Name:Chethan Kumar K M
SRN :PES1PG23CA329

Lab Exercises -3

1. **Building a Blog API**
   You are tasked with creating a basic blogging platform API using Express.js. Implement the following:

   - A route to fetch all blog posts (GET /api/posts).
   - A route to fetch a single blog post by its ID (GET /api/posts/:id).
   - A route to create a new blog post (POST /api/posts) with title and content in the request body.
   - A route to update a blog post by ID (PUT /api/posts/:id).
   - A route to delete a blog post by ID (DELETE /api/posts/:id).

Ensure that:

- The API returns appropriate HTTP status codes (e.g., 200, 201, 404).
- Error handling is implemented for invalid IDs or missing data in the request body.
- Use middleware to log all incoming requests with their method and URL.

```
const express = require('express');

const app = express();


app.use(express.json());


// In-memory storage for demonstration

let posts = [

  { id: 1, title: 'First Post', content: 'This is the first blog post.' },

  { id: 2, title: 'Second Post', content: 'This is another blog post.' }

];
```

```
// Middleware for logging
const requestLogger = (req, res, next) => {
        console.log(`[${new    Date().toISOString()}]    ${req.method}
   ${req.url}`);
  next();
};

// Apply the logging middleware globally
app.use(requestLogger);

// GET /api/posts - Fetch all posts
app.get('/api/posts', (req, res) => {
  res.status(200).json(posts);
});

// GET /api/posts/:id - Fetch a single post by ID
app.get('/api/posts/:id', (req, res) => {
  const post = posts.find(p => p.id === parseInt(req.params.id));
  if (post) {
    res.status(200).json(post);
  } else {
    res.status(404).json({ error: "Post not found" });
  }
});

// POST /api/posts - Create a new post
app.post('/api/posts', (req, res) => {
  const { title, content } = req.body;
  if (!title || !content) {
      return res.status(400).json({ error: "Title and content are required" });
  }

  const newPost = { id: posts.length + 1, title, content };
  posts.push(newPost);
  res.status(201).json(newPost);
});

// PUT /api/posts/:id - Update a post by ID
app.put('/api/posts/:id', (req, res) => {
  const { title, content } = req.body;
      const postIndex = posts.findIndex(p => p.id === parseInt(req.params.id));

  if (postIndex === -1) {
    return res.status(404).json({ error: "Post not found" });
  }

  if (!title || !content) {
      return res.status(400).json({ error: "Title and content are required" });
  }

  posts[postIndex] = { ...posts[postIndex], title, content };
  res.status(200).json(posts[postIndex]);
});
```

```
// DELETE /api/posts/:id - Delete a post by ID
app.delete('/api/posts/:id', (req, res) => {
    const postIndex = posts.findIndex(p => p.id === parseInt(req.params.id));

  if (postIndex === -1) {
    return res.status(404).json({ error: "Post not found" });
  }

  // Remove the post from the array
  const deletedPost = posts.splice(postIndex, 1)[0];
  res.status(200).json({ message: "Post deleted", post: deletedPost });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

OUTPUT:

```
[
 {
  "id": 1,
  "title": "First Post",
  "content": "This is the first blog post."
 },
 {
  "id": 2,
  "title": "Second Post",
  "content": "This is another blog post."
 }
]
```

2. **E-commerce Product Management**
   You are developing an API for managing products in an e-commerce application. Implement the following features:
   - A route to list all products (GET /products). Support query parameters like ?category=electronics to filter products by category.
   - A route to fetch details of a specific product by its ID (GET /products/:id).
   - A route to add a new product (POST /products) with fields like name, price, and category in the request body. Validate that all required fields are provided.
   - Use middleware to check if the Content-Type header is set to application/json for POST requests. If not, return an error response.

```
const express = require('express');

const app = express();


// Middleware to parse JSON body

app.use(express.json());


// In-memory storage for demonstration

let products = [

  { id: 1, name: "Laptop", price: 999.99, category: "electronics" },

  { id: 2, name: "Book", price: 20.00, category: "books" }

];
```

```
// Middleware to check Content-Type for POST requests
const checkContentType = (req, res, next) => {
  if (req.method === 'POST' && req.headers['content-type'] !== 'application/json') {
    return res.status(415).json({ error: "Content-Type must be application/json" });
  }
  next();
};

// Apply Content-Type middleware to POST /products
app.use('/products', checkContentType);

// GET /products - List all products with optional category filter
app.get('/products', (req, res) => {
  const category = req.query.category;
  let filteredProducts = products;

  if (category) {
    filteredProducts = products.filter(product => product.category === category);
  }

  res.json(filteredProducts);
});

// GET /products/:id - Fetch details of a specific product by ID
app.get('/products/:id', (req, res) => {
  const productId = parseInt(req.params.id);
  const product = products.find(p => p.id === productId);

  if (product) {
    res.json(product);
  } else {
    res.status(404).json({ error: "Product not found" });
  }
});

// POST /products - Add a new product
app.post('/products', (req, res) => {
  const { name, price, category } = req.body;

  if (!name || !price || !category) {
    return res.status(400).json({ error: "Name, price, and category
 are required" });
  }

  const newProduct = {
    id: products.length + 1, // Simple ID generation for
 demonstration
    name,
    price,
    category
  };

  products.push(newProduct);
```

```
     res.status(201).json(newProduct);
   });


   const PORT = process.env.PORT || 3000;
   app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

output

```
[
  {
    "id": 1,
    "name": "Laptop",
    "price": 999.99,
    "category": "electronics"
  },
  {
    "id": 2,
    "name": "Book",
    "price": 20,
    "category": "books"
  }
]
```

3. **User Authentication System**
   Build an Express.js-based API for user authentication with the following routes:
   - POST /register: Accepts user details (name, email, password) in the request body and registers a new user. Validate that all fields are provided and return an error if any field is missing.
   - POST /login: Accepts email and password in the request body and checks if they match a registered user. Return success or failure accordingly.
   - GET /profile: Returns the profile of the logged-in user. Use middleware to simulate authentication by checking if a valid token is passed in the headers (e.g., Authorization: Bearer <token>). Return an error if no token is provided.

```
const express = require('express');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const app = express();


app.use(express.json());


// Simple in-memory storage for demonstration

let users = [];


// Secret key for JWT - in production, keep this in environment variables

const JWT_SECRET = 'your-secret-key';


// Middleware for authentication

const authenticate = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (token == null) return res.sendStatus(401);

  jwt.verify(token, JWT_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
};

// Register route
app.post('/register', async (req, res) => {
  const { name, email, password } = req.body;

  if (!name || !email || !password) {
    return res.status(400).json({ error: "All fields are required" });
  }

  // Check if user exists
  if (users.find(user => user.email === email)) {
    return res.status(400).json({ error: "Email already exists" });
  }

  const hashedPassword = await bcrypt.hash(password, 10);
  const user = { name, email, password: hashedPassword };
  users.push(user);
  res.status(201).json({ message: "User registered successfully" });
});

// Login route
app.post('/login', async (req, res) => {
  const { email, password } = req.body;

  const user = users.find(user => user.email === email);
  if (!user) {
    return res.status(400).json({ error: "User not found" });
  }

  const validPassword = await bcrypt.compare(password,
   user.password);
  if (!validPassword) {
    return res.status(400).json({ error: "Invalid password" });
  }

  const token = jwt.sign({ email: user.email }, JWT_SECRET);
  res.json({ token });
});

// Profile route
app.get('/profile', authenticate, (req, res) => {
  const user = users.find(u => u.email === req.user.email);
  if (!user) {
    return res.status(404).json({ error: "User not found" });
    }
    // Here you might want to sanitize or limit information sent back
    res.json({ name: user.name, email: user.email });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

4. **Error Handling in an Order Management System**
   Create an API for managing customer orders with the following routes:

   GET /orders: Fetches all orders. If no orders exist, return a message saying "No orders found."
   POST /orders: Creates a new order with details like customer name, product ID, and quantity in the request body. Validate that all fields are provided; otherwise, return an error response with status code 400.

   Add global error-handling middleware that catches unhandled errors and returns a JSON response with status code 500 and an error message like "Internal Server Error."

```
const express = require('express');
const app = express();

app.use(express.json());

// In-memory storage for demonstration
let orders = [];

// Route to fetch all orders
app.get('/orders', (req, res) => {
  if (orders.length === 0) {
    return res.status(200).json({ message: "No orders found." });
  }
  res.json(orders);
});

// Route to create a new order
app.post('/orders', (req, res) => {
  const { customerName, productId, quantity } = req.body;

  if (!customerName || !productId || !quantity) {
    return res.status(400).json({ error: "All fields are required" });
  }

  const newOrder = {
    id: orders.length + 1,
    customerName,
    productId,
    quantity
  };

  orders.push(newOrder);
  res.status(201).json(newOrder);
});

// Global error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
```

```
    res.status(500).json({ error: "Internal Server Error" });
});

// Server setup
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

5. **Nested Routes for Library Management**
   You are designing an API for managing books and their authors in a library system. Implement nested routes using express.Router() as follows:

- /authors (GET): Lists all authors.
- /authors/:authorId/books (GET): Lists all books written by a specific author based on their ID.
- /authors/:authorId/books/:bookId (GET): Fetches details of a specific book written by the author.

Requirements:

- Use modular routing (express.Router()) for authors and books routes separately and mount them in your main application file using app.use().
- Implement middleware that logs the current timestamp whenever any of these routes are accessed.

App.js

```
const express = require('express');

const app = express();


// Middleware for logging timestamp

const logTimestamp = (req, res, next) => {

  console.log(`[${new Date().toISOString()}] Request to ${req.method} ${req.originalUrl}`);

  next();

};


app.use(express.json());

app.use(logTimestamp);


// Importing route handlers

const authorsRouter = require('./routes/authors');

app.use('/authors', authorsRouter);


const PORT = process.env.PORT || 3000;
```

```
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

routes/authores.js

```
const express = require('express');
const router = express.Router();

// In-memory storage for demonstration
let authors = [
  { id: 1, name: 'J.K. Rowling', books: [{ id: 1, title: 'Harry Potter' }] },
  { id: 2, name: 'George Orwell', books: [{ id: 2, title: '1984' }] }
];

// GET /authors - List all authors
router.get('/', (req, res) => {
  res.json(authors);
});

// GET /authors/:authorId/books - List all books by a specific author
router.get('/:authorId/books', (req, res) => {
  const author = authors.find(a => a.id ===
   parseInt(req.params.authorId));
  if (author) {
    res.json(author.books);
  } else {
    res.status(404).json({ error: "Author not found" });
  }
});

// GET /authors/:authorId/books/:bookId - Fetch details of a specific
```

```
  book
router.get('/:authorId/books/:bookId', (req, res) => {
  const author = authors.find(a => a.id ===
  parseInt(req.params.authorId));
  if (author) {
    const book = author.books.find(b => b.id ===
  parseInt(req.params.bookId));
    if (book) {
      res.json(book);
    } else {
      res.status(404).json({ error: "Book not found" });
    }
  } else {
    res.status(404).json({ error: "Author not found" });
  }
});

module.exports = router;
```

6. **RESTful API for Task Management**
   Develop a task management API where users can manage their daily tasks with the following features:

   GET /tasks: Fetches all tasks with support for filtering tasks based on their status using query parameters (e.g., ?status=completed).

   POST /tasks: Adds a new task with fields like title, description, and status (default status should be "pending"). Validate that title and description are provided; otherwise, return an error response.

   PATCH /tasks/:id: Updates the status of a task by its ID (e.g., change from "pending" to "completed"). Return an error if the task ID does not exist.

```
const express = require('express');
const app = express();

app.use(express.json());

// In-memory storage for demonstration
let tasks = [
  { id: 1, title: 'Learn Express', description: 'Study Express.js', status: 'completed' },
  { id: 2, title: 'Write Code', description: 'Implement task API', status: 'pending' }
];

// GET /tasks - Fetch all tasks with optional status filter
app.get('/tasks', (req, res) => {
  const status = req.query.status;
  let filteredTasks = tasks;

  if (status) {
    filteredTasks = tasks.filter(task => task.status === status);
  }

  res.json(filteredTasks);
});

// POST /tasks - Add a new task
app.post('/tasks', (req, res) => {
  const { title, description } = req.body;

  if (!title || !description) {
        return res.status(400).json({ error: "Title and description are required" });
    }

    const newTask = {
        id: tasks.length + 1,  // Simple ID generation for demonstration
        title,
        description,
        status: 'pending'
    };

    tasks.push(newTask);
    res.status(201).json(newTask);
});

// PATCH /tasks/:id - Update task status
app.patch('/tasks/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { status } = req.body;

    const task = tasks.find(task => task.id === id);

    if (!task) {
        return res.status(404).json({ error: "Task not found" });
    }

    if (status) {
        task.status = status;
        res.json({ message: "Task status updated", task });
    } else {
        res.status(400).json({ error: "Status is required for update" });
    }
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

7. **Middleware for Logging and Authentication**
   Create an Express.js application where you implement middleware for logging and authentication:
   - **Middleware 1**: Logs every incoming request's method, URL, and timestamp before passing control to the next middleware or route handler. This should apply globally across all routes.
   - **Middleware 2:** Checks if an API key is passed as part of query parameters (e.g., /api/resource?apiKey=12345). If no API key is provided or it's invalid, return an error response with status code 401 ("Unauthorized"). Apply this middleware only to routes under /api.

```
const express = require('express');
const app = express();

app.use(express.json());

// Middleware 1: Logging middleware (global)
const loggingMiddleware = (req, res, next) => {
    const timestamp = new Date().toISOString();
    console.log(`[${timestamp}] ${req.method} ${req.url}`);
    next();
};

// Middleware 2: Authentication middleware (only for /api routes)
```

```
const authMiddleware = (req, res, next) => {
   const apiKey = req.query.apiKey;

   if (!apiKey || apiKey !== '12345') { // Here, '12345' is a placeholder for a valid API key
      return res.status(401).json({ error: "Unauthorized - API key missing or invalid" });
   }
   next();
};

// Apply global logging middleware
app.use(loggingMiddleware);

// Routes under /api should be protected by authMiddleware
app.use('/api', authMiddleware, express.Router()
   .get('/resource', (req, res) => {
      res.json({ message: "This is a protected resource!" });
   })
   .post('/another', (req, res) => {
      res.json({ message: "POST to protected route successful" });
   })
);

// Example of a public route not requiring authentication
app.get('/', (req, res) => {
   res.json({ message: "Public route - no API key needed" });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```