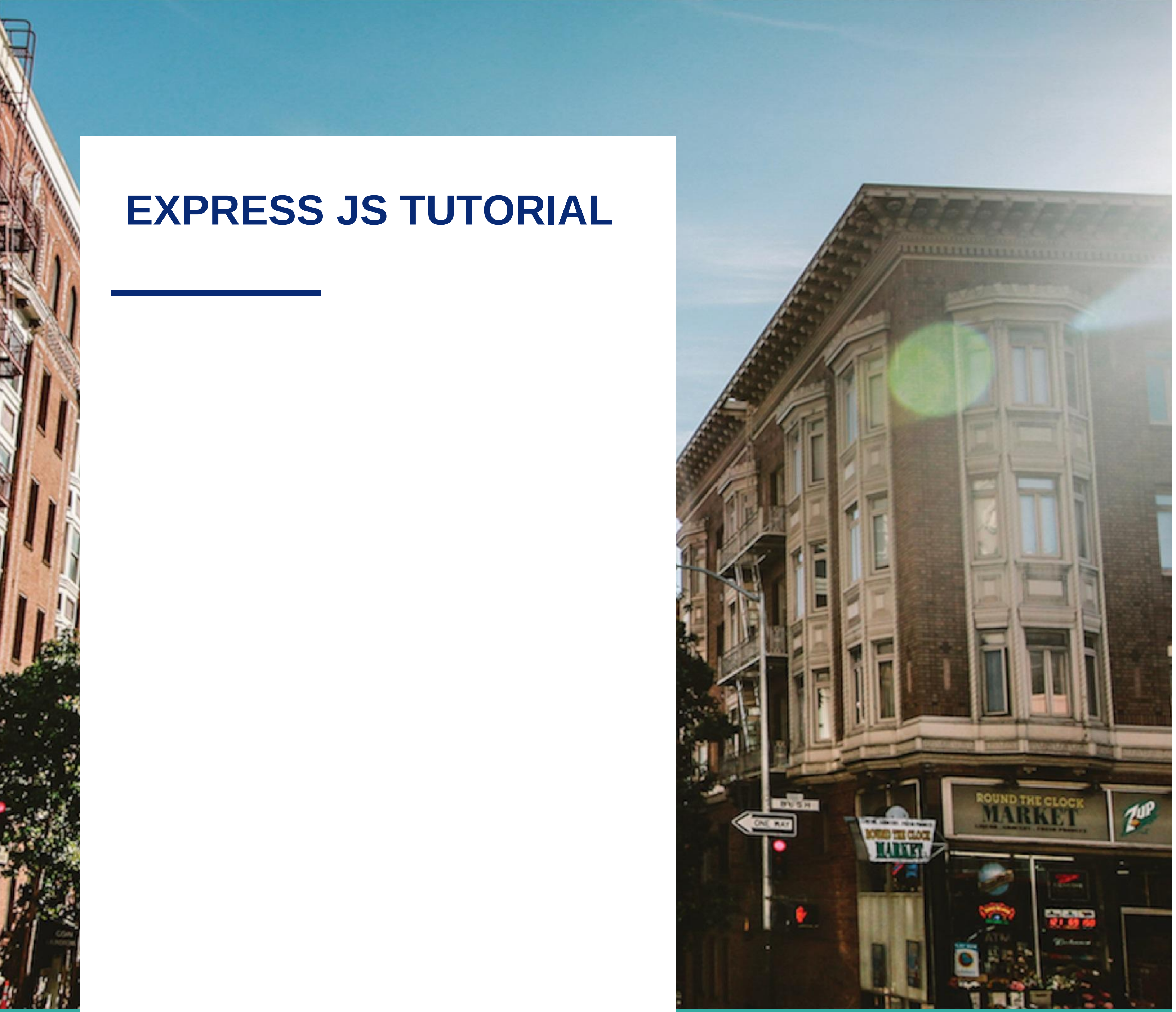

EXPRESS JS TUTORIAL

JANUARY 23

ABES ENGINEERING COLLAGE

1. Install Node.js and Express

Before starting with Express.js, make sure you have Node.js installed. You can download and install it from the Node.js website.

Once Node.js is installed, you can use npm (Node Package Manager) to install Express.

```
# Initialize a new Node.js project
npm init -y

# Install Express.js
npm install express
```

2. Create a Basic Express Server

In your project folder, create a file, say app.js, and add the following code to create a basic Express server:

```
const express = require('express');
const app = express();
const port = 3000;

// Define a basic route
app.get('/', (req, res) => {
 res.send('Hello, world!');
});

// Start the server
app.listen(port, () => {
 console.log(`Server is running at http://localhost:${port}`);
});
```

Run the server:

```
node app.js
```

Now, if you visit http://localhost:3000/ in your browser, you should see the message "Hello, world!".

3. Routing

Express.js allows you to handle different HTTP methods and routes. Here's an example with more routes:

```
// Handling different HTTP methods
app.get('/', (req, res) => {
 res.send('Welcome to the homepage');
});

app.post('/submit', (req, res) => {
 res.send('Form submitted!');
});

app.put('/update', (req, res) => {
 res.send('Data updated!');
});

app.delete('/delete', (req, res) => {
 res.send('Data deleted!');
});
```

4. Middleware

Middleware functions are used for tasks such as logging, authentication, or body parsing. Here’s an example of a basic middleware:

```
// Example of middleware to log request info
app.use((req, res, next) => {
 console.log(`${req.method} request for '${req.url}'`);
 next(); // Pass control to the next middleware or route handler
});

// Add your routes after the middleware
```

5. Serving Static Files

If you want to serve static files (e.g., images, stylesheets, JavaScript files), you can use the express.static() middleware:

```
// Serve static files from the 'public' directory
app.use(express.static('public'));
```

Make sure you have a folder named public in your project, and place your static files (e.g., index.html, styles.css) inside it.

6. Handling JSON and URL-encoded Data

You can easily handle incoming JSON data or URL-encoded data (like form submissions):

```
// Middleware to parse JSON bodies
app.use(express.json());

// Middleware to parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

app.post('/data', (req, res) => {
 console.log(req.body); // Logs the posted JSON or URL-encoded data
 res.send('Data received');
});
```

7. Error Handling

Express has built-in support for error handling. Here’s how you can create a simple error handler:

```
// A route that causes an error
app.get('/error', (req, res) => {
 throw new Error('Something went wrong!');
});

// Error handling middleware
app.use((err, req, res, next) => {
 console.error(err.stack);
 res.status(500).send('Something went wrong!');
});
```

Middleware

Introduction

Middleware functions are functions that have access to the request (req), response (res), and the next function in the application's request-response cycle. They can:

- Execute code.
- Modify the request and response objects.
- End the request-response cycle.
- Call the next() function to pass control to the next middleware function in the stack.

1. Basic Syntax of Middleware

Middleware functions are typically defined as:

```
app.use((req, res, next) => {
 // Perform some logic
 next(); // Pass control to the next middleware or route handler
});
```

2. Types of Middleware

1. Application-level Middleware

This type of middleware is bound to the entire application. It runs for all routes and requests unless specified otherwise.

Example of logging middleware:

```
app.use((req, res, next) => {
 console.log(`${req.method} request to ${req.url}`);
 next(); // Call the next middleware
});
```

In this case, this middleware logs every request's method and URL before passing control to the next middleware.

2. **Router-level Middleware**

Express allows you to define middleware for specific routers. For instance, if you want to apply middleware only to a particular group of routes, you can use express.Router().

Example:

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

router.use((req, res, next) => {
 console.log('Middleware for this specific route');
 next();
});

router.get('/profile', (req, res) => {
 res.send('User profile page');
});

app.use('/user', router); // Apply the middleware to all /user routes
```

3. **Error-handling Middleware**

Express provides a special kind of middleware for error handling, which takes 4 parameters: err, req, res, and next.

Example:

```
// Middleware to handle errors
app.use((err, req, res, next) => {
 console.error(err.stack); // Log the error stack
 res.status(500).send('Something went wrong!');
});
```

This error-handling middleware will catch any errors that occur in the app and return a response to the client.

4. **Built-in Middleware**

Express comes with a few built-in middleware functions that you can use out of the box:

a. express.static(): Serves static files like images, stylesheets, etc.

b. express.json(): Parses incoming requests with JSON payloads.

c. express.urlencoded(): Parses incoming requests with URL-encoded payloads.

Example of parsing JSON:

```
app.use(express.json()); // Middleware to parse JSON
```

5. **Third-party Middleware**

You can also use third-party middleware to extend the functionality of your Express app. Some popular third-party middleware includes:

a. morgan: HTTP request logger.

b. cors: Handles Cross-Origin Resource Sharing (CORS).

c. body-parser: Parses incoming request bodies (though express.json() and express.urlencoded() are now built-in).

Example using morgan:

```
npm install morgan
```

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

// Use morgan middleware to log HTTP requests
app.use(morgan('tiny'));
```

6. Custom Middleware

You can create your own middleware to perform specific actions. For example, let's create middleware to check if a user is authenticated:

```
function isAuthenticated(req, res, next) {
 if (req.isAuthenticated()) {  // Check if the user is authenticated
   return next();
 }
 res.status(401).send('Unauthorized');
}

// Apply the middleware to a specific route
app.get('/dashboard', isAuthenticated, (req, res) => {
 res.send('Welcome to your dashboard');
});
```

3. Order of Middleware Execution

The order in which middleware is added is very important. Express processes middleware in the order it's defined. That means:

- First-come, first-serve.
- The first app.use() or route handler that matches the request will be executed.

Example:

```
app.use((req, res, next) => {
 console.log('Middleware 1');
 next();
});

app.use((req, res, next) => {
 console.log('Middleware 2');
 next();
});

// Route handler
app.get('/', (req, res) => {
 res.send('Hello, Express!');
});
```

For a request to /, the output would be:

```
Middleware 1
Middleware 2
Hello, Express!
```

4. Middleware for Specific Routes

You can also apply middleware to specific routes or groups of routes.

Example:

```
// Apply middleware only to /admin routes
app.use('/admin', (req, res, next) => {
 console.log('Admin route accessed');
 next();
});

// Only the /admin route will trigger this middleware
```

```
app.get('/admin', (req, res) => {
 res.send('Admin page');
});
```

5. Chaining Middleware

You can chain multiple middleware functions together to handle various tasks.

Example:

```
app.use((req, res, next) => {
 console.log('First middleware');
 next();
});

app.use((req, res, next) => {
 console.log('Second middleware');
 next();
});

app.get('/', (req, res) => {
 res.send('Response after all middleware');
});
```

6. Middleware to Parse Request Bodies

Use built-in middleware functions to handle body parsing:

- For JSON bodies:

```
app.use(express.json());
```

- For URL-encoded data:

```
app.use(express.urlencoded({ extended: true }));
```

This way, Express will automatically parse incoming requests with JSON or URL-encoded bodies and populate the req.body object.

7. Ending the Request-Response Cycle

Sometimes, a middleware function will end the request-response cycle. For example:

```
app.use((req, res, next) => {
 res.send('Request finished here');
 // No need to call next() because we sent a response
});
```

HTTP Request

1. Accessing Request Properties

The request object provides various properties that allow you to access information about the incoming HTTP request.

1.1 req.method

The req.method property contains the HTTP method (GET, POST, etc.) used for the request.

```
app.get('/', (req, res) => {
 res.send(`Request Method: ${req.method}`);
});
```

1.2 req.url

The req.url property gives you the full URL of the incoming request (excluding the hostname).

```
app.get('/about', (req, res) => {
 res.send(`Request URL: ${req.url}`);
});
```

1.3 req.headers

The req.headers object contains all the HTTP headers sent by the client.

```
app.get('/headers', (req, res) => {
 res.json(req.headers); // Sends all headers as JSON
});
```

1.4 req.query

The req.query object contains query string parameters. Query strings are the part of the URL that comes after the ?, e.g., /search?term=express.

Example URL: http://localhost:3000/search?term=express

```
app.get('/search', (req, res) => {
 const searchTerm = req.query.term; // Access the 'term' query parameter
 res.send(`Search term: ${searchTerm}`);
});
```

For the above request, visiting http://localhost:3000/search?term=express will return:

```
Search term: express
```

1.5 req.params

The req.params object contains route parameters (variables in the URL path). These are dynamic segments of the URL that are captured and used in the route.

Example URL: http://localhost:3000/user/42

```
app.get('/user/:id', (req, res) => {
 const userId = req.params.id; // Access the 'id' parameter from the URL
 res.send(`User ID: ${userId}`);
});
```

Visiting http://localhost:3000/user/42 will return:

```
User ID: 42
```

1.6 req.body

The req.body object contains data sent in the request body. This is commonly used with POST, PUT, or PATCH requests when data is sent from the client to the server.

In order to access req.body, you'll need to use middleware to parse the request body, such as express.json() for JSON data or express.urlencoded() for URL-encoded data.

Example using express.json():

```
app.use(express.json()); // Middleware to parse JSON bodies

app.post('/submit', (req, res) => {
 const formData = req.body; // Access JSON data from the request body
 res.send(`Received data: ${JSON.stringify(formData)}`);
});
```

Example POST request body:

```
{
 "name": "Ashish",
 "age": 45
}
```

The server will respond with:

```
Received data: {"name":"Ashish","age":45}
```

2. Request Types

In Express, you can handle different types of requests using various HTTP methods (GET, POST, PUT, DELETE, etc.).

2.1 Handling GET Requests

```
app.get('/profile', (req, res) => {
 res.send('This is the profile page');
});
```

2.2 Handling POST Requests

```
app.post('/submit', (req, res) => {
 const data = req.body;
 res.send(`Form data received: ${JSON.stringify(data)}`);
});
```

2.3 Handling PUT Requests

```
app.put('/update/:id', (req, res) => {
 const id = req.params.id;
 const updatedData = req.body;
 res.send(`Updated user with ID ${id}: ${JSON.stringify(updatedData)}`);
});
```

2.4 Handling DELETE Requests

```
app.delete('/delete/:id', (req, res) => {
 const id = req.params.id;
 res.send(`User with ID ${id} has been deleted`);
});
```

3. Handling Middleware for Request Parsing

To handle different request bodies, you’ll often need middleware. Express has built-in middleware to parse incoming data.

3.1 JSON Body Parsing

```
app.use(express.json()); // Middleware to parse JSON bodies
```

3.2 URL-Encoded Body Parsing

```
app.use(express.urlencoded({ extended: true })); // Middleware for URL-encoded data
```

This will allow you to access form data sent as application/x-www-form-urlencoded in req.body.

4. Handling Errors and Missing Routes

If a route is not matched, Express will return a 404 by default. You can also set up your own error-handling middleware.

4.1 404 Not Found Handler

```
app.use((req, res) => {
 res.status(404).send('Route not found');
});
```

4.2 Error-handling Middleware

```
app.use((err, req, res, next) => {
 console.error(err.stack);
 res.status(500).send('Something went wrong!');
});
```

5. Accessing Cookies and Sessions

If you need to access cookies or manage sessions, you can use the cookie-parser and express-session middleware.

5.1 Using cookie-parser

```
npm install cookie-parser
```

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());

// Access cookies
app.get('/cookies', (req, res) => {
 res.send(req.cookies);
});
```

5.2 Using express-session

```
npm install express-session
```

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

app.use(session({
 secret: 'your_secret',
 resave: false,
 saveUninitialized: true
}));

// Access session
app.get('/session', (req, res) => {
 res.send(req.session);
});
```

HTTP RESPONSE

res.send() - Send a plain response (string, object, array, etc.).

res.json() - Send a JSON response.

res.status() - Set the HTTP status code.

res.sendStatus() - Send status code with default message.

res.redirect() - Redirect the client to a different URL.

res.sendFile() - Send a file to the client.

res.render() - Render a view template with dynamic data.

res.set() - Set response headers.

res.type() - Set the Content-Type of the response.

1. Sending Responses

The response object provides several methods to send responses back to the client. Here are the most common ones:

1.1 res.send()

The res.send() method sends a response to the client. This can be a string, an object, an array, a buffer, etc.

```
app.get('/', (req, res) => {
 res.send('Hello, world!');
});
```

You can also send objects or arrays as JSON:

```
app.get('/json', (req, res) => {
 const data = { message: 'Hello, JSON!' };
 res.send(data);  // Automatically converts to JSON
});
```

1.2 res.json()

The res.json() method is used specifically to send JSON responses. Express automatically sets the Content-Type header to application/json when sending the response.

```
app.get('/json', (req, res) => {
 const user = { name: 'Ashish', age: 45 };
 res.json(user);  // Automatically sends JSON with correct headers
});
```

This will send a JSON response like:

json

```
{
 "name": "Ashish",
 "age": 45
}
```

1.3 res.status()

The res.status() method is used to set the HTTP status code of the response. This method is typically used before sending the response data.

```
app.get('/notfound', (req, res) => {
 res.status(404).send('Page not found');
});
```

Here, we're sending a 404 Not Found status with a message.

1.4 res.sendStatus()

The res.sendStatus() method sets the HTTP status code and sends a message corresponding to that code.

```
app.get('/success', (req, res) => {
 res.sendStatus(200); // Sends HTTP status 200 (OK) with the default message
});
```

For 200, it will send OK as the body of the response.

1.5 res.redirect()

The res.redirect() method is used to redirect the client to another URL. You can pass the status code and the target URL.

```
app.get('/old-url', (req, res) => {
 res.redirect(301, '/new-url'); // Redirect with 301 (Moved Permanently)
});

app.get('/new-url', (req, res) => {
 res.send('You have been redirected to the new URL');
});
```

1.6 res.sendFile()

The res.sendFile() method is used to send a file as a response. The path to the file can be relative or absolute.

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

app.get('/file', (req, res) => {
 res.sendFile(path.join(__dirname, 'public', 'index.html'));  // Sends a file
});
```

This will send the index.html file located in the public directory.

1.7 res.render()

If you are using a template engine like EJS, Pug, or Handlebars, you can render a view using res.render().

DATABASE CONNECTIVITY

1. Database Connectivity with MongoDB

MongoDB is a NoSQL database, and it's a popular choice for Express applications due to its flexibility and ease of integration.

Step 1: Install Dependencies

First, you'll need to install the MongoDB client. Express uses the Mongoose package, which is a popular ODM (Object Data Modeling) library for MongoDB in Node.js.

```
npm install mongoose
```

Step 2: Set up MongoDB Connection

In your Express app, you can set up the MongoDB connection using Mongoose.

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const port = 3000;

// MongoDB connection string
const dbURI = 'mongodb://localhost:27017/userDB;  // Replace with your MongoDB
URI

// Connect to MongoDB using Mongoose
mongoose.connect(dbURI, { useNewUrlParser: true, useUnifiedTopology: true })
 .then(() => console.log('Connected to MongoDB'))
 .catch((err) => console.log('Error connecting to MongoDB:', err));

// Basic route to test connection
app.get('/', (req, res) => {
 res.send('Hello, MongoDB!');
});
```

```
app.listen(port, () => {
 console.log(`Server running at http://localhost:${port}`);
});
```

Step 3: Create a Model and Schema

Mongoose allows you to define models that represent your data structure. Here's an example of creating a User model.

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

const userSchema = new mongoose.Schema({
 name: String,
 email: { type: String, unique: true },
 age: Number
});


const User = mongoose.model('User', userSchema);


app.post('/create-user', async (req, res) => {
 const newUser = new User({
   name: 'Ashish',
   email: 'ab@gmail.com',
   age: 45
 });

 try {
   await newUser.save();
   res.send('User created successfully!');
 } catch (err) {
   res.status(400).send('Error creating user');
 }
});
```

In this example, we define a schema for a user with name, email, and age fields. The save() method is used to save the data to the MongoDB database.

2. Database Connectivity with MySQL

MySQL is a relational database, and it’s commonly used with Express for applications that require structured data storage.

Step 1: Install Dependencies

To connect to a MySQL database, you can use the mysql2 package, which provides a promise-based API for working with MySQL.

npm install mysql2

Step 2: Set up MySQL Connection

Here’s an example of how to set up MySQL connectivity using the mysql2 library:

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const port = 3000;

const pool = mysql.createPool({
 host: 'localhost',
 user: 'root',        // Your MySQL username
 password: '',        // Your MySQL password
 database: 'expressjs_db' // Your database name
});
```

```
app.get('/', (req, res) => {
 res.send('Hello, MySQL!');
});

app.get('/users', (req, res) => {
 pool.query('SELECT * FROM users', (err, results) => {
   if (err) {
     res.status(500).send('Error querying database');
     return;
   }
   res.json(results);
 });
});

app.listen(port, () => {
 console.log(`Server running at http://localhost:${port}`);
});
```

Step 3: Create a Table and Insert Data

You can use the MySQL CLI or a MySQL GUI (like phpMyAdmin or MySQL Workbench) to create the users table. Here's an example SQL query to create a users table:

```
CREATE TABLE users (
 id INT AUTO_INCREMENT PRIMARY KEY,
 name VARCHAR(100),
 email VARCHAR(100),
 age INT
);
```

You can also add some data to the users table with an SQL query:

```
INSERT INTO users (name, email, age) VALUES
('Ashish', 'ab@gmail.com', 45),
('Prashant', 'pt@gmail.com', 40);
```

Now, when you visit http://localhost:3000/users, you should see a list of users from the database.

3. Handling Environment Variables

For security reasons, sensitive information like database credentials (username, password, etc.) should not be hardcoded in your code. Instead, use environment variables to store them.

Step 1: Install dotenv

```
npm install dotenv
```

Step 2: Create a. env File

In the root of your project, create a .env file to store your database credentials:

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=yourpassword
DB_NAME=expressjs_db
```

Step 3: Load Environment Variables in Your Code

At the top of your main application file (e.g., app.js), require the dotenv package to load the environment variables:

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

const express = require('express');
const mongoose = require('mongoose');
const app = express();
const dbURI =
`mongodb://${process.env.DB_HOST}:27017/${process.env.DB_NAME}`;

mongoose.connect(dbURI, { useNewUrlParser: true, useUnifiedTopology: true })
 .then(() => console.log('Connected to MongoDB'))
 .catch((err) => console.log('Error connecting to MongoDB:', err));
```

For MySQL, you can modify the connection settings to use environment variables as follows:

```
const pool = mysql.createPool({
 host: process.env.DB_HOST,
 user: process.env.DB_USER,
 password: process.env.DB_PASSWORD,
 database: process.env.DB_NAME
});
```

Step 1: Install the MongoDB Node.js Driver

First, you need to install the MongoDB Node.js driver package:

```
npm install mongodb
```

This will install the mongodb package that allows you to connect to MongoDB using MongoClient.

Step 2: Set up MongoDB Connection in Express

Once you have the mongodb package installed, you can use the MongoClient to connect to your MongoDB database. Here's how to set up a basic Express app with MongoDB connectivity.

Create app.js

```
const express = require('express');
const { MongoClient } = require('mongodb');  // Import MongoClient from mongodb
package
const app = express();
const port = 3000;

// MongoDB connection URI
const uri = 'mongodb://localhost:27017/userDB;  // Replace with your MongoDB
URI
let db;

// Connect to MongoDB
MongoClient.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true })
 .then(client => {
   console.log('Connected to MongoDB');
   db = client.db();  // Store the database object
 })
 .catch(error => console.error('Error connecting to MongoDB:', error));

// Basic route to test MongoDB connection
app.get('/', (req, res) => {
 res.send('Hello, MongoDB with MongoClient!');
});

// Example of fetching data from a collection
app.get('/users', async (req, res) => {
 try {
  const usersCollection = db.collection('users');  // Get the 'users' collection
  const users = await usersCollection.find().toArray();  // Fetch all users
  res.json(users);  // Send the users as JSON response
 } catch (err) {
  console.error('Error fetching users:', err);
  res.status(500).send('Error fetching users');
 }
});



// Example of inserting data into MongoDB
app.post('/add-user', async (req, res) => {
 const newUser = { name: 'Ashish', email: 'ab@gmail.com', age: 45 };
 try {
  const usersCollection = db.collection('users');
  const result = await usersCollection.insertOne(newUser);
  res.json({ message: 'User added', result });
 } catch (err) {
  console.error('Error inserting user:', err);
  res.status(500).send('Error inserting user');
 }
});

// Start the server
app.listen(port, () => {
 console.log(`Server running at http://localhost:${port}`);
});
```

Step 3: Create the Users Collection

Before inserting data, make sure that you have created a users collection in your MongoDB database. You can do this manually via MongoDB shell or it will be created automatically the first time you insert data into it.

If you want to create a users collection manually, you can do this in the MongoDB shell:

```
use userDB
db.createCollection('users')
```

Alternatively, MongoDB will automatically create the collection when you insert data for the first time.

Step 4: Testing the Application

1. Start the Server: Run the Express server by using the following command:

   ```
   node app.js
   ```

2. Add a User: Send a POST request to /add-user (you can use Postman or any HTTP client to test this). This will insert a new user into the users collection.
3. Get Users: To view all users, visit http://localhost:3000/users in your browser or send a GET request to this route. It will return the list of users stored in the MongoDB database.

Step 5: Handling Errors and Closing the Database Connection

To ensure that errors are handled correctly and that the database connection is properly closed when the application shuts down, you can use process.on('SIGINT') for clean-up.

```
// Gracefully handle shutdown
process.on('SIGINT', () => {
 if (db) {
   db.client.close();  // Close the database connection when the server shuts down
 }
 console.log('Server shutting down...');
 process.exit(0);
});
```