Activity: Building a Basic Blog API with Express.js

Upload a JavaScript file named USPnumber1.USPnumber2.js. If you need to upload a file after the deadline use late.USPnumber1.USPnumber2.js

Instructions:

Initialize a new Node.js project and install Express.js.

In your main server file, set up a basic Express.js application. Ensure that your server is listening on a port of your choice.

Create a new JavaScript object to act as your database. This should contain an array of "posts". Each post should be an object with properties for "id", "title", "content", and "author".

Set up a middleware function that logs (console.log) the current date and request method to the console before each route. This will be useful for debugging purposes.

Set up the following routes:

GET /posts: This should return a list of all posts in your database object.
GET /posts/:id: This should return the post with the specified ID. If no such post exists, return an appropriate error message.
POST /posts: This should accept JSON input in the form of a new post, which should include properties for "title", "content", and "author". Add the new post to your database object and return the updated list of posts.
PUT /posts/:id: This should accept JSON input in the form of an updated post. Replace the post with the specified ID in your database object and return the updated list of posts.
DELETE /posts/:id: This should delete the post with the specified ID from your database object and return the updated list of posts.
Test your application using Insomnia, Postman, or another API testing tool.

Remember: Each route should send a response back to the client and handle any errors appropriately. You should also use middleware to parse incoming JSON.

Tips:

Follow the video course up to Lesson 10 and you will end up with a running web server. In lessons 11 and 12, the code is refactored to be more organized.

Lesson 10 explains the REST actions (GET, PUT, POST, and DELETE) and shows how to test the requests using the Postman tool (install in lesson 3).

After lesson 10, you have a running server that has GET, PUT, and DELETE services.

Good luck! Remember to take your time, think through the problem, and test your code often as you're working.

Annex: Express Example

```
// Importing the express module
const express = require('express');

// Creating an instance of express to setup the server

const app = express();
// Implementing middleware using the 'use' method. Middleware are functions that have access to the request and response objects,
// and can execute any code or make changes to the request and response objects.
// This particular middleware logs the request method (GET, POST, PUT, DELETE, etc.) and the request URL to the console
// 'next' is a function that, when called, executes the next middleware in the stack.
app.use((req, res, next) => {
    console.log(`Request Method: ${req.method}`);
    console.log(`Request URL: ${req.url}`);
    next();
});

// Setting up a GET route at the path '/'. When a GET request is made to this path,
// the server sends back the string 'Home Page' as a response.
app.get('/', (req, res) => {
    res.send('Home Page');
});

// Setting up a POST route at the path '/data'. When a POST request is made to this path,
// the server sends back the string 'Data received' as a response.
app.post('/data', (req, res) => {
    res.send('Data received');
});

// Setting up a PUT route at the path '/data'. When a PUT request is made to this path,
// the server sends back the string 'Data updated' as a response.
app.put('/data', (req, res) => {
    res.send('Data updated');
});

// Setting up a DELETE route at the path '/data'. When a DELETE request is made to this path,
// the server sends back the string 'Data deleted' as a response.
app.delete('/data', (req, res) => {
    res.send('Data deleted');
});
```

```
// Telling the server to listen for incoming requests on port 3000 and log to the console when it starts successfully
app.listen(3000, () => {
    console.log('Server is listening on port 3000');
});
```