# Express.js lab

**Setting up a new Node project and installing Express.js (5 minutes)**

* Create a new empty folder for this project
  + Make sure you are storing all your web dev projects in an easy-to-find and organized place on your computer.
  + Make sure you open this folder in VS Code.
  + Open a terminal only in VS code. Make sure it’s a “bash” terminal and not something like Powershell or something else.
  + Run an “ls” command to make sure you’re in your new directory.
* Let's create a new project directory and initialize it with npm with defaults accepted. This will create a package.json file.

mkdir express-demo

cd express-demo

npm init -y

* Now, let's install Express.js."

npm install express

* Add a .env file to the root of your project. Create a new file and name it “.env”. This is where you store global definitions, keys, and variables for you project. Open the .env file and add the following. The port can be 3001, 8000, or 8080 for example. Just don’t use 3000 or 5000.

PORT = 3001

* Create a server.js file in your root directory
* Your project structure should look like this:

express-demo/

├── node\_modules/

├── [.env]

├── [package.json]

├── [package-lock.json]

└── [server.js]

**Creating a Simple Server (10 minutes)**

* Open a new file named server.js and add the following code to import express into your server file *with Copilot turned on*:"

import express from 'express';

* Observe if Copilot starts to add in some form of the remaining code automatically. Make sure the that finished product is this and nothing extra. Sometimes Copilot will make incorrect assumptions

// import dependencies

import express from 'express';

//import port from .env file or default to 8080

const port = process.env.PORT || 8080;

// create an express app

const app = express();

// Set server to listen on port and console log it is running

app.listen(port, () => {

  console.log(`Server is running on http://localhost:${port}`);

});

//export app

export default app;

* Install nodemon in your root folder

npm i nodemon

* Update dev script in package.json file so server auto restarts with each save and pulls in PORT from .env file. Add the highlighted line to your scripts section under “test” line

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

,"dev": "nodemon --env-file=.env server.js"

  },

* Update package.json to also include type of module so you can use new import syntax instead of old syntax. See highlighted line below

"main": "index.js",

  "type": "module",

  "scripts": {

* Run the server using the following command:

npm run dev

* Open your browser and navigate to http://localhost:3001 to see what it says
* Add this command to the server after you create the express app but before the app.listen command

app.get('/', (req, res) => {

  res.send('Hello, World!');

});

* Save and see if you now get the message 'Hello, World!'". If your port number is not 3001, then alter your URL or server code as needed to open the proper port
* Recall how in slides where we compared this express code to the code using core Node modules http and url. Hopefully you see Express as simpler and more compelling to use

**Middleware in Express.js (10 minutes)**

* Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle.
* Let's create a custom middleware function to log the request method and URL:
* NOTE: This goes right after your **const app = express();**

//Middleware to parse JSON bodies in requests'

app.use(express.json());

// Middleware to log various examples of request properties

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

  console.log(`${req.method} `);

  next();

});

* Try updating the middleware log code to show more by adding this into the code

console.log(`${req.method} ${req.url}  `);

console.log(`${req.method} request made at ${req.url} -- Body: ${JSON.stringify(req.body)} `);

**Routing in Express.js (10 minutes)**

* Routing is a way to determine how an application responds to a client request to a particular endpoint.
* Let's add more routes to our server.js file:

app.get('/about', (req, res) => {

  res.send('About Page');

});

app.post('/submit', (req, res) => {

  res.send('Form Submitted');

});

app.put('/update', (req, res) => {

  res.send('Update Successful');

});

app.delete('/delete', (req, res) => {

  res.send('Delete Successful');

});

* Test these routes using Postman or a similar tool."

**Building a RESTful API in Express.js (15 minutes)**

Let's build a simple RESTful API for managing a simple list of employees.

Create an array to store employees and add routes for CRUD operations one at a time

|  |
| --- |
| **let** employeesList = [];  app.get('/api/employees', (req, res) => {  res.json(employeesList);  });  app.post('/api/employees', (req, res) => {  **const** { employee\_id, name, email } = req.body;  // Ensure employee\_id is stored as a string  **const** id = employee\_id ? String(employee\_id) : employee\_id;  **const** newEmployee = { employee\_id: id, name, email };  employeesList.push(newEmployee);  res.status(201).json(newEmployee);  });  app.put('/api/employees/:employee\_id', (req, res) => {    const { employee\_id } = req.params;    const { name, email } = req.body;    // Coerce types to string for a reliable comparison    const targetId = String(employee\_id);    const updatedEmployee = { employee\_id: targetId, name, email };      employeesList = employeesList.map(employee =>      String(employee.employee\_id) === targetId ? updatedEmployee : employee    );    res.json(updatedEmployee);  });  app.delete('/api/employees/:employee\_id', (req, res) => {    const { employee\_id } = req.params;    const targetId = String(employee\_id);    const originalLength = employeesList.length;    employeesList = employeesList.filter(employee => String(employee.employee\_id) !== targetId);    if (employeesList.length === originalLength) {      return res.status(404).json({ error: 'Employee not found' });    }    res.status(204).send();  }); |
|  |

As you add routes into the code, save it and test each one works with Postman start with GET, POST, PUT, and then DELETE   
  
Here’s some test data to put into the body when you test POST

**{"employee\_id":1, "name": "Clint", "email":"clint@mail.com"}**

**Error Handling and Debugging (5-10 minutes)**

Error handling is crucial for building robust applications. Let's add an error-handling middleware to our server.js file:

//Error-handling middleware should come after all your route handlers and other middleware. This ensures it catches any errors thrown during request processing.

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

  console.error(err.stack);

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

});

//404 handler should be the last middleware in your file. It only runs if no other route matched the request.

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

  res.status(404).json({ error: 'Route not found' });

});

**Q&A + Discussion (10-15 minutes)**

* Let's open the floor for questions
* Discussion ideas for Express.js
  + What are the key advantages of using Express.js for web application development?
  + How does Express.js simplify the process of creating a web server compared to using plain Node.js?
  + Explain the concept of middleware in Express.js and provide an example of how it can be useful.
  + What is the difference between app.use() and app.get() in Express.js?
  + How does Express.js handle different HTTP methods, and why is this important for building RESTful APIs?
* Discussion on what’s next?
  + How might our current implementation of storing data in a JavaScript array be limiting for a real-world application? What is our likely solution for this?
  + How do you think we might need to modify our current Express.js routes to work with a MySQL database instead of an array?
  + What challenges do you anticipate when integrating a MySQL database with our Express.js application?
  + How could using a database impact the performance of our API, both positively and negatively?
  + In what ways might our error handling need to change when working with a database?
  + How could we ensure data consistency and integrity when using MySQL with Express.js?
  + What security considerations should we keep in mind when connecting our Express.js application to a MySQL database?
  + How might the concept of database transactions relate to our RESTful API operations?
  + Can you think of any scenarios where we might want to use both in-memory storage and a MySQL database in the same application?