# **Documentation - trxWingoController.js**

The trxWingoController.js file is responsible for handling the logic and functionality related to the TRX Wingo game in the software project. It contains functions for rendering game pages, processing bets, retrieving game results, and managing user data and commissions.

### trxWingoPage

This function renders the TRX Wingo game page.

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

  return *res*.render("bet/trx\_wingo/win.ejs");

};

### betTrxWingo

This function processes a bet in the TRX Wingo game. It retrieves the bet details, checks if the bet is valid, calculates the win amount, and updates the user's balance and bet status accordingly.

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

  try {

*let* { typeid, join, x, money } = *req*.body;

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

    if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

      return *res*.status(200).json({

        message: "Error!",

        status: true,

      });

    }

*let* gameJoin = "";

    if (typeid == 1) gameJoin = "trx\_wingo";

    if (typeid == 3) gameJoin = "trx\_wingo3";

    if (typeid == 5) gameJoin = "trx\_wingo5";

    if (typeid == 10) gameJoin = "trx\_wingo10";

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

      `SELECT period FROM trx\_wingo\_game WHERE status = 0 AND game = ? ORDER BY id DESC LIMIT 1 `,

      [gameJoin],

    );

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

      "SELECT `phone`, `code`, `invite`, `level`, `money` FROM users WHERE token = ? AND veri = 1  LIMIT 1 ",

      [auth],

    );

    if (!trxWingoNow[0] || !user[0] || !isNumber(x) || !isNumber(money)) {

      return *res*.status(200).json({

        message: "Error!",

        status: true,

      });

    }

*let* userInfo = user[0];

*let* period = trxWingoNow[0].period;

*let* fee = x \* money \* 0.02;

*let* total = x \* money - fee;

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

*let* check = userInfo.money - total;

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

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

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

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

*let* id\_product =

      years + months + days + *Math*.floor(*Math*.random() \* 1000000000000000);

*let* formatTime = timerJoin();

*let* color = "";

    if (join == "l") {

      color = "big";

    } else if (join == "n") {

      color = "small";

    } else if (join == "t") {

      color = "violet";

    } else if (join == "d") {

      color = "red";

    } else if (join == "x") {

      color = "green";

    } else if (join == "0") {

      color = "red-violet";

    } else if (join == "5") {

      color = "green-violet";

    } else if (join % 2 == 0) {

      color = "red";

    } else if (join % 2 != 0) {

      color = "green";

    }

*let* checkJoin = "";

    if ((!isNumber(join) && join == "l") || join == "n") {

      checkJoin = `

        <div data-v-a9660e98="" class="van-image" style="width: 30px; height: 30px;">

            <img src="/images/${join == "n" ? "small" : "big"}.png" class="van-image\_\_img">

        </div>

        `;

    } else {

      checkJoin = `

        <span data-v-a9660e98="">${isNumber(join) ? join : ""}</span>

        `;

    }

*let* result = `

    <div data-v-a9660e98="" issuenumber="${period}" addtime="${formatTime}" rowid="1" class="hb">

        <div data-v-a9660e98="" class="item c-row">

            <div data-v-a9660e98="" class="result">

                <div data-v-a9660e98="" class="select select-${color}">

                    ${checkJoin}

                </div>

            </div>

            <div data-v-a9660e98="" class="c-row c-row-between info">

                <div data-v-a9660e98="">

                    <div data-v-a9660e98="" class="issueName">

                        ${period}

                    </div>

                    <div data-v-a9660e98="" class="tiem">${formatTime}</div>

                </div>

            </div>

        </div>

        <!---->

    </div>

    `;

*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(date.getTime());

    if (check >= 0) {

*const* sql = `INSERT INTO trx\_wingo\_bets SET

            id\_product = ?,

            phone = ?,

            code = ?,

            invite = ?,

            stage = ?,

            level = ?,

            money = ?,

            amount = ?,

            fee = ?,

            get = ?,

            game = ?,

            bet = ?,

            status = ?,

            today = ?,

            time = ?`;

      await connection.query(sql, [

        id\_product,

        userInfo.phone,

        userInfo.code,

        userInfo.invite,

        period,

        userInfo.level,

        total,

        x,

        fee,

        0,

        gameJoin,

        join,

        0,

        checkTime,

        timeNow,

      ]);

      await connection.query(

        "UPDATE `users` SET `money` = `money` - ? WHERE `token` = ? ",

        [money \* x, auth],

      );

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

        "SELECT `money`, `level` FROM users WHERE token = ? AND veri = 1  LIMIT 1 ",

        [auth],

      );

      await rosesPlus(auth, money \* x);

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

      // let level0 = level[0];

      // const sql2 = `INSERT INTO roses SET

      // phone = ?,

      // code = ?,

      // invite = ?,

      // f1 = ?,

      // f2 = ?,

      // f3 = ?,

      // f4 = ?,

      // time = ?`;

      // let total\_m = money \* x;

      // let f1 = (total\_m / 100) \* level0.f1;

      // let f2 = (total\_m / 100) \* level0.f2;

      // let f3 = (total\_m / 100) \* level0.f3;

      // let f4 = (total\_m / 100) \* level0.f4;

      // await connection.query(sql2, [userInfo.phone, userInfo.code, userInfo.invite, f1, f2, f3, f4, timeNow]);

      // console.log(level);

      return *res*.status(200).json({

        message: "Successful bet",

        status: true,

        data: result,

        change: users[0].level,

        money: users[0].money,

      });

    } else {

      return *res*.status(200).json({

        message: "The amount is not enough",

        status: false,

      });

    }

  } catch (error) {

    return *res*.status(200).json({

      message: "Error!",

      status: false,

    });

  }

};

The provided code is an asynchronous function named betTrxWingo that handles a betting transaction. Here's a breakdown of the code:

1. The function expects req (request) and res (response) objects as parameters.
2. Destructures typeid, join, x, and money from the request body and auth from request cookies.
3. Validates typeid and sets gameJoin based on the value of typeid.
4. Queries the database for game and user information.
5. Performs checks on retrieved data and calculates fees and total amounts.
6. Generates an id\_product and formatted time strings using helper functions.
7. Determines color based on the join value.
8. Generates HTML content based on conditions and data.
9. Further logic for inserting bets into the database and updating user information based on the outcome of the checks.

### listOrderOld

This function retrieves a list of previous orders in the TRX Wingo game. It takes the game type, page number, and number of items per page as inputs and returns the corresponding list of orders.

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

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

  if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

    return *res*.status(200).json({

      message: "Error!",

      status: true,

    });

  }

  if (pageno < 0 || pageto < 0) {

    return *res*.status(200).json({

      code: 0,

      msg: "No more data",

      data: {

        gameslist: [],

      },

      page: 1,

      status: false,

    });

  }

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

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

    "SELECT `phone`, `code`, `invite`, `level`, `money` FROM users WHERE token = ? AND veri = 1  LIMIT 1 ",

    [auth],

  );

*let* game = "";

  if (typeid == 1) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_1;

  if (typeid == 3) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_3;

  if (typeid == 5) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_5;

  if (typeid == 10) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_10;

*const* [trx\_wingo] = await connection.query(

    "SELECT \* FROM trx\_wingo\_game WHERE status != 0 AND game = ? ORDER BY id DESC LIMIT ?, ?",

    [game, *Number*(pageno), *Number*(pageto)],

  );

*const* [trx\_wingoAll] = await connection.query(

    "SELECT COUNT(\*) as game\_length FROM trx\_wingo\_game WHERE status != 0 AND game = ?",

    [game],

  );

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

    "SELECT period FROM trx\_wingo\_game WHERE status = 0 AND game = ? ORDER BY id DESC LIMIT 1",

    [game],

  );

  if (!trx\_wingo[0]) {

    return *res*.status(200).json({

      code: 0,

      msg: "No more data",

      data: {

        gameslist: [],

      },

      page: 1,

      status: false,

    });

  }

  if (!pageno || !pageto || !user[0] || !trx\_wingo[0] || !period[0]) {

    return *res*.status(200).json({

      message: "Error!",

      status: true,

    });

  }

*let* page = Math.ceil(trx\_wingoAll[0].game\_length / 10);

  return *res*.status(200).json({

    code: 0,

    msg: "Receive success",

    data: {

      gameslist: trx\_wingo,

    },

    period: period[0].period,

    page: page,

    status: true,

  });

};

This code defines an asynchronous function listOrderOld that receives req and res objects. It extracts typeid, pageno, and pageto from the request body.

1. The function checks if typeid is not one of the specified values (1, 3, 5, 10), it returns an error response. If pageno or pageto is less than zero, it returns a response indicating no more data is available.
2. It then queries the database to fetch user data, selects the appropriate game type based on typeid, and retrieves game data from the database using the selected game type, pagination limits.
3. The function calculates the total number of games for pagination and prepares the final response with game data, period information, pagination details, and a success status.
4. If certain required data is missing or if game data is not found, it returns an error response with the appropriate message.

### Stat\_listOrderOld

This function retrieves a list of previous orders' results in the TRX Wingo game. It takes the game type, page number, and number of items per page as inputs and returns the corresponding list of results.

*const* Stat\_listOrderOld = *async* (*req*, *res*) *=>* {

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

  if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

    return *res*.status(200).json({

      message: "Error!",

      status: true,

    });

  }

*let* game = "";

  if (typeid == 1) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_1;

  if (typeid == 3) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_3;

  if (typeid == 5) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_5;

  if (typeid == 10) game = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_10;

*const* [trx\_wingo] = await connection.query(

    "SELECT result FROM trx\_wingo\_game WHERE status != 0 AND game = ? ORDER BY id DESC LIMIT ?, ?",

    [game, *Number*(pageno), *Number*(pageto)],

  );

  if (!trx\_wingo[0]) {

    return *res*.status(200).json({

      code: 0,

      msg: "No more data",

      data: {

        gameslist: [],

      },

      page: 1,

      status: false,

    });

  }

  if (!pageno || !pageto || !trx\_wingo[0]) {

    return *res*.status(200).json({

      message: "Error!",

      status: true,

    });

  }

  return *res*.status(200).json({

    code: 0,

    msg: "Receive success",

    data: {

      gameslist: trx\_wingo,

    },

    status: true,

  });

};

The provided code is an async function in JavaScript that fetches and handles data based on some conditions.

1. It receives three parameters from the request body: typeid, pageno, pageto. It then checks if the typeid is not equal to 1, 3, 5, or 10 (specific values), it returns an error response.
2. Based on the typeid value, it assigns a corresponding game value from a map (TRX\_WINGO\_GAME\_TYPE\_MAP).
3. It makes a database query using the game value, pageno, and pageto parameters, and then processes the result.
4. If the result is empty or if any of the parameters pageno, pageto, or result is falsy, it returns an error response.
5. Otherwise, it returns a success response with the obtained data.

### GetMyEmerdList

This function retrieves a list of the user's previous orders in the TRX Wingo game. It takes the game type, page number, and number of items per page as inputs and returns the corresponding list of orders.

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

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

  // if (!pageno || !pageto) {

  //     pageno = 0;

  //     pageto = 10;

  // }

  if (typeid != 1 && typeid != 3 && typeid != 5 && typeid != 10) {

    return *res*.status(200).json({

      message: "Error!",

      status: true,

    });

  }

  if (pageno < 0 || pageto < 0) {

    return *res*.status(200).json({

      code: 0,

      msg: "No more data",

      data: {

        gameslist: [],

      },

      page: 1,

      status: false,

    });

  }

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

*let* game = "";

  if (typeid == 1) game = "trx\_wingo";

  if (typeid == 3) game = "trx\_wingo3";

  if (typeid == 5) game = "trx\_wingo5";

  if (typeid == 10) game = "trx\_wingo10";

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

    "SELECT `phone`, `code`, `invite`, `level`, `money` FROM users WHERE token = ? AND veri = 1 LIMIT 1",

    [auth],

  );

*const* [trx\_wingo\_bets] = await connection.query(

    "SELECT \* FROM trx\_wingo\_bets WHERE phone = ? AND game = ? ORDER BY id DESC LIMIT ?, ?",

    [user[0].phone, game, *Number*(pageno), *Number*(pageto)],

  );

*const* [trx\_wingo\_betsAll] = await connection.query(

    "SELECT COUNT(\*) as bet\_length FROM trx\_wingo\_bets WHERE phone = ? AND game = ? ORDER BY id DESC",

    [user[0].phone, game],

  );

  if (!trx\_wingo\_bets[0]) {

    return *res*.status(200).json({

      code: 0,

      msg: "No more data",

      data: {

        gameslist: [],

      },

      page: 1,

      status: false,

    });

  }

  if (!pageno || !pageto || !user[0] || !trx\_wingo\_bets[0]) {

    return *res*.status(200).json({

      message: "Error!",

      status: true,

    });

  }

*let* page = Math.ceil(trx\_wingo\_betsAll[0].bet\_length / 10);

*let* datas = trx\_wingo\_bets.map((*data*) *=>* {

*let* { id, phone, code, invite, level, game, ...others } = *data*;

    return others;

  });

  return *res*.status(200).json({

    code: 0,

    msg: "Receive success",

    data: {

      gameslist: datas,

    },

    page: page,

    status: true,

  });

};

The provided code defines an asynchronous function named GetMyEmerdList that expects req and res parameters.

1. It then extracts typeid, pageno, and pageto from the request body. It checks for valid typeid values and negative pageno and pageto values.
2. Based on the typeid, it assigns a corresponding game value. It then queries the database to fetch user information, bets related to a specific game, and total bet length for a user in that game.
3. It handles different scenarios for missing data or errors. It calculates the number of pages based on the total bet length. It extracts relevant data from bet records and returns a JSON response with the bet data and pagination details.

### handlingTrxWingo1P

This function handles the result distribution for the TRX Wingo game with a game type of 1. It retrieves the pending game result, updates the bet results, calculates the win amounts, and updates the user balances accordingly.

*const* handlingTrxWingo1P = *async* (*typeid*) *=>* {

  try {

*let* game = "";

    if (*typeid* == 1) game = "trx\_wingo";

    if (*typeid* == 3) game = "trx\_wingo3";

    if (*typeid* == 5) game = "trx\_wingo5";

    if (*typeid* == 10) game = "trx\_wingo10";

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

      "SELECT \* FROM trx\_wingo\_game WHERE status = 1 AND release\_status = 1 AND game = ? ORDER BY id DESC LIMIT 1",

      [game],

    );

    if (trxWingoNow.length === 0) {

      return;

    }

    await connection.query(

      "UPDATE trx\_wingo\_bets SET result = ? WHERE status = 0 AND game = ?",

      [trxWingoNow[0].result, game],

    );

*let* result = *Number*(trxWingoNow[0].result);

    await batchUpdateBetStatus(result, game);

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

      "SELECT \* FROM trx\_wingo\_bets WHERE status = 0 AND game = ?",

      [game],

    );

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

*let* orders = order[i];

*let* id = orders.id;

*let* result = orders.result;

*let* bet = orders.bet;

*let* total = orders.money;

*let* phone = orders.phone;

*let* winAmount = calculateWinAmount(bet, result, total);

      // x - green

      // t - Violet

      // d - red

      // Sirf 1-4 aur 6-9 tk hi \*9 aana chahiye

      // Aur 0 aur 5 pe \*4.5

      // Aur red aur green pe \*2

      // 1,2,3,4,6,7,8,9

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

        "SELECT `money` FROM `users` WHERE `phone` = ?",

        [phone],

      );

*let* totals = parseFloat(users[0].money) + parseFloat(winAmount);

      await connection.query(

        "UPDATE `trx\_wingo\_bets` SET `get` = ?, `status` = 1 WHERE `id` = ?",

        [parseFloat(winAmount), id],

      );

      await connection.query(

        "UPDATE `users` SET `money` = ? WHERE `phone` = ?",

        [totals, phone],

      );

    }

    await connection.query(

      "UPDATE trx\_wingo\_game SET release\_status = 2 WHERE period = ? AND game = ?",

      [trxWingoNow[0].period, game],

    );

  } catch (error) {

    console.log(error);

  }

};

The provided code is an asynchronous function in JavaScript that handles transactions related to a specific game based on the 'typeid' parameter. Here is a breakdown of the code:

1. The function receives a 'typeid' parameter that is used to determine the game.
2. It queries the database to retrieve the latest game data based on the 'game' determined by 'typeid'.
3. If no game data is found, the function exits.
4. It updates the 'result' field in 'trx\_wingo\_bets' table based on the retrieved game data.
5. It then calculates the win amount for each user's bet based on specific conditions and updates their balances accordingly.
6. Finally, it updates the release status of the game in the database for the specific period and game.
7. If any errors occur during the process, they are caught and logged to the console.

### addTrxWingo

This function adds a new TRX Wingo game to the database. It generates a new game period, retrieves the current timestamp, and inserts the game details into the database.

*const* addTrxWingo = *async* (*game*) *=>* {

  try {

*let* join = "";

    if (*game* == 1) join = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_1;

    if (*game* == 3) join = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_3;

    if (*game* == 5) join = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_5;

    if (*game* == 10) join = TRX\_WINGO\_GAME\_TYPE\_MAP.MIN\_10;

    // console.log(join)

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

      "SELECT period FROM trx\_wingo\_game WHERE status = 0 AND game = ? ORDER BY id DESC LIMIT 1",

      [join],

    );

*const* isPendingGame = trxWingoNow.length;

*const* PendingGamePeriod = trxWingoNow?.[0]?.period

      ? parseInt(trxWingoNow?.[0]?.period)

      : 0;

    if (isPendingGame) {

      // console.log("TRX WINGO GAME PENDING GAME INSERTIONS Start")

*const* isAdminManipulatedResult = false;

      if (isAdminManipulatedResult) {

      } else {

*let* response = await axios({

          method: "GET",

          url: "https://apilist.tronscanapi.com/api/block?sort=-balance&start=0&limit=20&producer=&number=&start\_timestamp=&end\_timestamp=",

          headers: {

            "TRON-PRO-API-KEY": *process*.*env*.TRON\_API\_KEY,

          },

        });

       // console.log(response.data.data);

*const* NextBlock = response.data.data

          .map((item) *=>* {

            return {

              id: item.number,

              hash: item.hash,

              blockTime: item.timestamp,

              timeSS: moment(item.timestamp).format("ss"),

            };

          })

          .find((item) *=>* item.timeSS === *process*.*env*.TRX\_WINGO\_GAME\_TIME\_SS);

        if (NextBlock === undefined) {

          throw new Error("NextBlock is undefined");

        }

*const* BlockId = NextBlock.id;

*const* BlockTime = NextBlock.blockTime;

*const* Hash = NextBlock.hash;

*let* Result = generateResultByHash(Hash);

        // console.log({

        //    BlockId,

        //    BlockTime: moment(BlockTime).format("HH:mm:ss"),

        //    Hash,

        //    Result,

        // })

        await connection.query(

          `

               UPDATE trx\_wingo\_game

               SET result = ?, status = ?, block\_id = ?, block\_time = ?, hash = ?, release\_status = 1

               WHERE period = ? AND game = ?

               `,

          [

            Result,

            TRX\_WINGO\_GAME\_STATUS\_MAP.COMPLETED,

            BlockId,

            BlockTime,

            Hash,

            PendingGamePeriod,

            join,

          ],

        );

        // console.log("TRX WINGO GAME PENDING GAME INSERTIONS Successfully")

      }

    }

*let* gameRepresentationId = GameRepresentationIds.TRXWINGO[*game*];

*let* NewGamePeriod = generatePeriod(gameRepresentationId);

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

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

      "SELECT period FROM trx\_wingo\_game WHERE period = ? AND game = ?",

      [NewGamePeriod, join],

    );

    if (trxWinGoTest.length > 0) {

      return;

    }

    await connection.query(

      "INSERT INTO trx\_wingo\_game SET period = ?, result = 0, game = ?, status = 0, block\_id = 0, block\_time = '', hash = '', time = ?",

      [NewGamePeriod, join, timeNow],

    );

    // console.log("TRX WINGO GAME INSERT SUCCESSFULLY")

  } catch (error) {

    if (error?.response?.status === 403) {

      console.log("API Quota Exceeded for 30 seconds");

      console.log(error.response.data.*Error*);

    } else console.log(error);

  }

};

The provided code defines an asynchronous function addTrxWingo that takes a game parameter. The function performs various operations related to a game process using Tron blockchain data. Here's a breakdown of the code:

1. Checks the value of game and assigns a corresponding value to the join variable based on predefined mappings.
2. Queries the database to retrieve the latest period for a specific game.
3. Processes the pending game if it exists by fetching blockchain data, determining the next block, and updating the game status accordingly.
4. Generates a new game period and inserts a new game record into the database if no existing game matches the period and game type.
5. The code also includes error handling to log specific messages based on different error scenarios, such as API quota exceeded or other general errors.

### trxWingoPage3, trxWingoPage5, trxWingoPage10

These functions are similar to trxWingoPage and render the TRX Wingo game pages with different game types.

*const* trxWingoPage10 = *async* (*req*, *res*) *=>* {

  return *res*.render("bet/trx\_wingo/win10.ejs");

};

*const* trxWingoPage3 = *async* (*req*, *res*) *=>* {

  return *res*.render("bet/trx\_wingo/win3.ejs");

};

*const* trxWingoPage5 = *async* (*req*, *res*) *=>* {

  return *res*.render("bet/trx\_wingo/win5.ejs");

};

### trxWingoBlockPage

This function renders the TRX Wingo block page, used for the admin panel or moderation purposes.

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

  return *res*.render("bet/trx\_wingo/trx\_block.ejs");

};

### distributeCommission

This function distributes commissions to users based on their bets in the TRX Wingo game. It retrieves the bets made during the previous day, calculates the commissions, and updates the user balances accordingly.

*const* distributeCommission = *async* () *=>* {

  try {

*const* { startOfYesterdayTimestamp, endOfYesterdayTimestamp } =

      yesterdayTime();

*const* [levelResult] = await connection.query("SELECT f1 FROM level");

*const* levels = levelResult.map((*row*) *=>* *row*.f1 / 100);

    // const [bets] = await connection.query('SELECT phone, SUM(money + fee) AS total\_money FROM minutes\_1 WHERE time > ? AND time <= ? GROUP BY phone', [startOfDay, endTime]);

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

      `

      SELECT phone, SUM(total\_money) AS total\_money

      FROM (

        SELECT phone, SUM(money + fee) AS total\_money

        FROM trx\_wingo\_bets

        WHERE time > ? AND time <= ?

        GROUP BY phone

        UNION ALL

        SELECT phone, SUM(money + fee) AS total\_money

        FROM trx\_wingo\_bets

        WHERE time > ? AND time <= ?

        GROUP BY phone

      ) AS combined

      GROUP BY phone

      `,

      [

        startOfYesterdayTimestamp,

        endOfYesterdayTimestamp,

        startOfYesterdayTimestamp,

        endOfYesterdayTimestamp,

      ],

    );

*const* promises = bets.map((*bet*) *=>*

      rosesPlus1(*bet*.phone, *bet*.total\_money, levels, endOfYesterdayTimestamp),

    );

*const* response = await *Promise*.all(promises);

    return {

      success: true,

      message: "Commissions distributed successfully.",

    };

  } catch (error) {

    console.error(error);

    return { success: false, message: error.message };

  }

};

The provided code defines an asynchronous function named distributeCommission that will distribute commissions based on certain criteria. Here's a breakdown of the code:

1. The function uses try...catch to handle any errors that might occur during execution.
2. It destructures the startOfYesterdayTimestamp and endOfYesterdayTimestamp values from the result of calling a function yesterdayTime().
3. It queries the database to fetch f1 from the level table and calculates the levels based on the fetched data.
4. It executes a complex SQL query to calculate total money for each phone number within a specific time range and combines the results before grouping them by phone number.
5. It then maps over the fetched bets and calls rosesPlus1 function for each bet, passing required parameters.
6. Finally, it awaits all promises to resolve and returns a success message if everything was executed successfully, or an error message if an error occurred during the process.

### isNumber(params)

This function checks if the given params is a number using a regular expression pattern. It returns true if params is a number, otherwise it returns false.

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

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

  return pattern.test(*params*);

};

The provided code defines a function isNumber that takes a parameter params and checks if it is a number.

1. Inside the function, a regular expression pattern ^[0-9]\*\d$ is used. This pattern allows for zero or more occurrences of digits (0-9) followed by exactly one digit at the end of the string.
2. The function uses the test method of the regular expression object to test if the input parameter params matches the defined pattern. It then returns the result of this test, which is a boolean indicating whether the params is a number according to the pattern.

### formateT(params)

This function formats the given params by adding a leading zero if it is less than 10. It returns the formatted result.

*function* formateT(*params*) {

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

  return result;

}

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

Within the function, it checks if the params value is less than 10. If true, it appends a '0' in front of the params value converted to a string using strconv.Itoa(). If false, it simply converts the params value to a string.

Finally, the function returns the resulting string based on the conditions.

### timerJoin(params, addHours)

This function joins timer components (years, months, days, hours, minutes, seconds, and AM/PM) to generate a formatted string representing the date and time. It takes an optional params parameter as a timestamp and an optional addHours parameter to add a specified number of hours to the time. If no params is provided, the current date and time are used. It returns the formatted date and time string.

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

*let* date = "";

  if (*params*) {

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

  } else {

    date = new *Date*();

  }

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

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

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

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

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

  hours = hours === 0 ? 12 : hours;

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

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

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

  return (

    years +

    "-" +

    months +

    "-" +

    days +

    " " +

    hours +

    ":" +

    minutes +

    ":" +

    seconds +

    " " +

    ampm

  );

}

The provided function timerJoin takes two parameters: params and addHours, with default values of an empty string and 0 respectively.

It first checks if params is truthy. If it is, it converts params to a number and creates a new Date object using that value. Otherwise, it uses the current date and time to create a Date object.

It then adds the specified number of addHours to the hours of the Date object.

Next, it extracts the year, month, day, hour (in 12-hour format with AM/PM), minutes, and seconds from the Date object after formatting them using the formateT function (not provided in the code).

Finally, it constructs a string combining these formatted date and time components and returns it.

### minutesPassedSince(time)

This function calculates the number of minutes passed since the specified time. The time parameter should be in a valid format that can be parsed by the moment library. It returns the number of minutes.

*function* minutesPassedSince(*time*) {

*const* inputTime = moment(*time*);

*const* minutesDiff = moment().diff(inputTime, "minutes");

  return minutesDiff;

}

### Key Points:

1. time**Parameter**:

* This is the input passed to the function. It represents a timestamp or date that we want to calculate the difference from.

1. moment(time):

* The moment library is used here to parse the input time into a moment object. This makes it easier to perform operations like calculating differences between dates/times.

1. moment().diff(inputTime, "minutes"):

* moment() represents the current date and time.
* .diff(inputTime, "minutes") calculates the difference between the current time and the inputTime. The "minutes" argument specifies that the result should be in minutes.

1. **Return Value**:

* The function returns the number of minutes that have passed since the time provided as input.

### 

### sleep(ms)

This function introduces a delay of ms milliseconds. It returns a promise that resolves after the specified delay.

*function* sleep(*ms*) {

  return new *Promise*((resolve) *=>* setTimeout(resolve, *ms*));

}

### Key Points:

1. **Purpose**:

* The sleep function is used to pause code execution for a specified duration (in milliseconds). However, it does not block the entire program; instead, it allows asynchronous execution.

1. **Parameters**:

* ms: This represents the duration (in milliseconds) for which the program should "pause."

1. setTimeout(resolve, ms):

* The setTimeout function is a built-in JavaScript function that executes a callback after a specified delay (in this case, ms milliseconds).
* Here, the callback is resolve, which completes the promise after the delay.

1. **Return Value**:

* The function returns a **Promise**.
* The promise resolves after ms milliseconds, allowing asynchronous code to wait for the specified time before proceeding.

### generateResultByHash(hash)

This function generates a result based on the given hash. It iterates through the characters in the hash from right to left and returns the first numeric character encountered. If no numeric character is found, it returns an empty string. Note that the isNumber function within this function is different from the isNumber function defined earlier in the code.

*const* generateResultByHash = (*hash*) *=>* {

*const* hashItemList = *hash*.split("");

*let* Result = "";

  for (*let* index = 0; index < hashItemList.length; index++) {

*const* hashItem = hashItemList[hashItemList.length - 1 - index];

*const* NUMBER\_LIST = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"];

*const* isNumber = NUMBER\_LIST.includes(hashItem);

    if (isNumber) {

      Result = hashItem;

      break;

    }

  }

  return Result;

};

The given code defines a function generateResultByHash that takes a hash as input and returns the last digit from the input string hash.

1. It splits the hash string into an array using split() method, then iterates over the array in reverse order. Inside the loop, it checks if the current character is a number by comparing it with a predefined NUMBER\_LIST.
2. If a number is found, it assigns that number to the Result variable and breaks out of the loop.
3. Finally, it returns the last number found in the hash string. If no numbers are found, an empty string is returned.

### getNthMinuteSinceDayStart()

This function calculates the number of minutes passed since the start of the day. It uses the moment library to get the current time and the start of the day, and then calculates the difference in minutes. It returns the number of minutes passed.

*function* getNthMinuteSinceDayStart() {

*const* now = moment();

*const* startOfDay = moment().startOf("day");

*const* diff = now.diff(startOfDay, "minutes");

  return diff;

}

### Key Points:

1. **now**:
   * moment() generates the current date and time as a moment object.
2. **startOfDay**:
   * moment().startOf("day") returns a new moment object representing the exact start of the current day (00:00:00). This is helpful for calculating time differences relative to the beginning of the day.
3. **diff**:
   * now.diff(startOfDay, "minutes") calculates the difference in minutes between the current time (now) and midnight (startOfDay). The "minutes" parameter ensures the result is in minutes.
4. **Return Value**:
   * The function returns an integer representing how many minutes have passed since midnight.

### batchUpdateBetStatus(result, game)

This function batch updates the status of bets in the trx\_wingo\_bets table. It takes a result and game as parameters. It retrieves valid bets based on the result using the getValidBets function, and then updates a batch of bets with the specified game, with a status of 0 (not processed), and not included in the valid bets. The batchSize parameter determines the number of bets to update in a single query. This can be adjusted based on data volume and server capacity. The function continues updating batches until no more rows are affected. It uses a database connection (connection) to execute the update queries.

*const* batchUpdateBetStatus = *async* (*result*, *game*) *=>* {

*const* validBets = getValidBets(*result*);

*const* batchSize = 1000; // Adjust this based on your data volume and server capacity

*let* offset = 0;

  while (true) {

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

      `UPDATE trx\_wingo\_bets SET status = 2

       WHERE status = 0 AND game = ? AND bet NOT IN (${validBets.map(() *=>* "?").join(",")})

       LIMIT ${batchSize}`,

      [*game*, ...validBets],

    );

    if (rows.affectedRows === 0) break; // No more rows to update

    offset += batchSize;

  }

};

The provided code defines an asynchronous function batchUpdateBetStatus that takes two parameters result and game. Here's a breakdown of the code:

1. validBets is assigned the result of calling a function getValidBets with the result parameter.
2. batchSize is set to 1000 to determine the number of rows to update in each batch.
3. A while loop is used to update rows in batches until all rows are updated.
4. Inside the loop, an SQL query is executed using connection.execute to update rows in the trx\_wingo\_bets table where status is 0, game matches the parameter, and bet is not in the validBets array. The query updates rows in batches of batchSize.
5. If no rows are affected by the update (indicating no more rows to update), the loop breaks.
6. The offset is incremented by batchSize after each batch update.

**Example:**

await batchUpdateBetStatus("8", "game123");

### getValidBets(result)

This function generates valid bets based on the given result. It converts the result to a number, and then generates an array of base valid bets. The base valid bets include the result itself, "d" if the result is even, "x" if the result is odd, and "t" if the result is 0 or 5. Additional valid bets are added based on the value of the result (e.g., "n" if the result is less than or equal to 4). The function returns the array of valid bets.

*const* getValidBets = (*result*) *=>* {

*result* = *Number*(*result*);

*const* baseValidBets = [*result*.toString()];

  if (*result* % 2 === 0) baseValidBets.push("d");

  else baseValidBets.push("x");

  if (*result* === 0 || *result* === 5) baseValidBets.push("t");

  if (*result* <= 4) baseValidBets.push("n");

  else baseValidBets.push("l");

  return baseValidBets;

};

The provided code defines a function getValidBets that takes a single parameter result. Inside the function:

1. The result parameter is converted to a number using Number(result).
2. An array baseValidBets is initialized with a single element which is the string representation of the result.
3. If the result is an even number (checked by result % 2 === 0), the letter 'd' is pushed to baseValidBets; otherwise, 'x' is pushed.
4. If the result equals 0 or 5, the letter 't' is added to baseValidBets.
5. If the result is less than or equal to 4, the letter 'n' is included in baseValidBets; otherwise, 'l' is added.
6. Finally, the function returns the baseValidBets array.

**Example:**

console.log(getValidBets("7")); // Output: ["7", "x", "n"]

### calculateWinAmount(bet, result, total)

This function calculates the win amount for a given bet, result, and total amount. The function evaluates different conditions based on the value of the bet and the result to determine the win amount. The win amount is returned as a number. Note that the function assumes bet and result are strings representing single characters.

*const* calculateWinAmount = (*bet*, *result*, *total*) *=>* {

*let* winAmount = 0;

  if (*bet* == "l" || *bet* == "n") {

    winAmount = *total* \* 2;

  } else {

    if (*result* == 0 || *result* == 5) {

      if (*bet* == "d" || *bet* == "x") {

        winAmount = *total* \* 1.5;

      } else if (*bet* == "t") {

        winAmount = *total* \* 4.5;

      } else if (*bet* == "0" || *bet* == "5") {

        winAmount = *total* \* 4.5;

      }

    } else {

      if (*result* == 1 && *bet* == "1") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 1 && *bet* == "x") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 2 && *bet* == "2") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 2 && *bet* == "d") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 3 && *bet* == "3") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 3 && *bet* == "x") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 4 && *bet* == "4") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 4 && *bet* == "d") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 6 && *bet* == "6") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 6 && *bet* == "d") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 7 && *bet* == "7") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 7 && *bet* == "x") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 8 && *bet* == "8") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 8 && *bet* == "d") {

          winAmount = *total* \* 2;

        }

      }

      if (*result* == 9 && *bet* == "9") {

        winAmount = *total* \* 9;

      } else {

        if (*result* == 9 && *bet* == "x") {

          winAmount = *total* \* 2;

        }

      }

    }

  }

  return winAmount;

};

The provided code defines a function calculateWinAmount that takes three parameters: bet, result, and total. It calculates the win amount based on these inputs and returns the calculated value.

The function first initializes winAmount to 0. It then checks the values of bet and result to determine the win amount following specific conditions as described in the nested if-else statements.

The conditions check for various combinations of bet and result, assigning the appropriate win amount based on the rules provided. The win amount is calculated by multiplying the total with a multiplier specific to each condition.

Overall, the function efficiently evaluates the win amount based on the input parameters and returns the final result.

**Example:**

console.log(calculateWinAmount("1", "1", 10)); // Output: 90

console.log(calculateWinAmount("x", "1", 10)) // Output: 20

# **rosesPlus**

rosesPlus is an asynchronous function that takes two parameters: auth and money. It retrieves data from the level table and the users table using SQL queries.

This function then performs a series of operations to calculate and update the roses\_f field for certain users in the database based on their user level and total money. It also updates the money field for certain users and inserts data into the roses and turn\_over tables.

*const* rosesPlus = *async* (*auth*, *money*) *=>* {

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

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

    "SELECT `phone`, `code`, `invite`, `user\_level`, `total\_money` FROM users WHERE token = ? AND veri = 1 LIMIT 1 ",

    [*auth*],

  );

*let* userInfo = user[0];

*const* [f1] = await connection.query(

    "SELECT `phone`, `code`, `invite`, `rank`, `user\_level`, `total\_money` FROM users WHERE code = ? AND veri = 1 LIMIT 1 ",

    [userInfo.invite],

  );

  if (userInfo.total\_money >= 100) {

    if (f1.length > 0) {

*let* infoF1 = f1[0];

      for (*let* levelIndex = 1; levelIndex <= 6; levelIndex++) {

*let* rosesF = 0;

        if (infoF1.user\_level >= levelIndex && infoF1.total\_money >= 100) {

          rosesF = (*money* / 100) \* level[levelIndex - 1].f1;

          if (rosesF > 0) {

            await connection.query(

              "UPDATE users SET money = money + ?, roses\_f = roses\_f + ?, roses\_today = roses\_today + ? WHERE phone = ? ",

              [rosesF, rosesF, rosesF, infoF1.phone],

            );

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

*const* sql2 = `INSERT INTO roses SET

                            phone = ?,

                            code = ?,

                            invite = ?,

                            f1 = ?,

                            time = ?`;

            await connection.query(sql2, [

              infoF1.phone,

              infoF1.code,

              infoF1.invite,

              rosesF,

              timeNow,

            ]);

*const* sql3 = `

                            INSERT INTO turn\_over (phone, code, invite, daily\_turn\_over, total\_turn\_over)

                            VALUES (?, ?, ?, ?, ?)

                            ON DUPLICATE KEY UPDATE

                            daily\_turn\_over = daily\_turn\_over + VALUES(daily\_turn\_over),

                            total\_turn\_over = total\_turn\_over + VALUES(total\_turn\_over)

                            `;

            await connection.query(sql3, [

              infoF1.phone,

              infoF1.code,

              infoF1.invite,

*money*,

*money*,

            ]);

          }

        }

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

          "SELECT `phone`, `code`, `invite`, `rank`, `user\_level`, `total\_money` FROM users WHERE code = ? AND veri = 1 LIMIT 1 ",

          [infoF1.invite],

        );

        if (fNext.length > 0) {

          infoF1 = fNext[0];

        } else {

          break;

        }

      }

    }

  }

};

This code defines an async function rosesPlus which takes two parameters auth and money. Inside the function, it performs a series of database queries using an object called connection.

1. First, it retrieves a record from the level table and stores it in the level variable. Then, it retrieves user information based on the provided auth token from the users table and stores it in the userInfo variable. It also retrieves information about a user (referred to as f1) based on the invite code from another query.
2. The code checks if the total money of the user is greater than or equal to 100 and if there is a f1 user. If these conditions are met, it iterates through levels and performs calculations to update the user's rosesF balance, make database updates, and insert new records into the database.
3. Finally, the code continues to query for the next invited user (fNext) and repeats the process until there are no more users to process.

Here is an example usage of the rosesPlus function:

const auth = 'user-token';

const money = 100;

rosesPlus(auth, money)

.then(() => {

console.log('Roses calculated and inserted successfully.');

})

.catch((error) => {

console.error('An error occurred:', error);

});

# **rosesPlus1**

rosesPlus1 is an asynchronous function that takes three parameters: phone, money, and levels. It also has an optional parameter timeNow.

This function retrieves data from the users table using an SQL query and performs a series of operations to calculate commissions for referrers based on the provided levels array and money parameter. It inserts commission data into the commissions table and updates the money field for certain users in the database.

*const* rosesPlus1 = *async* (*phone*, *money*, *levels* = [], *timeNow* = "") *=>* {

  try {

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

      "SELECT `phone`, `code`, `invite`, `money` FROM users WHERE phone = ? AND veri = 1 LIMIT 1",

      [*phone*],

    );

*const* userInfo = userResult[0];

    if (!userInfo) {

      return;

    }

*let* userReferrer = userInfo.invite;

*let* commissionsToInsert = [];

*let* usersToUpdate = [];

*let* timeNow1 = *Date*.now();

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

*const* levelCommission = *levels*[i] \* *money*;

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

        "SELECT phone, money, code, invite FROM users WHERE code = ?",

        [userReferrer],

      );

*const* referrerInfo = referrerRows[0];

      if (referrerInfo) {

*const* commissionId = generateCommissionId();

        commissionsToInsert.push([

          commissionId,

          referrerInfo.phone,

          userInfo.phone,

          levelCommission,

          i + 1,

          timeNow1,

        ]);

        usersToUpdate.push([levelCommission, referrerInfo.phone]);

        userReferrer = referrerInfo.invite;

      } else {

        console.log(`Level ${i + 1} referrer not found.`);

        break;

      }

    }

    if (commissionsToInsert.length > 0) {

      await connection.query(

        "INSERT INTO commissions (commission\_id, phone, from\_user\_phone, money, level, time) VALUES ?",

        [commissionsToInsert],

      );

    }

    if (usersToUpdate.length > 0) {

*const* updatePromises = usersToUpdate.map(([*money*, *phone*]) *=>*

        connection.query("UPDATE users SET money = money + ? WHERE phone = ?", [

*money*,

*phone*,

        ]),

      );

      await *Promise*.all(updatePromises);

    }

    return {

      success: true,

      message: "Commissions calculated and inserted successfully.",

    };

  } catch (error) {

    console.error(error);

    return { success: false, message: error.message };

  }

};

The provided code defines an asynchronous function named rosesPlus1 that takes phone, money, levels, and timeNow as parameters.

1. It tries to fetch user details based on phone from a database table using a SQL SELECT query. If the user exists, it calculates commissions based on the provided levels and money values.
2. It iterates over each level, fetches the referrer details, generates a commission ID, and prepares data to insert commissions and update user balances accordingly in the database. Finally, it returns a success message if the process completes without errors.
3. In case of any errors during the execution (like database connection errors), it catches the error, logs it, and returns an object with a failure message containing the error message.

Here is an example usage of the rosesPlus1 function:

const phone = 'user-phone';

const money = 100;

const levels = [0.1, 0.2, 0.3, 0.4, 0.5];

rosesPlus1(phone, money, levels)

.then((result) => {

console.log(result.message);

})

.catch((error) => {

console.error('An error occurred:', error);

});

# **Constants**

### TRX\_WINGO\_GAME\_STATUS\_MAP

This constant maps the status codes of the TRX Wingo game.

export *const* TRX\_WINGO\_GAME\_STATUS\_MAP = {

  PENDING: 0,

  COMPLETED: 1,

};

### TRX\_WINGO\_GAME\_TYPE\_MAP

This constant maps the game types of the TRX Wingo game.

export *const* TRX\_WINGO\_GAME\_TYPE\_MAP = {

  MIN\_1: "trx\_wingo",

  MIN\_3: "trx\_wingo3",

  MIN\_5: "trx\_wingo5",

  MIN\_10: "trx\_wingo10",

};

# **Variables**

### GameRepresentationIds

This variable imports the game representation IDs used in the project.

import GameRepresentationIds from "../constants/game\_representation\_id.js";

### generateCommissionId, generatePeriod, yesterdayTime

These variables import helper functions used in the code.

import { generateCommissionId,

  generatePeriod,

  yesterdayTime, } from "../helpers/games.js";

### moment, connection, axios, \_

These variables import external libraries and modules used in the code.

import moment from "moment";

import connection from "../config/connectDB.js";

import axios from "axios";

import \_ from "lodash";