Изображение выглядит как текст, диаграмма, Параллельный, линия

Контент, сгенерированный ИИ, может содержать ошибки.

# e-Tendering System Documentation

Project Structure

.

├── .gitignore

├── azure-pipelines.yml

├── category\_form.html

├── dbDiagram.drawio

├── eslint.config.js

├── index.html

├── package.json

├── README.md

├── server.js

├── test.html

├── testCreateCategory.html

├── testEdit.html

├── vite.config.js

├── models/

│ ├── bid.js

│ ├── category.js

│ ├── Tender.js

│ ├── user\_category.js

│ ├── user.js

│ ├── userType.js

├── public/

│ ├── icon.png

│ ├── vite.svg

└── src/

├── App.css

├── App.jsx

├── index.css

├── main.jsx

├── components/

├── Navbar.jsx

├── SubmitBid.jsx

├── CreateTender/

│ ├── CreateTender.css

│ ├── CreateTender.jsx

├── DetailedInfo/

│ ├── DetailedInfo.css

│ ├── DetailedInfo.jsx

├── Login/

│ ├── Login.css

│ ├── Login.jsx

├── TenderList/

│ ├── TenderList.css

│ ├── TenderList.jsx

Running the Project

* **Prerequisites**
  + Node.js (v20.x)
  + MongoDB Atlas or a local MongoDB instance

**Setup**

**1. Clone the repository:**

git clone https://e2205685@dev.azure.com/e2205685/e-Tendering/\_git/e-Tendering

cd e-Tendering

**2. Install dependencies:**

npm install

**3. Start the server:**

npm start

**4. Start the front-end development server:**

npm run dev

API Documentation

**Base URL:** http://localhost:5500

**Endpoints**

Users

* Create User
  + - URL: /create\_user
    - Method: POST
    - Body:

{

"name": "John Doe",

"address": "123 Main St",

"user\_type": "Admin",

"password": "password123",

"email": "john.doe@example.com",

"categories": []

}

* Response:

{

"message": "User created successfully",

"user": { ... }

}

* Get All Users
  + - URL: users
    - Method: GET
    - Response:

[

{

"user\_id": "USR-123456789",

"name": "John Doe",

"address": "123 Main St",

"user\_type": "Admin",

"email": "john.doe@example.com",

"categories": [ ... ]

},

...

]

* Delete User
  + - URL: /delete\_user/:user\_id`
    - Method: DELETE
    - Response:

{

"message": "User deleted successfully"

}

Tenders

* Create Tender
  + - URL: /save\_tender
    - Method: POST
    - Body:

{

"tender\_name": "Road Construction",

"construction\_from": "2025-04-01",

"construction\_to": "2025-10-01",

"date\_of\_tender\_notice": "2025-03-10",

"date\_of\_tender\_close": "2025-03-25",

"date\_of\_tender\_winner": "2025-03-30",

"bidding\_price": 5000000,

"tender\_status": "Open",

"staff\_id": "STF-12345"

}

* Response:

{

"message": "Tender saved successfully"

}

* Get All Tenders
  + - URL: /find
    - Method: GET
    - Response:

[

{

"tender\_id": "TND-123456789",

"tender\_name": "Road Construction",

"construction\_from": "2025-04-01",

"construction\_to": "2025-10-01",

"date\_of\_tender\_notice": "2025-03-10",

"date\_of\_tender\_close": "2025-03-25",

"date\_of\_tender\_winner": "2025-03-30",

"bidding\_price": 5000000,

"tender\_status": "Open",

"staff\_id": "STF-12345"

},

...

]

* Delete Tender
  + - URL: /delete\_tender/:tender\_id
    - Method: DELETE
    - Response:

{

"message": "Tender deleted successfully"

}

* Edit Tender

Endpoint

app.put('/update\_tender/:tender\_id', async (req, res) => {

try {

const { tender\_id } = req.params; // Extract tender ID from the URL

const updatedTender = await Tender.findOneAndUpdate(

{ tender\_id }, // Find the tender by its unique ID

req.body, // Update the tender with the data provided in the request body

{ new: true } // Return the updated tender document

);

if (!updatedTender) {

return res.status(404).json({ error: 'Tender not found' }); // Handle case where tender is not found

}

res.json({ message: 'Tender updated successfully', updatedTender }); // Respond with the updated tender

} catch (err) {

console.error('Error updating tender:', err); // Log any errors

res.status(500).json({ error: 'Error updating tender', details: err.message }); // Respond with an error message

}

});

How It Works

1. Request Method: PUT
2. URL: /update\_tender/:tender\_id
   * The:tender\_id in the URL is a placeholder for the unique ID of the tender to be updated.
3. Request Body:
   * The request body should contain the fields to be updated. For example:

{

"tender\_name": "Updated Tender Name",

"description": "Updated Description",

"construction\_from": "2025-04-01",

"construction\_to": "2025-10-01",

"bidding\_price": 6000000

}

1. Database Operation:
   * The Tender.findOneAndUpdate() function is used to locate the tender by its tender\_id and update its details with the data provided in the request body.
2. Response:
   * On success

{

"message": "Tender updated successfully",

"updatedTender": { ... }

}

* If the tender is not found

{

"error": "Tender not found"

}

Bids

* Create Bid
  + - URL: /create\_bid
    - Method: POST
    - Body:

{

"amount": 100000,

"user\_id": "USR-123456789",

"tender\_id": "TND-123456789"

}

* Response:

{

"message": "Bid created successfully",

"bid": { ... }

}

* Get All Bids
  + - URL: /bids
    - Method: GET
    - Response:

[

{

"bid\_id": "BID-123456789",

"amount": 100000,

"date": "2025-03-15T12:00:00Z",

"user": { ... },

"tender": { ... }

},

...

]

* Delete Bid
  + - URL: /delete\_bid/:bid\_id
    - Method: DELETE
    - Response:

{

"message": "Bid deleted successfully"

}

Categories

* Create Category
  + - URL: /create\_category
    - Method: POST
    - Body:

{

"category\_id": "CAT-123456789",

"category\_name": "Construction"

}

* Response:

{

"message": "Category created successfully",

"category": { ... }

}

* Get All Categories
  + - URL: /categories
    - Method: GET
    - Response:

[

{

"category\_id": "CAT-123456789",

"category\_name": "Construction",

"users": [ ... ]

},

...

]

* Edit Category

Endpoint

app.put('/update\_category/:category\_id', async (req, res) => {

try {

const { category\_id } = req.params; // Extract category ID from the URL

const updatedCategory = await Category.findOneAndUpdate(

{ category\_id }, // Find the category by its unique ID

req.body, // Update the category with the data provided in the request body

{ new: true } // Return the updated category document

);

if (!updatedCategory) {

return res.status(404).json({ error: 'Category not found' }); // Handle case where category is not found

}

res.json({ message: 'Category updated successfully', updatedCategory }); // Respond with the updated category

} catch (err) {

console.error('Error updating category:', err); // Log any errors

res.status(500).json({ error: 'Error updating category', details: err.message }); // Respond with an error message

}

});

How It Works

1. Request Method: PUT
2. URL: /update\_category/:category\_id
   * The :category\_id in the URL is a placeholder for the unique ID of the category to be updated.
3. Request Body:
   * The request body should contain the fields to be updated. For example:

{

"category\_name": "Updated Category Name"

}

1. Database Operation:
   * The Category.findOneAndUpdate() function is used to locate the category by its category\_id and update its details with the data provided in the request body.
2. Response:
   * On success

{

"message": "Category updated successfully",

"updatedCategory": { ... }

}

* If the category is not found

{

"error": "Category not found"

}

* Add User to Category
  + - URL: `/add\_user\_to\_category`
    - Method: POST
    - Body:

{

"user\_id": "USR-123456789",

"category\_id": "CAT-123456789"

}

* Response:

{

"message": "User added to category successfully"

}

* Remove User from Category
  + - URL: /remove\_user\_from\_category
    - Method: POST
    - Body:

{

"user\_id": "USR-123456789",

"category\_id": "CAT-123456789"

}

* Response:

{

"message": "User removed from category successfully"

}

User Types

* Create User Type
  + - URL: /create\_user\_typeuser
    - Method: POST
    - Body:

{

"type\_id": "UT-123456789",

"type\_name": "Admin"

}

* Response:

{

"message": "User type created successfully",

"userType": { ... }

}

* Get All User Types
  + - URL: /user\_types
    - Method: GET
    - Response:

[

{

"type\_id": "UT-123456789",

"type\_name": "Admin"

},

...

]

* Edit User Types

app.put('/update\_user\_type/:type\_id', async (req, res) => {

try {

const { type\_id } = req.params; // Extract user type ID from the URL

const updatedUserType = await UserType.findOneAndUpdate(

{ type\_id }, // Find the user type by its unique ID

req.body, // Update the user type with the data provided in the request body

{ new: true } // Return the updated user type document

);

if (!updatedUserType) {

return res.status(404).json({ error: 'User type not found' }); // Handle case where user type is not found

}

res.json({ message: 'User type updated successfully', updatedUserType }); // Respond with the updated user type

} catch (err) {

console.error('Error updating user type:', err); // Log any errors

res.status(500).json({ error: 'Error updating user type', details: err.message }); // Respond with an error message

}

});

How It Works

1. Request Method: PUT
2. URL: /update\_user\_type/:type\_id
   * The :type\_id in the URL is a placeholder for the unique ID of the user type to be updated.
3. Request Body:
   * The request body should contain the fields to be updated. For example:

{

"type\_name": "Updated User Type Name"

}

1. Database Operation:
   * The UserType.findOneAndUpdate() function is used to locate the user type by its type\_id and update its details with the data provided in the request body.
2. Response:
   * On success:

{

"message": "User type updated successfully",

"updatedUserType": { ... }

}

* If the user type is not found

{

"error": "User type not found"

}

Database Models

**User Model**

* File: user.js
* Schema:

const userSchema = new mongoose.Schema({

user\_id: { type: String, required: true, unique: true },

name: { type: String, required: true },

address: { type: String, required: true },

user\_type: { type: String, required: true },

password: { type: String, required: true },

lock: { type: Boolean, default: false },

email: { type: String, required: true, unique: true },

categories: [{ type: mongoose.Schema.Types.ObjectId, ref: 'CATEGORY' }],

tenders: [{ type: mongoose.Schema.Types.ObjectId, ref: 'TENDER' }],

bids: [{ type: mongoose.Schema.Types.ObjectId, ref: 'BID' }]

}, { collection: 'USER' });

Category Model

* File: category.js
* Schema:

const categorySchema = new mongoose.Schema({

category\_id: { type: String, required: true, unique: true },

category\_name: { type: String, required: true },

users: [{ type: mongoose.Schema.Types.ObjectId, ref: 'USER' }]

}, { collection: 'CATEGORY' });

Tender Model

* File: Tender.js
* Schema:

const tenderSchema = new mongoose.Schema({

tender\_id: { type: String, required: true, unique: true },

tender\_name: { type: String, required: true },

construction\_from: { type: Date, required: true },

construction\_to: { type: Date, required: true },

date\_of\_tender\_notice: { type: Date, required: true },

date\_of\_tender\_close: { type: Date, required: true },

date\_of\_tender\_winner: { type: Date },

bidding\_price: { type: Number, required: true, min: 0 },

tender\_status: { type: String, enum: ['Open', 'Closed', 'Pending'], default: 'Pending' },

staff\_id: { type: String, required: true },

winner: { type: mongoose.Schema.Types.ObjectId, ref: 'USER', default: null },

bids: [{ type: mongoose.Schema.Types.ObjectId, ref: 'BID' }]

}, { collection: 'TENDER' });

Bid Model

* File: bid.js
* Schema:

const bidSchema = new mongoose.Schema({

bid\_id: { type: String, required: true, unique: true },

amount: { type: Number, required: true, min: 0 },

date: { type: Date, default: Date.now },

user: { type: mongoose.Schema.Types.ObjectId, ref: 'USER', required: true },

tender: { type: mongoose.Schema.Types.ObjectId, ref: 'TENDER', required: true }

}, { collection: 'BID' });

User Type Model

* File: userType.js
* Schema:

const userTypeSchema = new mongoose.Schema({

type\_id: { type: String, required: true, unique: true },

type\_name: { type: String, required: true }

}, { collection: 'USER\_TYPE' });

Server.js Overview

* File: server.js
* Description: This file initializes the Express application, connects to MongoDB using Mongoose, and defines various API endpoints for managing users, tenders, bids, categories, and user types.

**Key Sections**

**1. Imports and Initialization:**

import mongoose from 'mongoose';

import express from 'express';

import cors from 'cors';

import User from './models/user.js';

import Category from './models/category.js';

import Tender from './models/tender.js';

import Bid from './models/bid.js';

import UserType from './models/userType.js';

const app = express();

const port = 5500;

app.use(cors());

app.use(express.json());

**2. MongoDB Connection:**

const uri = "mongodb+srv://storeDataUser:g1MfHieubCImPSXV@cluster0.noqzo.mongodb.net/e-Tendering?retryWrites=true&w=majority";

mongoose.connect(uri)

.then(() => console.log('Connected to MongoDB Atlas'))

.catch((err) => {

console.error('Failed to connect to MongoDB Atlas:', err.message);

console.error(err.stack);

});

**3. Error Logging Middleware:**

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Something broke!');

});

**4. API Endpoints:**

* Save Tender:

app.post('/save\_tender', (req, res) => {

const tender = new Tender({ ...req.body, tender\_id: 'TND-' + Date.now() });

tender.save()

.then(() => res.json({ message: 'Tender saved successfully' }))

.catch((err) => {

console.error('Error saving tender:', err);

res.status(500).json({ error: 'Error saving tender', details: err });

});

});

* Get All Tenders:

app.get('/find', (req, res) => {

Tender.find()

.then(tenders => res.json(tenders))

.catch((err) => {

console.error('Error fetching tenders:', err);

res.status(500).json({ error: 'Error fetching tenders', details: err });

});

});

* Create User:

app.post('/create\_user', async (req, res) => {

try {

const { name, address, user\_type, password, email, categories } = req.body;

const existingUser = await User.findOne({ email });

if (existingUser) {

return res.status(400).json({ error: 'Email already in use' });

}

const user\_id = 'USR-' + Date.now();

const user = new User({ user\_id, name, address, user\_type, password, email, categories });

await user.save();

res.json({ message: 'User created successfully', user });

} catch (err) {

console.error('Error creating user:', err);

res.status(500).json({ error: 'Error creating user', details: err.message });

}

});

* Get All Users:

app.get('/users', async (req, res) => {

try {

const users = await User.find().populate('categories').populate('user\_type');

res.json(users);

} catch (err) {

console.error('Error fetching users:', err);

res.status(500).json({ error: 'Error fetching users', details: err.message });

}

});

* Create Bid:

app.post('/create\_bid', async (req, res) => {

try {

const { amount, user\_id, tender\_id } = req.body;

const user = await User.findOne({ user\_id });

const tender = await Tender.findOne({ tender\_id });

if (!user || !tender) {

return res.status(400).json({ error: 'Invalid user or tender ID' });

}

const bid\_id = 'BID-' + Date.now();

const bid = new Bid({

bid\_id,

amount,

user: user.\_id,

tender: tender.\_id

});

await bid.save();

res.json({ message: 'Bid created successfully', bid });

} catch (err) {

console.error('Error creating bid:', err);

res.status(500).json({ error: 'Error creating bid', details: err.message });

}

});

* Get All Bids:

app.get('/bids', async (req, res) => {

try {

const bids = await Bid.find().populate('user').populate('tender');

res.json(bids);

} catch (err) {

console.error('Error fetching bids:', err);

res.status(500).json({ error: 'Error fetching bids', details: err.message });

}

});

* Create Category:

app.post('/create\_category', async (req, res) => {

try {

const { category\_id, category\_name } = req.body;

const category = new Category({ category\_id, category\_name });

await category.save();

res.json({ message: 'Category created successfully', category });

} catch (err) {

console.error('Error creating category:', err);

res.status(500).json({ error: 'Error creating category', details: err.message });

}

});

* Get All Categories:

app.get('/categories', async (req, res) => {

try {

const categories = await Category.find().populate('users');

res.json(categories);

} catch (err) {

console.error('Error fetching categories:', err);

res.status(500).json({ error: 'Error fetching categories', details: err.message });

}

});

* Add User to Category

app.post('/add\_user\_to\_category', async (req, res) => {

try {

const { user\_id, category\_id } = req.body;

const user = await User.findOne({ user\_id });

const category = await Category.findOne({ category\_id });

if (!user || !category) {

return res.status(400).json({ error: 'Invalid user or category ID' });

}

user.categories.push(category.\_id);

category.users.push(user.\_id);

await user.save();

await category.save();

res.json({ message: 'User added to category successfully' });

} catch (err) {

console.error('Error adding user to category:', err);

res.status(500).json({ error: 'Error adding user to category', details: err.message });

}

});

* Remove User from Category

app.post('/remove\_user\_from\_category', async (req, res) => {

try {

const { user\_id, category\_id } = req.body;

const user = await User.findOne({ user\_id });

const category = await Category.findOne({ category\_id });

if (!user || !category) {

return res.status(400).json({ error: 'Invalid user or category ID' });

}

user.categories.pull(category.\_id);

category.users.pull(user.\_id);

await user.save();

await category.save();

res.json({ message: 'User removed from category successfully' });

} catch (err) {

console.error('Error removing user from category:', err);

res.status(500).json({ error: 'Error removing user from category', details: err.message });

}

});

* Create User Type

app.post('/create\_user\_type', async (req, res) => {

try {

const { type\_id, type\_name } = req.body;

const userType = new UserType({ type\_id, type\_name });

await userType.save();

res.json({ message: 'User type created successfully', userType });

} catch (err) {

console.error('Error creating user type:', err);

res.status(500).json({ error: 'Error creating user type', details: err.message });

}

});

* Get All User Types

app.get('/user\_types', async (req, res) => {

try {

const userTypes = await UserType.find();

res.json(userTypes);

} catch (err) {

console.error('Error fetching user types:', err);

res.status(500).json({ error: 'Error fetching user types', details: err.message });

}

});

* Endpoint to Edit a User

app.put('/update\_user/:user\_id', async (req, res) => {

try {

const { user\_id } = req.params; // Extract the user ID from the URL parameters

const updatedUser = await User.findOneAndUpdate(

{ user\_id }, // Find the user by their unique `user\_id`

req.body, // Update the user with the data provided in the request body

{ new: true } // Return the updated user document

);

if (!updatedUser) {

return res.status(404).json({ error: 'User not found' }); // Handle case where user is not found

}

res.json({ message: 'User updated successfully', updatedUser }); // Respond with the updated user

} catch (err) {

console.error('Error updating user:', err); // Log any errors

res.status(500).json({ error: 'Error updating user', details: err.message }); // Respond with an error message

}

});

**How It Works**

1. **Request Method**: PUT
   * This method is used to update an existing resource (in this case, a user).
2. **URL**: /update\_user/:user\_id
   * The :user\_id in the URL is a placeholder for the unique ID of the user to be updated.
3. **Request Body**:
   * The request body should contain the fields to be updated. For example:

{

"name": "Updated Name",

"address": "Updated Address",

"user\_type": "Updated User Type",

"email": "updated.email@example.com"

}

1. **Database Operation**:
   * The User.findOneAndUpdate() function is used to locate the user by their user\_id and update their details with the data provided in the request body.
2. **Response**:
   * If the user is found and updated successfully, the server responds with:

{

"message": "User updated successfully",

"updatedUser": {

"user\_id": "USR-123456789",

"name": "Updated Name",

"address": "Updated Address",

"user\_type": "Updated User Type",

"email": "updated.email@example.com",

"categories": [...],

"tenders": [...],

"bids": [...]

}

}

{

"message": "User updated successfully",

"updatedUser": {

"user\_id": "USR-123456789",

"name": "Updated Name",

"address": "Updated Address",

"user\_type": "Updated User Type",

"email": "updated.email@example.com",

"categories": [...],

"tenders": [...],

"bids": [...]

}

}

If the user is not found, the server responds with:

{

"error": "User not found"

}

If there is an error during the update process, the server responds with:

{

"error": "Error updating user",

"details": "Detailed error message"

}

* How to Test the Endpoint

Using Postman or cURL

1. Request URL:

http://localhost:5500/update\_user/USR-123456789

1. Request Method:

PUT

1. Request Headers:

Content-Type: application/json

1. Request Body:

{

"name": "Updated Name",

"address": "Updated Address",

"user\_type": "Updated User Type",

"email": "updated.email@example.com"

}

1. Response:

* On success

{

"message": "User updated successfully",

"updatedUser": {

"user\_id": "USR-123456789",

"name": "Updated Name",

"address": "Updated Address",

"user\_type": "Updated User Type",

"email": "updated.email@example.com",

"categories": [...],

"tenders": [...],

"bids": [...]

}

}

**Key Points**

* Validation: Ensure that the user\_id exists in the database before attempting to update.
* Partial Updates: Only the fields provided in the request body will be updated. Other fields will remain unchanged.
* Error Handling: Proper error handling is implemented to manage cases where the user is not found or an internal server error occurs.