**✈️ Flight Management Backend – Request Flow Documentation**

**1. Application Startup**

* **Entry Point:** src/index.js
* Creates Express app → Adds middlewares (json, urlencoded) → Mounts /api routes → Starts server at PORT (from .env or default 3000).

**2. Routing Structure**

1. /api → handled in routes/index.js
2. /api/V1 → handled in routes/V1/index.js
3. /api/V1/airoplane → handled in routes/V1/airoplane-route.js
4. POST / → calls AiroplaneController.createAiroplane

**3. Controller Layer**

* File: controllers/airoplane-controller.js
* Job: Accepts request, extracts input (modelNumber, capacity), calls Service.
* Returns:
  + ✅ **201** + created airplane (on success)
  + ❌ **500** + error details (on failure)

**4. Service Layer**

* File: services/airoplane-service.js
* Job: Handles business logic (currently just delegates to repository).
* Calls: airoplaneRepository.create(data)
* Returns: Created airplane record (or error).

**5. Repository Layer**

* File: repositories/airoplane-repository.js
* Inherits from CrudRepository.
* Points to Airoplane model.

**CrudRepository (common operations)**

* create(data) → uses Sequelize’s .create() to save record.
* Logs error if DB operation fails.

**6. Model Layer**

* File: models/airoplane.js
* Sequelize definition:
  + modelNumber: STRING, not null
  + capacity: INTEGER, not null
* Sequelize handles DB interaction.

**7. Full Request → Response Flow**

Client (POST /api/V1/airoplane/)

↓

Express Routers

↓

Controller (createAiroplane)

↓

Service (createAiroplane)

↓

Repository (create)

↓

CRUD Repository → Sequelize Model → Database

↑

Response (success/error) back through layers

**8. Responses**

**✅ Success (201 Created)**

{

"success": true,

"message": "Successfully create an Airoplane",

"data": {

"id": 1,

"modelNumber": "Boeing-737",

"capacity": 180,

"createdAt": "...",

"updatedAt": "..."

},

"error": {}

}

**❌ Error (500 Internal Server Error)**

{

"success": false,

"message": "Something went wrong while creating a airplane",

"data": {},

"error": "Error details here"

}

👉 This is a **layered architecture (MVC + Repository + Service)** that keeps each responsibility separate:

* **Controller** = API handling
* **Service** = Business logic
* **Repository** = Database operations
* **Model** = Data schema

**Model Creation**

* Create New Model

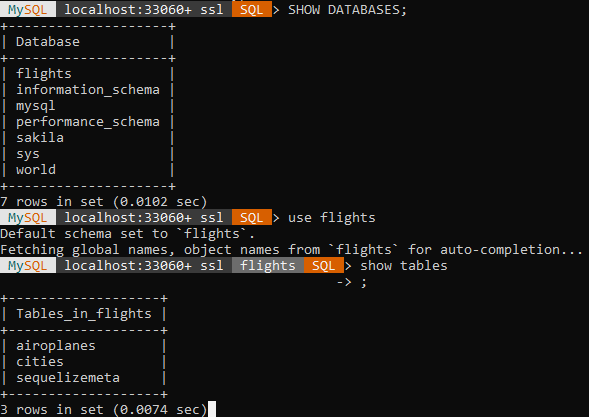
npx sequelize-cli model:generate --name City --attributes names:string

When we use npx sequelize-cli model:generate, Sequelize creates both a **model file** (under models/) and a **migration file** (under migrations/). The generated migration file contains extra default fields like id, createdAt, and updatedAt along with our defined attributes. If we only need the migration file (without the model), we can directly run the migration:generate command instead.

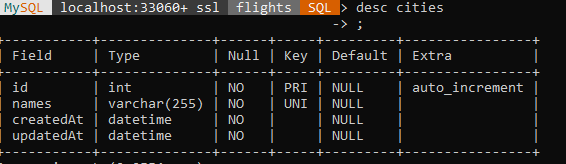
* Create Model based an migrate

npx sequelize db:migrate

it migrates all code in migrate file and create model table in Database.



Cities tables are created.



With these attributes.

PROMPT FOR CREATION STRUCTURE

Create a Node.js + Express backend with Sequelize following this exact structure and conventions:

- package.json scripts:

- "dev": nodemon src/index.js

- "start": node src/index.js

- "migrate": sequelize db:migrate

- "seed": sequelize db:seed:all

- Dependencies:

- express, sequelize, pg pg-hstore OR mysql2 (parametrize by env), dotenv, winston, cors, joi

- devDeps: nodemon, sequelize-cli

- Project layout:

src/

index.js

config/

index.js

server-config.js

logger-config.js

config.json // Sequelize environments; load from env

controllers/

airplane-controller.js

city-controller.js

info-controller.js

index.js

middlewares/

airplane-middleware.js // request validation with Joi

city-middleware.js

index.js

models/

index.js // Sequelize init and associations

airplane.js

city.js

airport.js

migrations/

TIMESTAMP-create-airplane.js

TIMESTAMP-create-city.js

TIMESTAMP-create-airport.js

seeders/

TIMESTAMP-add-airplanes.js

repositories/

crud-repository.js // Base class with create/get/update/destroy and pagination

airplane-repository.js // Extends CrudRepository; add domain-specific queries

city-repository.js

index.js // export { AirplaneRepository, CityRepository }

services/

airplane-service.js // Business logic; uses repositories; throws AppError

city-service.js

index.js

routes/

index.js // mounts /api

V1/

index.js // mounts /v1

airplane-routes.js // uses middlewares, controllers

city-routes.js

utils/

app-error.js // typed error with statusCode, explanation, details

index.js

common/

index.js

error-response.js // { success:false, message, data:null, err }

success-response.js // { success:true, message, data, err:{} }

logs/

combined.log

error.log

- Conventions:

- Use layered architecture: controllers → services → repositories → models.

- Versioned API under /api/v1 with airplane and city resources.

- Controller methods return standardized success/error responses from utils/common.

- Services encapsulate business rules and throw AppError on failures.

- Repositories encapsulate Sequelize access; CrudRepository provides generic methods.

- Use Winston logger configured via logger-config.js writing to logs/combined.log and logs/error.log.

- Validation middlewares with Joi for create/update endpoints.

- Environment-driven config: PORT, DB credentials, DIALECT, LOG\_LEVEL.

- Add a health/info controller at GET /api/v1/info.

- Endpoints to scaffold:

- Airplanes: GET /, GET /:id, POST /, PATCH /:id, DELETE /:id

- Cities: same set as airplanes

- Info: GET /api/v1/info → { status:"ok", uptime, version }

- Sequelize specifics:

- Models: Airplane(id, modelNumber:string, capacity:number), City(id, name:string, code:string unique), Airport(id, name, code unique, address, cityId FK).

- Associations: City hasMany Airport; Airport belongsTo City.

- Create matching migrations and a sample seeder for Airplanes.

- Error handling:

- Global error handler maps AppError to HTTP status; unknown errors → 500.

- Not found handler for unmatched routes.

- Centralized logger for request/response and errors.

- Quality:

- Add basic ESLint + Prettier config (optional).

- Nodemon config to watch src.

Deliver runnable code with placeholders wired correctly, correct spellings ("airplane", "utils"), and ensure imports in repository index point to './airplane-repository' and './city-repository'.

**AUTH**

**�� JWT Authentication Implementation Summary**

I have successfully implemented a complete JWT authentication system for your Sahaja Krushi backend. Here's what has been created:

**✅ Files Created/Modified:**

1. **User Model** (src/models/user.js)

* User table with id, username, email, password, role, isActive fields
* Role-based system: SUPER\_ADMIN and ADMIN

1. **JWT Utilities** (src/utils/jwt.js)

* Token generation and verification functions
* Configurable secret and expiration

1. **Password Utilities** (src/utils/password.js)

* Password hashing with bcrypt
* Password comparison for login

1. **Authentication Middleware** (src/middlewares/auth-middleware.js)

* authenticate() - Verifies JWT tokens
* authorize() - Role-based access control

1. **User Repository** (src/repositories/user-repository.js)

* Database operations for users
* Password hashing and validation

1. **Authentication Service** (src/services/auth-service.js)

* Business logic for authentication
* User management operations

1. **Authentication Controller** (src/controllers/auth-controller.js)

* API endpoints for login and user management
* Error handling and response formatting

1. **Authentication Routes** (src/routes/V1/auth-routes.js)

* Public login endpoint
* Protected user management endpoints (SUPER\_ADMIN only)

1. **Database Migration** (migrations/20250906122422-create-users.js)

* Creates users table with all required fields

1. **Super Admin Seeder** (seeders/20250906122528-create-super-admin.js)

* Creates default super admin user

1. **Updated Routes** (src/routes/V1/index.js)

* Added auth routes to the API

**🔧 Database Setup:**

* ✅ Migration run successfully - users table created
* ✅ Seeder run successfully - super admin user created

**🔐 Authentication Flow:**

1. **Super Admin Login:**

* Email: superadmin@sahajakrushi.com
* Password: superadmin123
* Role: SUPER\_ADMIN

1. **API Endpoints:**

* POST /api/V1/auth/login - Login (public)
* POST /api/V1/auth/users - Create user (SUPER\_ADMIN only)
* GET /api/V1/auth/users - Get all users (SUPER\_ADMIN only)
* GET /api/V1/auth/users/:id - Get user by ID (SUPER\_ADMIN only)
* PUT /api/V1/auth/users/:id - Update user (SUPER\_ADMIN only)
* DELETE /api/V1/auth/users/:id - Delete user (SUPER\_ADMIN only)

**🔒 Security Features:**

* ✅ JWT token-based authentication
* ✅ Password hashing with bcrypt
* ✅ Role-based authorization
* ✅ Token expiration (24 hours default)
* ✅ Input validation
* ✅ Error handling

**�� Key Features:**

* ✅ No signup - only super admin can create users
* ✅ Super admin can create admin users
* ✅ Both can login with their credentials
* ✅ Protected routes require valid JWT token
* ✅ Role-based access control
* ✅ Secure password storage

The authentication system is now fully functional and ready to use! The super admin can create other admin users, and all users can login using their credentials. All protected routes require a valid JWT token in the Authorization header.

Review Changes