To create a modern CRUD API in Node.js with TypeScript and PostgreSQL, we'll follow a step-by-step process. Below is a folder structure and example files for your project:

### Folder Structure:

```

hospital-admission-control/

│

├── src/

│ ├── controllers/

│ │ └── admissionController.ts

│ ├── entities/

│ │ └── Admission.ts

│ ├── interfaces/

│ │ └── AdmissionInterface.ts

│ ├── database/

│ │ └── db.ts

│ ├── routes/

│ │ └── admissionRoutes.ts

│ ├── app.ts

│ └── server.ts

│

├── .env

├── .eslintignore

├── .eslintrc.json

├── package.json

└── tsconfig.json

```

### 1. Initialize Project:

```bash

mkdir hospital-admission-control

cd hospital-admission-control

npm init -y

```

### 2. Install Dependencies:

```bash

npm install express pg dotenv typescript @types/node @types/express body-parser eslint httpclient

```

### Example Files:

#### `.env`:

```plaintext

DB\_HOST=localhost

DB\_USER=your\_username

DB\_PASSWORD=your\_password

DB\_NAME=your\_database\_name

DB\_PORT=5432

```

#### `tsconfig.json`:

```json

{

"compilerOptions": {

"target": "ES2018",

"module": "CommonJS",

"strict": true,

"esModuleInterop": true,

"skipLibCheck": true,

"forceConsistentCasingInFileNames": true

},

"include": ["src/\*\*/\*.ts"],

"exclude": ["node\_modules"]

}

```

#### `.eslintrc.json`:

```json

{

"root": true,

"parser": "@typescript-eslint/parser",

"plugins": ["@typescript-eslint"],

"extends": ["eslint:recommended", "plugin:@typescript-eslint/recommended"],

"rules": {

"@typescript-eslint/explicit-module-boundary-types": "off"

}

}

```

#### `src/app.ts`:



#### `src/server.ts`:



#### `src/database/db.ts`:



#### `src/entities/Customer.ts`:

#### `src/controllers/admissionController.ts`:

### Testing:

For testing, you can use tools like `jest` along with `supertest` for API endpoint testing and `jest` for unit testing the functions inside controllers.

```bash

npm install jest @types/jest supertest @types/supertest --save-dev

```

You can then create test files for each component and run tests using `

npm test`.

This setup provides a structured, object-oriented, and modular approach to building a CRUD API in Node.js with TypeScript and PostgreSQL, including detailed error handling and testing capabilities.

Sure, let's add logic examples for all CRUD operations and endpoint routes in the `admissionController.ts` file.

```typescript

import { Request, Response } from 'express';

import pool from '../database/db';

import Admission from '../entities/Admission';

class AdmissionController {

async createAdmission(req: Request, res: Response) {

try {

const { patientName, admissionDate } = req.body;

// Validate request body

if (!patientName || !admissionDate) {

return res.status(400).json({ message: 'Patient name and admission date are required' });

}

// Insert admission into the database

const result = await pool.query('INSERT INTO admissions (patient\_name, admission\_date) VALUES ($1, $2) RETURNING \*', [patientName, admissionDate]);

const newAdmission = new Admission(result.rows[0].id, patientName, admissionDate);

res.status(201).json({ message: 'Admission created successfully', admission: newAdmission });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

}

async getAdmissions(req: Request, res: Response) {

try {

// Retrieve all admissions from the database

const result = await pool.query('SELECT \* FROM admissions');

const admissions: Admission[] = result.rows.map((row: any) => new Admission(row.id, row.patient\_name, row.admission\_date));

res.status(200).json({ admissions });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

}

async updateAdmission(req: Request, res: Response) {

try {

const admissionId = parseInt(req.params.id);

const { patientName, admissionDate } = req.body;

// Validate request body

if (!patientName || !admissionDate) {

return res.status(400).json({ message: 'Patient name and admission date are required' });

}

// Check if admission exists

const existingAdmission = await pool.query('SELECT \* FROM admissions WHERE id = $1', [admissionId]);

if (existingAdmission.rows.length === 0) {

return res.status(404).json({ message: 'Admission not found' });

}

// Update admission in the database

await pool.query('UPDATE admissions SET patient\_name = $1, admission\_date = $2 WHERE id = $3', [patientName, admissionDate, admissionId]);

res.status(200).json({ message: 'Admission updated successfully' });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

}

async deleteAdmission(req: Request, res: Response) {

try {

const admissionId = parseInt(req.params.id);

// Check if admission exists

const existingAdmission = await pool.query('SELECT \* FROM admissions WHERE id = $1', [admissionId]);

if (existingAdmission.rows.length === 0) {

return res.status(404).json({ message: 'Admission not found' });

}

// Delete admission from the database

await pool.query('DELETE FROM admissions WHERE id = $1', [admissionId]);

res.status(200).json({ message: 'Admission deleted successfully' });

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

}

}

export default new AdmissionController();

```

This code provides logic for each CRUD operation (`create`, `read`, `update`, `delete`) in the `AdmissionController` class. It also includes detailed error handling for various scenarios such as missing fields, non-existing admissions, and internal server errors.

Certainly! Here's an example of how you can implement the logic for handling each CRUD method in the `admissionRoutes.ts` file:

```typescript

import express, { Request, Response } from 'express';

import admissionController from '../controllers/admissionController';

const router = express.Router();

// POST /admissions

router.post('/', async (req: Request, res: Response) => {

try {

await admissionController.createAdmission(req, res);

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

});

// GET /admissions

router.get('/', async (req: Request, res: Response) => {

try {

await admissionController.getAdmissions(req, res);

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

});

// PUT /admissions/:id

router.put('/:id', async (req: Request, res: Response) => {

try {

await admissionController.updateAdmission(req, res);

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

});

// DELETE /admissions/:id

router.delete('/:id', async (req: Request, res: Response) => {

try {

await admissionController.deleteAdmission(req, res);

} catch (error) {

console.error(error);

res.status(500).json({ message: 'Internal server error' });

}

});

export default router;

```

In this implementation, each route handler calls the respective method in the `admissionController`, and any errors occurring within these methods are caught and handled with a generic 500 Internal Server Error response. This ensures that your API remains robust and can handle unexpected errors gracefully.