}

In the provided code snippet, there is an asynchronous function named login that handles user authentication.

The function destructures the req.body object to extract username, pwd, and countrycode. It then checks if any of these values are missing. If any of them is missing, it returns an error message.

Inside a try-catch block, the code attempts to query a database table to find a user with the provided username, encrypted pwd, and countrycode. If a user is found, it checks the user's status. If the status is active, it generates a JWT token, updates the user's token in the database, and sends a success response with the token. If the status is not active, it returns an account locked message. If no user is found, it returns an incorrect username or password message.

If any errors occur during the database operations, they are caught in the catch block and logged to the console.

**Example usage:**

...

**register(req, res)**

This function handles the registration process by validating the username, password, and invite code and returning the appropriate response.

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

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

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

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

*let* otp2 = randomNumber(100000, 999999);

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

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

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

*let* time = timeCreate();

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

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

            message: 'ERROR!!!',

            status: false

        });

    }

    if (username.length < 9 || username.length > 10 || !isNumber(username)) {

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

            message: 'phone error',

            status: false

        });

    }

    try {

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

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

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

        if (check\_u.length == 1 && check\_u[0].veri == 1) {

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

                message: 'Registered phone number',

                status: false

            });

        } else {

            if (check\_i.length == 1) {

               // if (check\_ip.length <= 3) {

*let* ctv = '';

                    if (check\_i[0].level == 2) {

                        ctv = check\_i[0].phone;

                    } else {

                        ctv = check\_i[0].ctv;

                    }

*const* sql = "INSERT INTO users SET id\_user = ?,phone = ?,name\_user = ?,password = ?, plain\_password = ?, money = ?,code = ?,invite = ?,ctv = ?,veri = ?,otp = ?,ip\_address = ?,status = ?,time = ?,dial\_code = ?";

                    await connection.execute(sql, [id\_user, username, name\_user, md5(pwd), pwd, 0, code, invitecode, ctv, 1, otp2, ip, 1, time, countrycode]);

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

*let* [check\_code] = await connection.query('SELECT \* FROM users WHERE invite = ? ', [invitecode]);

                    if(check\_i.name\_user !=='Admin'){

*let* levels = [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44];

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

                            if (check\_code.length >= levels[i]) {

*let* [mainUser] = await connection.query('SELECT \* FROM users WHERE code = ? ', [invitecode]);

                                if(parseInt(mainUser[0].user\_level) != parseInt(i + 1))

                                {

                                    await connection.execute('UPDATE users SET user\_level = ? WHERE code = ?', [i + 1, invitecode]);

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

                                    await connection.query(sql\_noti, [mainUser[0].id, "Congartualtions..! Your level is updated to Level : " + parseInt(i +1), '0', "Level"]);

                                }

                            } else {

                                break;

                            }

                        }

                    }

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

                    await connection.query(sql\_noti, [check\_i[0].id, "Your invitee "+ username.replace(/[0-9]+5/g,*i=>*"\*\*\*\*\*".slice(0,*i*.length)) +" had joined our platform, referral code " + code +" Congartualtions...!", '0', "Referral"]);

*let* sql4 = 'INSERT INTO turn\_over SET phone = ?, code = ?, invite = ?';

                    await connection.query(sql4, [username, code, invitecode]);

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

                        message: "Registered successfully",

                        status: true

                    });

                // } else {

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

                //         message: 'Registered IP address',

                //         status: false

                //     });

                // }

            } else {

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

                    message: 'Referrer code does not exist',

                    status: false

                });

            }

        }

    } catch (error) {

        if (error) console.log(error);

    }

}

The given code is a Node.js route handler function implemented using async/await features. Here's a breakdown of the code:

1. Await input data from the request body like username, password, invite code, and country code.
2. Perform validations on the input data to ensure all required fields are provided and meet certain criteria.
3. Query the database to check if the user with the provided phone number or invite code already exists.
4. If the checks pass, insert the user details into the 'users' table and perform related operations like updating user levels and sending notifications.
5. Return appropriate JSON responses for different scenarios such as successful registration, existing phone number, non-existent referrer code, etc.

**Example usage:**

...

**forgotPage(req, res)**

This function renders the forgot password page.

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

    return *res*.render("account/forgot.ejs");

}

**verifyCode(req, res)**

This function handles the verification process for a given phone number by sending an OTP code via SMS.

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

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

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

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

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

    if (phone.length < 9 || phone.length > 10 || !isNumber(phone)) {

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

            message: 'phone error',

            status: false

        });

    }

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

    if (rows.length == 0) {

        await request(`http://47.243.168.18:9090/sms/batch/v2?appkey=NFJKdK&appsecret=brwkTw&phone=84${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("INSERT INTO users SET phone = ?, otp = ?, veri = 0, time\_otp = ? ", [phone, otp, timeEnd]);

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

                    message: 'Submitted successfully',

                    status: true,

                    timeStamp: timeNow,

                    timeEnd: timeEnd,

                });

            }

        });

    } else {

*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${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, 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,

            });

        }

    }

}

This code is an asynchronous function named verifyCode that handles a verification process for a user. Here's a breakdown of the code:

1. phone, now, timeEnd, and otp are initialized to specific values.
2. A check is performed on the phone number length and whether it is a numeric value. If the conditions are not met, an error response is sent.
3. A database query is made to check if the user with the provided phone number exists.
4. If the user does not exist, a request is made to send an OTP via SMS. If successful, the user is inserted into the database with the OTP and relevant details.
5. If the user exists and the OTP has expired, a new OTP is sent via SMS, and the user's OTP and time details are updated in the database.
6. If the user exists and the OTP is still valid, an error response is sent indicating to send SMS regularly.

**Example usage:**

...

**verifyCodePass(req, res)**

This function handles the verification process for a given phone number during password reset by sending an OTP code via SMS.

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

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

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

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

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

    if (phone.length < 9 || phone.length > 10 || !isNumber(phone)) {

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

            message: 'phone error',

            status: false

        });

    }

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

    if (rows.length == 0) {

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

            message: 'Account does not exist',

            status: false,

            timeStamp: timeNow,

        });

    } else {

*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${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, 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 code defines an asynchronous function named verifyCodePass that takes req and res as parameters.

1. It extracts the phone from req.body and calculates the current time and the time after two minutes, and generates a random OTP.
2. It then checks if the phone number is valid. If not valid, it returns an error response.
3. It queries a database to check if the user account exists. If not found, it returns an error response.
4. If the user account exists and the OTP expiration time has not passed, it sends an SMS with the OTP code to the user's phone number and updates the database with the new OTP and expiration time.
5. If the SMS is sent successfully, it returns a success response; otherwise, it returns an error response.

**Example usage:**

...

**forGotPassword(req, res)**

This function handles the password reset process by validating the username, OTP code, and new password.

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

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

*let* otp = *req*.body.otp;

*let* pwd = *req*.body.pwd;

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

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

*let* otp2 = randomNumber(100000, 999999);

    if (username.length < 9 || username.length > 10 || !isNumber(username)) {

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

            message: 'phone error',

            status: false

        });

    }

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

    if (rows.length == 0) {

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

            message: 'Account does not exist',

            status: false,

            timeStamp: timeNow,

        });

    } else {

*let* user = rows[0];

        if (user.time\_otp - now > 0) {

            if (user.otp == otp) {

                await connection.execute("UPDATE users SET password = ?, otp = ?, time\_otp = ? WHERE phone = ? ", [md5(pwd), otp2, timeEnd, username]);

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

                await connection.query(sql\_noti, [user.id, "Password Changed Successfully", '0', "Password"]);

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

                    message: 'Change password successfully',

                    status: true,

                    timeStamp: timeNow,

                    timeEnd: timeEnd,

                });

            } else {

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

                    message: 'OTP code is incorrect',

                    status: false,

                    timeStamp: timeNow,

                });

            }

        } else {

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

                message: 'OTP code has expired',

                status: false,

                timeStamp: timeNow,

            });

        }

    }

}

The given code defines an asynchronous function forGotPassword that handles the process of resetting a user's password. Here's an explanation of what the code does:

1. username, otp, and pwd variables are extracted from the request body.
2. A random six-digit otp2 is generated using the randomNumber function.
3. Validation checks are performed on the username to ensure it meets certain criteria. If not valid, an error response is sent back.
4. A database query is executed to fetch user details based on the phone number (assuming connection is an active DB connection).
5. If the user exists and the OTP has not expired, a comparison is made between the provided OTP and the saved OTP.
6. If the OTP matches, the user's password is updated in the database, a notification is added, and a success response is sent.
7. If the OTP is incorrect, an error response is sent.
8. If the OTP has expired, an error response indicating the expiry is sent.

**Example usage:**

...

**keFuMenu(req, res)**

This function renders the keFu menu page with the appropriate Telegram settings based on the user's level or the admin's settings.

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

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

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

*let* telegram = '';

    if (users.length == 0) {

*let* [settings] = await connection.query('SELECT `telegram`, `cskh` FROM admin');

        telegram = settings[0].telegram;

    } else {

        if (users[0].level != 0) {

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

        } else {

*var* [check] = await connection.query('SELECT `telegram` FROM point\_list WHERE phone = ?', [users[0].ctv]);

            if (check.length == 0) {

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

            } else {

*var* [settings] = await connection.query('SELECT `telegram` FROM point\_list WHERE phone = ?', [users[0].ctv]);

            }

        }

        telegram = settings[0].telegram;

    }

    return *res*.render("keFuMenu.ejs", {telegram});

}

In this code snippet, a function named keFuMenu is defined as an asynchronous arrow function that takes req and res as parameters.

Inside the function, the code fetches the auth token from req.cookies.auth and uses it to query the database to retrieve user information including the level and ctv.

Depending on the retrieved data, the function then determines the telegram value either from the admin table or the point\_list table. Finally, it renders an ejs template named keFuMenu.ejs passing the telegram value to it.

**Example usage:**

...

**updateAvatarAPI(req, res)**

This function handles the update of the user's avatar by accepting the avatar URL in the request body and updating the corresponding database record.

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

    try {

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

*let* avatar = *req*.body.avatar;

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

        "SELECT \* FROM users WHERE token = ?",

        [auth],

      );

      if (rows.length == 0) {

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

          message: "Account does not exist",

          status: false,

        });

      }

      await connection.execute("UPDATE users SET avatar = ? WHERE token = ?", [

        avatar,

        auth,

      ]);

      return *res*.status(200).json({

        message: "Change avatar successfully",

        status: true,

      });

    } catch (error) {

      return *res*

        .status(500)

        .json({ message: "Internal Server Error", status: false });

    }

  };

This code defines an asynchronous function named updateAvatarAPI that handles the update of a user's avatar. Here is a breakdown of the code:

1. auth is assigned the value of req.cookies.auth, and avatar is assigned the value of req.body.avatar.
2. A database query is executed using the connection.query method to select rows from the users table where the token matches auth. If no rows are returned, a response with an error message is sent back.
3. If rows are found, an update query is executed to set the avatar for the user identified by the token.
4. If all operations are successful, a success response is returned with a message indicating the avatar change was successful.
5. If any errors occur during the process, a 500 status error response is sent back.