โครงสร้างของไฟล์

```
my-backend/
├─ .env
├─ package.json
├─ src/
│  ├─ config
│  │   └─ cloudnary.js
│  │   └─ db_mydb.js
│  │   └─ globalKey.js
│  └─ controler
│  │   └─ users.js// และ อื่นๆตาม collection ทีมี
│  ├─ middleware
│  │   └─ auth.js
│  └─ router
│  │   └─ route.js
│  ├─ service
│  │   └─ message.js
│  │   └─ response.js
│  │   └─ service.js
│  │   └─ validate.js
│  ├─ index.js
├─ prisma/
│  └─ schema.prisma
```

โครงสร้างของ MongoDB Collection

```
use travel_booking;

/* ==========  countries  ========== */
db.createCollection("countries", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "iso2", "iso3", "currency"],
      properties: {
        name: { bsonType: "string" },
        iso2: { bsonType: "string", description: "2-letter code" },
        iso3: { bsonType: "string", description: "3-letter code" },
        phoneCode: { bsonType: "string" },
        currency: {
          bsonType: "object",
          required: ["code"],
          properties: {
            code: { bsonType: "string" }, // เช่น LAK, USD, THB
            name: { bsonType: "string" },
            symbol: { bsonType: "string" }
```

```
        }
       },
       createdAt: { bsonType: "date" },
       updatedAt: { bsonType: "date" }
     }
    }
  }
});
db.countries.createIndex({ iso2: 1 }, { unique: true });
db.countries.createIndex({ iso3: 1 }, { unique: true });

/* ==========  provinces  ========== */
db.createCollection("provinces", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "country_id"],
      properties: {
        name: { bsonType: "string" },
        country_id: { bsonType: "objectId" },
        createdAt: { bsonType: "date" },
        updatedAt: { bsonType: "date" }
      }
    }
  }
});
db.provinces.createIndex({ country_id: 1, name: 1 }, { unique: true });

/* ==========  cities  ========== */
db.createCollection("cities", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "province_id", "country_id"],
      properties: {
        name: { bsonType: "string" },
        province_id: { bsonType: "objectId" },
        country_id: { bsonType: "objectId" },
        location: {
          bsonType: "object",
          properties: {
            type: { enum: ["Point"] },
            coordinates: {
              bsonType: "array",
              items: [{ bsonType: "double" }, { bsonType: "double" }],
              description: "[lng, lat]"
            }
          }
```

```
      },
      createdAt: { bsonType: "date" },
      updatedAt: { bsonType: "date" }
    }
  }
 }
});
db.cities.createIndex({ province_id: 1, name: 1 }, { unique: true });
db.cities.createIndex({ location: "2dsphere" });

/* ==========  attractions  ========== */
db.createCollection("attractions", {
 validator: {
  $jsonSchema: {
   bsonType: "object",
   required: ["name", "city_id", "province_id", "country_id"],
   properties: {
    name: { bsonType: "string" },
    description: { bsonType: "string" },
    city_id: { bsonType: "objectId" },
    province_id: { bsonType: "objectId" },
    country_id: { bsonType: "objectId" },
    location: {
     bsonType: "object",
     properties: {
      type: { enum: ["Point"] },
      coordinates: {
       bsonType: "array",
       items: [{ bsonType: "double" }, { bsonType: "double" }],
       description: "[lng, lat]"
      }
     }
    },
    categories: { bsonType: "array", items: { bsonType: "string" } },
    images: { bsonType: "array", items: { bsonType: "string" } },
    ratingAvg: { bsonType: ["double", "int"] },
    ratingCount: { bsonType: "int" },
    isActive: { bsonType: "bool" },
    createdAt: { bsonType: "date" },
    updatedAt: { bsonType: "date" }
   }
  }
 }
});
db.attractions.createIndex({ country_id: 1, province_id: 1, city_id: 1 });
db.attractions.createIndex({ location: "2dsphere" });
db.attractions.createIndex({ name: "text", description: "text" });
```

```
/* ==========  packages  ========== */
db.createCollection("packages", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "code", "baseCurrency", "priceAdult", "durationDays", "isActive"],
      properties: {
        name: { bsonType: "string" },
        code: { bsonType: "string" },
        description: { bsonType: "string" },
        durationDays: { bsonType: "int" },
        startCity_id: { bsonType: "objectId" },
        country_id: { bsonType: "objectId" },
        baseCurrency: { bsonType: "string" }, // เช่น LAK / USD / THB
        priceAdult: { bsonType: ["double", "int"] },
        priceChild: { bsonType: ["double", "int"] },
        images: { bsonType: "array", items: { bsonType: "string" } },
        itinerary: {
          bsonType: "array",
          items: {
            bsonType: "object",
            required: ["day", "title"],
            properties: {
              day: { bsonType: "int" },
              title: { bsonType: "string" },
              detail: { bsonType: "string" },
              attractions: { bsonType: "array", items: { bsonType: "objectId" } }
            }
          }
        },
        availableDates: { bsonType: "array", items: { bsonType: "date" } },
        tags: { bsonType: "array", items: { bsonType: "string" } },
        ratingAvg: { bsonType: ["double", "int"] },
        ratingCount: { bsonType: "int" },
        isActive: { bsonType: "bool" },
        createdAt: { bsonType: "date" },
        updatedAt: { bsonType: "date" }
      }
    }
  }
});
db.packages.createIndex({ code: 1 }, { unique: true });
db.packages.createIndex({ country_id: 1, isActive: 1 });
db.packages.createIndex({ name: "text", description: "text", tags: "text" });

/* ==========  users  ========== */
db.createCollection("users", {
  validator: {
```

```
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "email"],
      properties: {
        name: { bsonType: "string" },
        email: { bsonType: "string" },
        phone: { bsonType: "string" },
        role: { enum: ["customer", "admin", "staff"] },
        passwordHash: { bsonType: "string" },
        passport: {
          bsonType: "object",
          properties: {
            number: { bsonType: "string" },
            nationality: { bsonType: "string" },
            expiryDate: { bsonType: "date" }
          }
        },
        addresses: {
          bsonType: "array",
          items: {
            bsonType: "object",
            properties: {
              label: { bsonType: "string" },
              line1: { bsonType: "string" },
              city: { bsonType: "string" },
              province: { bsonType: "string" },
              country: { bsonType: "string" },
              zip: { bsonType: "string" }
            }
          }
        },
        loyaltyPoints: { bsonType: "int" },
        createdAt: { bsonType: "date" },
        updatedAt: { bsonType: "date" }
      }
    }
  }
});
db.users.createIndex({ email: 1 }, { unique: true });
db.users.createIndex({ phone: 1 }, { unique: true, sparse: true });

/* ==========  bookings  (มี items ฝังอยู่)  ========== */
db.createCollection("bookings", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["bookingNo", "user_id", "status", "items", "currency", "amounts"],
      properties: {
```

```
bookingNo: { bsonType: "string" }, // เลขทีเอกสาร
user_id: { bsonType: "objectId" },
status: { enum: ["pending", "paid", "confirmed", "completed", "canceled", "refunded"] },
travelWindow: {
  bsonType: "object",
  properties: {
    startDate: { bsonType: "date" },
    endDate: { bsonType: "date" }
  }
},
travelers: {
  bsonType: "array",
  items: {
    bsonType: "object",
    required: ["type"],
    properties: {
      type: { enum: ["adult", "child", "infant"] },
      firstName: { bsonType: "string" },
      lastName: { bsonType: "string" },
      dob: { bsonType: "date" },
      passportNo: { bsonType: "string" }
    }
  }
},
items: {
  bsonType: "array",
  minItems: 1,
  items: {
    bsonType: "object",
    required: ["package_id", "title", "qtyAdults", "priceAdult"],
    properties: {
      package_id: { bsonType: "objectId" },
      title: { bsonType: "string" }, // snapshot ชื่อแพ็กเกจตอนจอง
      qtyAdults: { bsonType: "int" },
      qtyChildren: { bsonType: "int" },
      priceAdult: { bsonType: ["double", "int"] },
      priceChild: { bsonType: ["double", "int"] },
      options: {
        bsonType: "array",
        items: {
          bsonType: "object",
          properties: {
            name: { bsonType: "string" },
            price: { bsonType: ["double", "int"] }
          }
        }
      },
      subtotal: { bsonType: ["double", "int"] }
```

```
          }
        }
      },
      currency: { bsonType: "string" }, // LAK/USD/THB
      amounts: {
        bsonType: "object",
        required: ["itemsTotal", "discount", "tax", "fee", "grandTotal"],
        properties: {
          itemsTotal: { bsonType: ["double", "int"] },
          discount: { bsonType: ["double", "int"] },
          tax: { bsonType: ["double", "int"] },
          fee: { bsonType: ["double", "int"] },
          grandTotal: { bsonType: ["double", "int"] }
        }
      },
      payment: {
        bsonType: "object",
        properties: {
          method: { bsonType: "string" }, // card, cash, bank, wallet
          status: { enum: ["unpaid", "paid", "failed", "refunded", "partial"] },
          paidAt: { bsonType: "date" },
          transactions: {
            bsonType: "array",
            items: {
              bsonType: "object",
              properties: {
                ref: { bsonType: "string" },
                amount: { bsonType: ["double", "int"] },
                at: { bsonType: "date" }
              }
            }
          }
        }
      },
      notes: { bsonType: "string" },
      createdAt: { bsonType: "date" },
      updatedAt: { bsonType: "date" }
    }
  }
 }
});
db.bookings.createIndex({ bookingNo: 1 }, { unique: true });
db.bookings.createIndex({ user_id: 1, status: 1 });
db.bookings.createIndex({ "travelWindow.startDate": 1 });

/* ==========  reviews  ========== */
db.createCollection("reviews", {
  validator: {
```

```
    $jsonSchema: {
      bsonType: "object",
      required: ["user_id", "rating", "target"],
      properties: {
        user_id: { bsonType: "objectId" },
        rating: { bsonType: "int", minimum: 1, maximum: 5 },
        comment: { bsonType: "string" },
        photos: { bsonType: "array", items: { bsonType: "string" } },
        // target = { type: 'package'|'attraction', id: ObjectId }
        target: {
          bsonType: "object",
          required: ["type", "id"],
          properties: {
            type: { enum: ["package", "attraction"] },
            id: { bsonType: "objectId" }
          }
        },
        createdAt: { bsonType: "date" },
        updatedAt: { bsonType: "date" }
      }
    }
  }
});
db.reviews.createIndex({ "target.type": 1, "target.id": 1 });
db.reviews.createIndex({ user_id: 1, "target.type": 1, "target.id": 1 }, { unique: true });
```

โครงสร้างของ Model prisma

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}

// ================== Country ==================
model Country {
  id          String     @id @default(auto()) @map("_id") @db.ObjectId
  name        String
  iso2        String     @unique
  iso3        String     @unique
  phoneCode   String?
```

```
  currency    Currency   // Embedded type แทน relation
  provinces   Province[]
  cities      City[]
  attractions Attraction[]
  packages    Package[]
  createdAt   DateTime?
  updatedAt   DateTime?
}

// ================== Currency Embedded Type ==================
type Currency {
  code   String
  name   String?
  symbol String?
}

// ================== Province ==================
model Province {
  id          String      @id @default(auto()) @map("_id") @db.ObjectId
  name        String
  country_id  String      @db.ObjectId
  country     Country     @relation(fields: [country_id], references: [id])
  cities      City[]
  attractions Attraction[]
  createdAt   DateTime?
  updatedAt   DateTime?
}

// ================== City ==================
model City {
  id          String      @id @default(auto()) @map("_id") @db.ObjectId
  name        String
  province_id String      @db.ObjectId
  province    Province    @relation(fields: [province_id], references: [id])
  country_id  String      @db.ObjectId
  country     Country     @relation(fields: [country_id], references: [id])
  location    Location?
  attractions Attraction[]
  packages    Package[]
  createdAt   DateTime?
  updatedAt   DateTime?
}

// ================== Location Embedded Type ==================
type Location {
  type        String
  coordinates Float[]
}
```

```
// ================== Attraction ==================
model Attraction {
  id          String    @id @default(auto()) @map("_id") @db.ObjectId
  name        String
  description String?
  city_id     String    @db.ObjectId
  city        City      @relation(fields: [city_id], references: [id])
  province_id String    @db.ObjectId
  province    Province  @relation(fields: [province_id], references: [id])
  country_id  String    @db.ObjectId
  country     Country   @relation(fields: [country_id], references: [id])
  location    Location?
  categories  String[]
  images      String[]
  ratingAvg   Float?
  ratingCount Int?
  isActive    Boolean?
  createdAt   DateTime?
  updatedAt   DateTime?
}

// ================== Package ==================
model Package {
  id             String      @id @default(auto()) @map("_id") @db.ObjectId
  name           String
  code           String      @unique
  description    String?
  durationDays   Int
  startCity_id   String?     @db.ObjectId
  startCity      City?       @relation(fields: [startCity_id], references: [id])
  country_id     String?     @db.ObjectId
  country        Country?    @relation(fields: [country_id], references: [id])
  baseCurrency   String
  priceAdult     Float
  priceChild     Float?
  images         String[]
  itinerary      Itinerary[]
  availableDates DateTime[]
  tags           String[]
  ratingAvg      Float?
  ratingCount    Int?
  isActive       Boolean
  bookings       Booking[]
  createdAt      DateTime?
  updatedAt      DateTime?
}
```

```
// ================== Itinerary Embedded Type ==================
type Itinerary {
  day         Int
  title       String
  detail      String?
  attractions String[] // ObjectId strings
}

// ================== User ==================
model User {
  id           String    @id @default(auto()) @map("_id") @db.ObjectId
  uuid         String    @unique @default(uuid())
  name         String
  email        String    @unique
  phone        String?   @unique
  role         String    @default("customer")
  passwordHash String?
  passport     Passport?
  addresses    Address[]
  loyaltyPoints Int      @default(0)
  bookings     Booking[]
  reviews      Review[]
  createdAt    DateTime  @default(now())
  updatedAt    DateTime  @updatedAt
}

// ================== Passport Embedded Type ==================
type Passport {
  number      String?
  nationality String?
  expiryDate  DateTime?
}

// ================== Address Embedded Type ==================
type Address {
  label    String?
  line1    String?
  city     String?
  province String?
  country  String?
  zip      String?
}

// ================== Booking ==================
model Booking {
  id          String        @id @default(auto()) @map("_id") @db.ObjectId
  bookingNo   String        @unique
  user_id     String        @db.ObjectId  // ตรงกับ MongoDB collection
```

```
  user         User          @relation(fields: [user_id], references: [id])
  status       String
  travelWindow TravelWindow?
  travelers    Traveler[]
  items        BookingItem[]
  currency     String
  amounts      Amounts
  payment      Payment?
  notes        String?
  package_id   String?       @db.ObjectId
  package      Package?      @relation(fields: [package_id], references: [id])
  createdAt    DateTime?
  updatedAt    DateTime?
}

// ================== TravelWindow Embedded Type ==================
type TravelWindow {
  startDate DateTime?
  endDate   DateTime?
}

// ================== Traveler Embedded Type ==================
type Traveler {
  type       String
  firstName  String?
  lastName   String?
  dob        DateTime?
  passportNo String?
}

// ================== BookingItem Embedded Type ==================
type BookingItem {
  package_id  String  @db.ObjectId
  title       String
  qtyAdults   Int
  qtyChildren Int?
  priceAdult  Float
  priceChild  Float?
  options     Option[]
  subtotal    Float?
}

// ================== Option Embedded Type ==================
type Option {
  name  String?
  price Float?
}
```

```
// ================== Amounts Embedded Type ==================
type Amounts {
  itemsTotal Float
  discount   Float
  tax        Float
  fee        Float
  grandTotal Float
}

// ================== Payment Embedded Type ==================
type Payment {
  method       String?
  status       String?
  paidAt       DateTime?
  transactions Transaction[]
}

// ================== Transaction Embedded Type ==================
type Transaction {
  ref    String?
  amount Float?
  at     DateTime?
}

// ================== Review ==================
model Review {
  id       String    @id @default(auto()) @map("_id") @db.ObjectId
  user_id  String    @db.ObjectId  // เปลี่ยนจาก userId เป็น user_id ให้ตรงกับ MongoDB
  user     User      @relation(fields: [user_id], references: [id])
  rating   Int
  comment  String?
  photos   String[]
  target   Target
  createdAt DateTime?
  updatedAt DateTime?
}

// ================== Target Embedded Type ==================
type Target {
  type String
  id   String  @db.ObjectId
}
```

โครงสร้างของ [Auth.js]

```
import { EMessage } from "../service/message.js";
import { SendError } from "../service/response.js";
import { VerifyToken } from "../service/service.js";

export const auth = async (req, res, next) => {
  try {
    const authorization = req.headers['authorization'];
    if (!authorization) {
      return SendError(res, 401, EMessage.Unauthorization);
    }

    const token = authorization.replace("Bearer ", "");
    console.log("Token received:", token);

    const verify = await VerifyToken(token);
    if (!verify) {
      return SendError(res, 401, EMessage.Unauthorization);
    }

    console.log("User verified:", verify);
    req.user = verify;
    next();
  } catch (error) {
    console.error("Auth middleware error:", error);
    return SendError(res, 401, EMessage.Unauthorization);
  }
};
```

โสชครงสร้างของ User.js

```
import { ObjectId } from "mongodb";
import getDB from "../config/db_mydb.js";
import { EMessage, SMessage } from "../service/message.js";
import { SendCreate, SendError, SendSuccess } from
"../service/response.js";
import { ValidateData } from "../service/validate.js";
import {
  Encrypt,
  Decrypt,
  GenerateToken,
  VerifyRefreshToken,
} from "../service/service.js";
import { v4 as uuidv4 } from "uuid";
```

```
// usersCollection เป็น async function
const usersCollection = async () => (await
getDB()).collection("users");

export default class UserController {
  static async Register(req, res) {
    try {
      const { name, email, phone, password, role } = req.body;
      const validate = await ValidateData({
        name,
        email,
        phone,
        password,
        role,
      });
      if (validate.length > 0)
        return SendError(res, 400, EMessage.BadRequest +
validate.join(","));

      const passwordHash = await Encrypt(password);
      const uuid = uuidv4(); // สร้าง uuid
      const user = {
        _id: new ObjectId(),
        uuid, // เพิ่ม uuid field
        name,
        email,
        phone,
        role: role || "customer",
        passwordHash,
        passport: {},
        addresses: [],
        loyaltyPoints: 0,
        createdAt: new Date(),
        updatedAt: new Date(),
      };

      const collection = await usersCollection();
      const result = await collection.insertOne(user);
      if (!result.insertedId) return SendError(res, 500,
EMessage.ErrInsert);

      delete user.passwordHash;
```

```
      return SendCreate(res, SMessage.Register, user);
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }

  static async Login(req, res) {
    try {
      const { email, password } = req.body;
      const validate = await ValidateData({ email, password });
      if (validate.length > 0)
        return SendError(res, 400, EMessage.BadRequest +
validate.join(","));

      const collection = await usersCollection();
      const user = await collection.findOne({ email });
      if (!user) return SendError(res, 404, EMessage.NotFound);

      const decrypted = await Decrypt(user.passwordHash);
      if (password !== decrypted)
        return SendError(res, 404, EMessage.IsNotMatch);

      delete user.passwordHash;

      // ใช้ uuid ในการสร้าง token (ถ้าไม่มี uuid ให้ใช้ _id)
      const tokenId = user.uuid || user._id.toString();
      const token = await GenerateToken(tokenId);
      const data = { ...user, ...token };
      return SendSuccess(res, SMessage.Login, data);
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }

  static async SelectAll(req, res) {
    try {
      const collection = await usersCollection();
      const users = await collection.find({}, {
        projection: { passwordHash: 0 } // ไม่ส่ง passwordHash
      }).toArray();
      if (!users || users.length === 0)
        return SendError(res, 404, EMessage.NotFound, "users");
      return SendSuccess(res, SMessage.SelectAll, users);
```

```
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }

  static async SelectOne(req, res) {
    try {
      const userIdentifier = req.params.userID;
      const collection = await usersCollection();

      // ลองหาด้วย uuid ก่อน ถ้าไม่เจอค่อยหาด้วย _id
      let user = await collection.findOne({ uuid: userIdentifier });
      if (!user && ObjectId.isValid(userIdentifier)) {
        user = await collection.findOne({ _id: new
ObjectId(userIdentifier) });
      }

      if (!user) return SendError(res, 404, EMessage.NotFound, "user");

      delete user.passwordHash;
      return SendSuccess(res, SMessage.SelectOne, user);
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }

  static async updateProfile(req, res) {
    try {
      const userIdentifier = req.params.userID;
      const { name, phone, email } = req.body;
      const validate = await ValidateData({ name, phone, email });
      if (validate.length > 0)
        return SendError(res, 400, EMessage.BadRequest +
validate.join(","));

      const collection = await usersCollection();

      // ลองหาด้วย uuid ก่อน ถ้าไม่เจอค่อยหาด้วย _id
      let filter = { uuid: userIdentifier };
      if (ObjectId.isValid(userIdentifier)) {
        filter = { $or: [{ uuid: userIdentifier }, { _id: new
ObjectId(userIdentifier) }] };
      }
```

```
      const result = await collection.findOneAndUpdate(
        filter,
        { $set: { name, phone, email, updatedAt: new Date() } },
        { returnDocument: "after", projection: { passwordHash: 0 } }
      );
      if (!result.value) return SendError(res, 404,
EMessage.ErrUpdate);
      return SendSuccess(res, SMessage.Update, result.value);
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }

  static async deleteUser(req, res) {
    try {
      const userIdentifier = req.params.userID;
      const collection = await usersCollection();

      // ลองหาด้วย uuid ก่อน ถ้าไม่เจอค่อยหาด้วย _id
      let filter = { uuid: userIdentifier };
      if (ObjectId.isValid(userIdentifier)) {
        filter = { $or: [{ uuid: userIdentifier }, { _id: new
ObjectId(userIdentifier) }] };
      }

      const user = await collection.findOne(filter);
      if (!user) return SendError(res, 404, EMessage.NotFound, "user");

      await collection.deleteOne(filter);
      return SendSuccess(res, SMessage.Delete);
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }

  static async RefreshToken(req, res) {
    try {
      const { refreshToken } = req.body;
      if (!refreshToken)
        return SendError(res, 400, EMessage.BadRequest +
"refreshToken");
```

```
      const result = await VerifyRefreshToken(refreshToken);
      if (!result) return SendError(res, 404, EMessage.NotFound);

      return SendSuccess(res, SMessage.Update, result);
    } catch (err) {
      return SendError(res, 500, EMessage.ServerInternal, err);
    }
  }
}
```

โครงสร้างของ routes.js

```
import express from "express";
import UserController from "../controllers/user.js";
import { auth } from "../middleware/auth.js";

const router = express.Router();
const user = "/user";

router.get(`${user}/selAll`, /*auth,*/ UserController.SelectAll);
router.get(`${user}/selOne/:userID`,/*auth,*/
UserController.SelectOne); // userID จะเป็น uuid
router.post(`${user}/login`, UserController.Login);
router.post(`${user}/register`, UserController.Register);
router.put(`${user}/refresh`, UserController.RefreshToken);
router.put(`${user}/update/:userID`, /*auth,*/
UserController.updateProfile); // userID จะเป็น uuid
router.delete(`${user}/delete/:userID`, /*auth,*/
UserController.deleteUser); // userID จะเป็น uuid

export default router;
```

ถ้าเรา คอมเม้น auth ดึงข้อมูลได้ แต่พอไม่คอมเม้น auth ดึงไม่ได้

โครงสร้างของ Service.js

```
import CryptoJS from "crypto-js";
import { PrismaClient } from "@prisma/client";
import { SECRETE_KEY, SECRETE_KEY_REFRESH } from
"../config/globalKey.js";
import { EMessage, SMessage } from "./message.js";
```

```
import jwt from "jsonwebtoken";

const prisma = new PrismaClient();

// Verify Refresh Token
export const VerifyRefreshToken = async (refreshToken) => {
  return new Promise(async (resolve, reject) => {
    try {
      jwt.verify(refreshToken, SECRETE_KEY_REFRESH, async (err, decode)
=> {
        if (err) {
          console.error("JWT verify error:", err);
          return reject("Refresh Token Invalid");
        }

        console.log("Decoded refresh token id:", decode.id);

        // ใช้ uuid ในการหา user
        const user = await prisma.user.findUnique({
          where: { uuid: decode.id },
          select: {
            id: true,
            uuid: true,
            name: true,
            email: true,
            phone: true,
            role: true,
            passport: true,
            addresses: true,
            loyaltyPoints: true
          }
        });

        if (!user) {
          console.error("User not found with uuid:", decode.id);
          return reject("Error Verify Refresh Token");
        }

        const tokens = await GenerateToken(user.uuid); // ใช้ uuid
        return resolve(tokens);
      });
    } catch (error) {
      console.error("VerifyRefreshToken error:", error);
```

```
      reject(error);
    }
  });
};

// Verify Access Token
export const VerifyToken = async (token) => {
  return new Promise(async (resolve, reject) => {
    try {
      jwt.verify(token, SECRETE_KEY, async (err, decode) => {
        if (err) {
          console.error("JWT verify error:", err);
          return reject("Token Invalid");
        }

        console.log("Decoded token id:", decode.id);

        const user = await prisma.user.findUnique({
          where: { uuid: decode.id }, // ใช้ uuid
          select: {
            id: true,
            uuid: true,
            name: true,
            email: true,
            phone: true,
            role: true,
            passport: true,
            addresses: true,
            loyaltyPoints: true
          }
        });

        if (!user) {
          console.error("User not found with uuid:", decode.id);
          return reject("Error Verify Token");
        }

        console.log("User found:", user);
        return resolve(user);
      });
    } catch (error) {
      console.error("VerifyToken error:", error);
      reject(error);
```

```
    }
  });
};

// Generate Token
export const GenerateToken = async (userUuid) => {
  const payload = { id: userUuid };
  const payload_refresh = { id: userUuid };

  const token = jwt.sign(payload, SECRETE_KEY, { expiresIn: "3h" });
  const refreshToken = jwt.sign(payload_refresh, SECRETE_KEY_REFRESH, {
    expiresIn: "5h",
  });

  return { token, refreshToken };
};

export const FindOneOrder = async (orderID) => {
  return new Promise(async (resolve, reject) => {
    try {
      const data = await prisma.order.findFirst({
        where: { orderID: orderID },
      });
      if (!data) {
        reject(EMessage.NotFound + "order");
      }
      resolve(data);
    } catch (error) {
      console.log(error);
      reject(error);
    }
  });
};

export const FindOneProduct = async (productID) => {
  return new Promise(async (resolve, reject) => {
    try {
      const data = await prisma.product.findFirst({
        where: { productID: productID },
      });
      if (!data) {
        reject(EMessage.NotFound + "product");
      }
```

```
      resolve(data);
    } catch (error) {
      console.log(error);
      reject(error);
    }
  });
};

export const FindOneEmail = async (email) => {
  return new Promise(async (resolve, reject) => {
    try {
      const data = await prisma.user.findFirst({
        where: { email: email }
      });
      if (!data) {
        reject(EMessage.NotFound + "Email");
      }
      resolve(data);
    } catch (error) {
      console.log(error);
      reject(error);
    }
  });
};

export const CheckEmail = async (email) => {
  return new Promise(async (resolve, reject) => {
    try {
      const data = await prisma.user.findFirst({
        where: { email: email }
      });
      if (data) {
        reject(SMessage.Already);
      }
      resolve(true);
    } catch (error) {
      reject(error);
    }
  });
};

export const Encrypt = async (data) => {
  return CryptoJS.AES.encrypt(data, SECRETE_KEY).toString();
```

```
};

export const Decrypt = async (data) => {
  return CryptoJS.AES.decrypt(data,
SECRETE_KEY).toString(CryptoJS.enc.Utf8);
};
```