**Exercise 1 – Expressjs**

**(http Methods)**

1. Develop a basic Express.js application that simulates a simple user management system. The application should define a /users route that allows interaction through different HTTP methods. Begin by initializing an in-memory array to hold user objects, where each object includes three properties: id, name, and email. Implement functionality to return all users using the GET method on /users. Allow new users to be added via the POST method to the same route. Each new user should receive a unique ID that increments sequentially. Additionally, implement the DELETE method on /users/:id to remove a specific user by ID. Ensure that the application uses express.json() middleware to parse incoming JSON request bodies, and return appropriate JSON responses for each action. Include basic error handling to display a clear message when attempting to delete a user that does not exist.
2. Create an Express.js application that manages a catalog of products. Start with a predefined array of products, each having the properties id, name, price, and stock. Implement a PUT route at /products/:id that updates an existing product’s details. The request body may contain any combination of the three updatable fields: name, price, and stock. Your application should update only the fields provided in the request and leave the rest unchanged. If a product with the given ID is not found, return a 404 status with a message indicating that the product does not exist. Ensure the use of express.json() middleware to process JSON bodies and return structured JSON responses that confirm the update and show the latest state of the product. Keep the application implementation in a single JavaScript file, without using a database or external files.
3. Design and implement an Express.js application to manage a list of books. Each book must have an id, title, and author. Initialize the application with at least two predefined books stored in an in-memory array. The application must support the following functionalities:
   1. The GET /books route should return the list of all books.
   2. The POST /books route should accept a new book's title and author in the request body, assign it a unique id, add it to the array, and return a confirmation message.
   3. The PUT /books/:id route should update an existing book’s title and/or author based on the id provided in the URL. Only the fields present in the request body should be updated. If the book is not found, return a 404 status with an appropriate message.
   4. The DELETE /books/:id route should remove a book by its id. If the book does not exist, return a 404 error. Otherwise, return a success message indicating which book was deleted.
   5. Ensure proper use of express.json() middleware and return all responses in plain text format using res.send().
4. Develop an Express.js application that provides basic functionality for managing student records. Each student should have an id, name, and course. Initialize the application with at least three student records in an in-memory array. The application must implement the following:
   1. The route GET /students should return a list of all students.
   2. The route GET /students/:id should return details of a single student based on the ID provided in the URL.
   3. The route POST /students should accept a new student's name and course via the request body, assign a unique ID, add the student to the array, and return a message confirming the addition.
   4. The route PUT /students/:id should update the student’s name and/or course for the given ID. If the student does not exist, the server should return a 404 error.
   5. The route DELETE /students/:id should remove a student from the list and return the details of the deleted student.
   6. Use express.json() middleware to parse JSON input, and use res.send() for all output. Ensure appropriate status codes and messages are used for both success and error responses.
5. Create an Express.js application to simulate a simple task management system. Each task should have an id, title, description, and a status field (e.g., “pending”, “in-progress”, “completed”). Initialize the application with at least two tasks in an in-memory array.
   1. The application must implement the following functionality using appropriate HTTP methods:
   2. GET /tasks: Return the list of all tasks.
   3. GET /tasks/:id: Return details of a task by its ID.
   4. POST /tasks: Accept a new task's title and description from the request body, assign a unique ID, default the status to “pending”, and add it to the array.
   5. PUT /tasks/:id: Allow the user to update any of the task fields (title, description, or status). Only the provided fields should be updated.
   6. DELETE /tasks/:id: Remove a task by its ID. If the task doesn't exist, return a 404 error.
   7. Ensure use of express.json() middleware for parsing input, and use res.send() for all output messages.
6. Develop an Express.js application that manages a library of movies. Each movie should include an id, title, genre, and rating (1–10). Use an in-memory array initialized with at least three sample movies.
   1. Implement the following routes using the appropriate HTTP methods:
   2. GET /movies: Retrieve all movie records.
   3. GET /movies/:id: Retrieve details of a specific movie.
   4. POST /movies: Add a new movie to the array with data from the request body.
   5. PUT /movies/:id: Update one or more fields (title, genre, rating) of a specific movie.
   6. DELETE /movies/:id: Remove the movie from the array by its ID. Return a clear message even if the movie doesn’t exist.
   7. Ensure all inputs are JSON and responses are returned in text using res.send(). Include error handling for invalid movie IDs.