**Assignment 6: Error Handling and Validation**

**Implement error handling middleware in Express.js.**

**Add validation to your API routes.**

Below is an example of how you can implement error handling middleware in Express.js and add validation to your API routes using a popular validation library called express-validator. Make sure to install the required packages before proceeding.

Step 1: Install Required Packages

npm install express-validator

**Step 2: Modify app.js**

**Update your app.js file to include error handling middleware and validation for your API routes.**

| const express = require('express');  const mongoose = require('mongoose');  const bodyParser = require('body-parser');  const { body, validationResult } = require('express-validator');  const app = express();  const port = 3000;  // Connect to MongoDB  mongoose.connect('mongodb://localhost:27017/todo-app', { useNewUrlParser: true, useUnifiedTopology: true });  // Check MongoDB connection  const db = mongoose.connection;  db.on('error', console.error.bind(console, 'MongoDB connection error:'));  db.once('open', () => {  console.log('Connected to MongoDB');  });  // Use body-parser middleware for parsing JSON  app.use(bodyParser.json());  // Define a simple data model for a ToDo item  const todoSchema = new mongoose.Schema({  title: { type: String, required: true },  description: { type: String },  completed: { type: Boolean, default: false },  });  const Todo = mongoose.model('Todo', todoSchema);  // Validation middleware for POST and PUT requests  const validateTodo = [  body('title').notEmpty().withMessage('Title cannot be empty'),  body('completed').isBoolean().withMessage('Completed must be a boolean'),  // Add more validation rules as needed  ];  // Error handling middleware  app.use((err, req, res, next) => {  console.error(err.stack);  res.status(500).send('Something went wrong!');  });  // Define CRUD routes  // GET all todos  app.get('/todos', async (req, res) => {  try {  const todos = await Todo.find();  res.json(todos);  } catch (error) {  res.status(500).json({ message: error.message });  }  });  // GET a specific todo by ID  app.get('/todos/:id', async (req, res) => {  try {  const todo = await Todo.findById(req.params.id);  if (todo) {  res.json(todo);  } else {  res.status(404).json({ message: 'Todo not found' });  }  } catch (error) {  res.status(500).json({ message: error.message });  }  });  // POST a new todo with validation  app.post('/todos', validateTodo, async (req, res) => {  const errors = validationResult(req);  if (!errors.isEmpty()) {  return res.status(400).json({ errors: errors.array() });  }  const todo = new Todo({  title: req.body.title,  description: req.body.description,  completed: req.body.completed || false,  });  try {  const newTodo = await todo.save();  res.status(201).json(newTodo);  } catch (error) {  res.status(400).json({ message: error.message });  }  });  // PUT/update a specific todo by ID with validation  app.put('/todos/:id', validateTodo, async (req, res) => {  const errors = validationResult(req);  if (!errors.isEmpty()) {  return res.status(400).json({ errors: errors.array() });  }  try {  const updatedTodo = await Todo.findByIdAndUpdate(  req.params.id,  {  title: req.body.title,  description: req.body.description,  completed: req.body.completed || false,  },  { new: true }  );  if (updatedTodo) {  res.json(updatedTodo);  } else {  res.status(404).json({ message: 'Todo not found' });  }  } catch (error) {  res.status(400).json({ message: error.message });  }  });  // DELETE a specific todo by ID  app.delete('/todos/:id', async (req, res) => {  try {  const deletedTodo = await Todo.findByIdAndDelete(req.params.id);  if (deletedTodo) {  res.json({ message: 'Todo deleted successfully' });  } else {  res.status(404).json({ message: 'Todo not found' });  }  } catch (error) {  res.status(500).json({ message: error.message });  }  });  // Start the server  app.listen(port, () => {  console.log(`Server listening at http://localhost:${port}`);  }); |
| --- |

Step 3: Test Validation and Error Handling

Test Validation:

Use Postman to send invalid requests (e.g., missing title or completed as a non-boolean) to the POST and PUT routes.

Verify that you receive validation error messages.

Test Error Handling:

Introduce intentional errors in your code (e.g., force a database connection error).

Send requests to your routes and verify that you receive a 500 status and an error message.

These modifications ensure that your API routes have validation for input data and that errors are handled gracefully with appropriate error messages. Adjust the validation rules as needed for your application's requirements.