Express

1. Install Nodejs, postman, mongodb login

API

1. What do we understand by API
    a. Application programming interface
    b. An API, or Application Programming Interface, is like a menu in a restaurant. The menu provides a list of dishes you can order, along with a description of each dish. When you specify which dish you want, the kitchen (the system) prepares the dish and serves it to you (the user).
    c. Interface to use application’s features via programming

Node

Node.js is a runtime environment that allows you to run JavaScript on the server-side, outside of a web browser. It's built on Chrome's V8 JavaScript engine.

Express

Why Express ?

Express.js is a minimalist web framework for Node.js. While you can certainly build web servers and applications using just the built-in HTTP module in Node.js, Express.js provides a higher-level way to to do certain tasks like manage routes, requests, and views

some benefits and reasons why developers often choose Express.js over vanilla Node.js for web development:

1. Simpler and cleaner syntax
2. Code readability
3. Routing: Express provides a straightforward way to define routes and handle different HTTP methods (GET, POST, PUT, DELETE, etc.). In vanilla Node.js, you'd typically have to handle URLs and methods manually, which can be cumbersome.
4. Middleware: One of the core features of Express is its use of middleware, which 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. .
5. Request handling / routing much more scalable and manageable
6. Lot of heavy lifting of logic and features are done by express

Initializing project

1. Create a file index.js
2. Npm init -y
3. Npm i express
4. Npm i nodemon

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

const app = express();

app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

5. Ports
    a. What is a Port Number?
    b. In the context of computer networking, a port number is a way to identify a specific process or service within a device that uses the Internet Protocol (IP).

    c. Why the concept of ports was invented if it is not a physical thing?
    d. Think of ports as a mechanism for multitasking. While an IP address identifies a machine, many different applications and services run on that machine at once. Each of these services needs a way to receive the

appropriate data packets without confusion. Ports allow a single machine with one IP address to efficiently manage multiple services simultaneously. By assigning different services to different ports, a computer can easily determine which application should handle incoming data.

e.

6. Execute the code
    a. In the script in package. Json, add start -> node index.js and add another script -> dev : nodemon index.js
    b. Run -> node index.js
    c. Then add scripts for node and nodemon
7. App.use and Postman
    a. this app.use statement to run this code for every request

```
app.use(function (req, res) {
  res.status(201).send("Hello World");
});
```

    b. Send request from postman for any route and see the response

CRUD and HTTP methods

1. Create - post
2. Read - get
3. Update - patch
4. Delete - delete
5. Difference between put and patch

a. PUT: This method is used when you want to update a complete resource. When you make a PUT request, you provide a complete updated object. The server then replaces the existing resource with the provided object. If certain attributes are missing in the request, those are typically set to their default values or removed. Essentially, a PUT request entirely replaces the existing resource with a new version.
b.
c. Use PUT when you have the complete updated state of the resource.
d. PUT is idempotent, meaning that making the same request multiple times will result in the same state of the resource on the server.
e. PATCH: On the other hand, PATCH is used for partial updates. With a PATCH request, you only need to provide the specific changes to the resource, not the complete resource. The server then applies these changes to the existing resource.
f.
g. Use PATCH for updating parts of the resource or for situations where sending the complete resource is not feasible or necessary.
h. PATCH can be idempotent, but it's not a requirement. It depends on how the server processes the PATCH request.

Route Matching with their route handlers

```
app.get("/api/user", (req, res) => {
  res.json({
    status: 200,
    data: {
      name: "John Doe",
      age: 30,
    },
  });
});
```

1. Use the above code before the app.use code

```
app.post("/api/user", (req, res) => {
  console.log(req.body);
  res.json({
    status: 200,
    data: req.body,
  });
});
```

For the above post method , note that nothing logs while sending request from postman

to allow server to parse the user sent json, we need to add something like - app.use(express.json());

An important use case of app.use also is to have something like a 404 page. When no route matched then at the end have this saying page not found

Custom MIddleware

1. We saw function callbacks with req, res as parameters.
2. For middleware, we have a third next method which is called to invoke the next middleware in the code.
3. For built in middleware, the next is called internally
4. But if we are building a custom middleware, then we need to call the next middleware

```
app.use((req, res, next) => {
    console.log(`${req.method} request to ${req.path}`)
  next();  // This is required to move to the next
middleware/route
});
```

5. When you call next() inside this middleware, you're telling Express to proceed to the next handler in the line for the current request.This "next handler" could be another middleware or a route handler
    a. So, in simple terms, middleware functions are the individual "checkpoints" in your application that requests go through before reaching their final destination.
6. Order matters in middleware

7. Types of middleware
    a. app.use(...) -> global middleware
    b. app.get(path, handler).post(...) -> path specific middleware
8. Benefits
    a. Error handling
    b. 404
    c. Logger fn
    d. Json to js objects -> app.use(express.json)
    e. Another very important use case is for authentication and protecting routes. We can have an app.use(some handler to validate the user) and then only pass the control to the next handler in the code
9. Without the next, the request response cycle will be stuck

Project Building

1. Get => /api/user -> get all the users
2. Post => /api/user -> add a user
3. Create another app.js file
4. We will use the data.json file as the data source. Later we will switch to database
5. Starter code

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

```
require("dotenv").config();

const port = process.env.PORT || 3000;
const fs = require("fs");
const data = fs.readFileSync("./data.json", "utf8");
const userData = JSON.parse(data);

app.use(express.json());
app.get("/api/user", (req, res) => {
  res.json({
    status: 200,
    data: {
      name: "John Doe",
      age: 25,
    },
  });
});

app.post("/api/user", (req, res) => {
  console.log(req.body);
  res.json({
    status: 200,
    data: req.body,
  });
});
```

```
// app.use(function (req, res) {
//   res.status(201).send("Hello World");
// });

app.use(function (req, res) {
  res.status(404).send("Sorry can't find that!");
});

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

a.

Get user

```
app.get("/api/user", (req, res) => {
  let msg = "";
  if (userData.length === 0) {
    msg = "No user found";
  } else {
    msg = "Data found";
  }
  res.json({
    status: 200,
```

```
    data: userData,
    message: msg,
  });
});
```

Try catch

Put code in try catch block

```
app.get("/api/user", (req, res) => {
  try {
    let msg = "";
    if (userData.length === 0) {
      msg = "No user found";
      // throw new Error('No data found')
    } else {
      msg = "Data found";
    }
    res.json({
      status: 200,
      data: userData,
      message: msg,
    });
  } catch (err) {
    res.status(500).json({
      message: err.message,
    });
```

```
  }
});
```

POST USER

1. To add a user we need an id
2. Npm we have a package to generate an id -
   ```
   const short = require("short-uuid");
   ```
3.
   ```
   app.post("/api/user", (req, res) => {
    console.log(req.body);
    const id = short.generate();
    const userDetails = req.body;
    userDetails.id = id;
    console.log(userDetails);
    userData.push(userDetails);
    // write to file
    fs.writeFile("./data.json", JSON.stringify(userData), (err) => {
     if (err) {
      console.log(err);
     }
    });

    res.json({
     status: 200,
   ```

```
    data: req.body,
    message: `User created with id ${id}`,
  });
});
```

4. Let us add a small validation to check if the user has sent an empty object in the post method.
    a.

```
app.post("/api/user", (req, res) => {
 const userInput = req.body;
 const isEmpty = Object.keys(userInput).length === 0;
 if (isEmpty) {
   return res.status(400).json({
     status: 400,
     message: "Body cannot be empty",
   });
 } else {
   const id = short.generate();
   const userDetails = req.body;
   userDetails.id = id;
   console.log(userDetails);
   userData.push(userDetails);
   // write to file
   fs.writeFile("./data.json", JSON.stringify(userData), (err)
=> {
     if (err) {
       console.log(err);
```

```
    }
  });

  res.json({
    status: 200,
    data: req.body,
    message: `User created with id ${id}`,
  });
 }
});
```

5. if we need to create a middleware to implement a generic validation to check if the user input is empty or not
    a. implement the same check inside app.use(function(req, res, next){
    if(isEmpty){
    Return error
    } else {
    next()
    }
    }

Get user data : Dynamic routes

1. How do URLs with dynamic id look like
    a. How can we represent them?
        i. /api/user/:id

```
app.get("/api/user/:id", (req, res) => {
  const { id } = req.params;
  console.log("userid", id);
  return res.status(200).json({
    message: "User found",
  });
});
```

2. If we had more parameters in the url, we would be able to destructure them
3. Let us implement the whole find logic and see
4. Generally when we write backend code, we implement everything in try catch because if anything goes wrong while fetching or writing, we will handle that gracefully

```
app.get("/api/user/:id", (req, res) => {
  try {
    const { id } = req.params;
   const user = userData.find((user) => user.id == id);
    console.log("user", user);
   if (!user) {
      // res.status(404).json({
      //     status: 404,
      //     message: "User not found",
      //   })
      throw new Error("User not found");
    }
else {
```

```
      return res.status(200).json({
        message: user,
      });
    }
  } catch (err) {
    return res.status(500).json({
      message: err.message,
    });
  }
});
```

5. HW - user - patch request
6. HW - delete user