**MERN Stack: Backend Development with Node.js**

**Introduction**

This course provides a comprehensive guide to backend development, focusing on Node.js. By the end, you will gain the knowledge and skills needed to become an industry-ready backend developer.

**Topics Covered**

1. Client-Server Architecture: Understand the foundation of backend development and the interaction between clients and servers.
2. Single-threaded Backend: Learn about threads, multi-threading, and the single-threaded backend in Node.js.
3. Components of a Backend System: Explore the different components that form the backend system and how they work together.
4. Single-threaded Nature of JavaScript: Understand the single-threaded nature and synchronous behavior of JavaScript.
5. Setting up Node.js: Learn how to install Node.js on your computer and begin using it for backend development.
6. Running Node.js: Run a basic Node.js program and understand REPL (Read-Eval-Print Loop) operations for interactive JavaScript coding.

**Client-Server Architecture**

In any application, there is a client-side component and a server-side component. The client is the interface the user interacts with directly, while the server handles computations and processes unseen by the user. For instance, when you send a message on a messaging app, the server receives the request, processes it, and sends a response back to the client.

**Single-threaded Backend**

Most backend languages are multi-threaded, allowing them to handle multiple tasks concurrently. In contrast, Node.js operates in a single-threaded manner, managing one task at a time. Node.js uses an event loop to handle multiple tasks efficiently, providing a smooth user experience.

**Components of a Backend System**

A backend system consists of several interconnected components, including databases, servers, and applications. These elements work together to process client requests and deliver responses.

**Single-threaded Nature of JavaScript**

JavaScript, the programming language underlying Node.js, is single-threaded, meaning it processes one task at a time. JavaScript's event loop allows it to manage multiple tasks effectively, even in a single-threaded environment.

**Setting up Node.js**

Setting up Node.js is a straightforward process that allows you to start writing and running JavaScript code on your computer. Once installed, you can begin using Node.js for backend development.

**Running Node.js**

After setting up Node.js, you can run a basic Node.js program on your computer. Additionally, you can use REPL (Read-Eval-Print Loop) operations to interactively run JavaScript code in your terminal.

**MERN Stack: Node.js - Global Objects, Modules, and File System**

In this session, we explore the Node.js environment with a focus on Global Objects, Modules, and the File System. This provides a comprehensive understanding of how to utilize Node.js effectively.

**Node.js Global Objects**

Node.js has several global objects available in all modules, differing from the window object in JavaScript. Some key global objects in Node.js include:

* global: Represents the global namespace object.
* \_\_dirname: The directory name of the current module.
* \_\_filename: The file name of the current module.
* console: Allows printing to stdout and stderr.
* setTimeout(cb, ms): Executes a callback after a specified number of milliseconds.
* clearTimeout(t): Cancels a timer previously created with setTimeout().
* setInterval(cb, ms): Executes a callback repeatedly after a specified number of milliseconds.
* clearInterval(t): Cancels a timer previously created with setInterval().

**Node.js Modules**

Node.js uses modules to organize code into reusable parts, adhering to the CommonJS specification. Modules can be imported using the require() function, while module.exports is used to export a module for use in other parts of the application.

**Node.js File System**

Node.js includes a built-in fs module for interacting with the file system, offering methods for file and directory operations such as:

* fs.writeFile(file, data, callback): Writes data to a file asynchronously.
* fs.readFileSync(file): Reads a file synchronously.
* fs.mkdir(path, callback): Creates a new directory.
* fs.unlink(path, callback): Deletes a file.
* fs.rmdir(path, callback): Deletes a directory.

**Practical Application**

In the session, we demonstrated the practical application of these concepts by creating a Node.js script that exports an array of fruits and a function to display them. We then imported these in another script and displayed the fruits. We also used the fs module to create a file, write data to it, read the data, and delete the file.

# MERN Stack: Node.js, Server Development, and APIs

In this lecture, we'll cover the basics of Node.js, server development, and APIs. We'll revisit the global object in Node.js and its methods, discuss how to import and export items using CommonJS syntax, and explore how to manipulate the file system with Node.js.

## Node.js and File System

Node.js provides the global object, which is the root object in the Node.js environment. This object allows for global access to various methods and properties like setTimeout(), clearTimeout(), console, \_\_dirname, and \_\_filename.

The fs (file system) module in Node.js lets you interact with the file system, performing operations such as reading, writing, and deleting files and directories. It's essential for server development to understand how to work with the file system.

## Server Development Basics

### Client-Server Communication

In server development, understanding client-server communication is key. This involves knowing how data is exchanged between a client (e.g., a web browser) and a server. This communication often relies on IP addresses, which enable a browser to locate and connect with a server.

### HTTP Protocol and Methods

The HTTP (HyperText Transfer Protocol) is the foundation of data communication on the web. It involves various methods, such as:

* GET: Used to retrieve data from a server.
* POST: Used to send data to a server.
* DELETE: Used to remove data from a server.
* PUT: Used to update data on a server.

### HTTP Status Codes

HTTP status codes indicate the response from the server and whether a client's request has been successful. They are divided into series:

* 1xx: Informational, indicating the request was received.
* 2xx: Success, indicating the request was successful.
* 3xx: Redirection, indicating further action is needed.
* 4xx: Client error, indicating a request error.
* 5xx: Server error, indicating a server problem.

## APIs (Application Programming Interfaces)

APIs are a set of rules and protocols that allow different software applications to interact with one another. APIs work as intermediaries, handling requests and delivering data, much like a waiter taking orders from customers (clients) and delivering them to the kitchen (server).

## Conclusion

Understanding these key concepts is essential for developing robust server applications using Node.js. These include a solid grasp of Node.js global objects, file system manipulation, and the fundamentals of server communication via the HTTP protocol and APIs.

In the upcoming lectures, we'll dive deeper into Node.js server development and API integration, giving you a comprehensive understanding of how to build efficient and scalable server applications.

**MERN Stack: Node Server Creation and HTTP Handling**

**Node Server Basics**

* Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, known for its speed and efficiency.
* A server is a computer program or device that provides functionality to other programs or devices, called "clients."
* A Node server is a server-side platform that leverages Node.js to build fast and scalable network applications.

**Creating a Node Server**

1. Create a new folder and open it in your code editor (e.g., Visual Studio Code).
2. Create a new file named server.js in the folder.
3. Write server code in server.js:
   * Import the built-in http package.
   * Use http.createServer() to instantiate a server.
   * The createServer() method requires a callback function that handles request and response objects.
   * Use console.log() to log messages when the server receives requests.
   * Use server.listen() to bind the server to a specified port. The listen() method accepts three parameters: port number, local host, and a callback function.
   * The callback function logs a message when the server is ready to listen on the specified port.

**Running the Node Server**

* Execute node server.js in your terminal to run the server.
* The server will now listen on the designated port and handle incoming requests.

**Sending Responses from the Server**

* Use response.write() to write data to the client and response.end() to conclude the response.
* The server can send responses to the client in response to received requests.

**Serving HTML from the Server**

* Utilize the fs (file system) module in Node.js to read and serve HTML files.
* Use fs.readFile() to read HTML files. This method accepts the file path and a callback function.
* The callback function receives error and data parameters. If an error occurs, log it and send an error message as a response. Otherwise, send the data (HTML file content) as the response.

**Handling HTTP Status Codes in Node Server**

* HTTP status codes are standardized response codes provided by web servers.
* These codes help identify issues when a web page or resource doesn't load properly.
* You can set HTTP status codes using response.statusCode.
* The default status code is 200 (OK), indicating successful operation.
* For errors, set the status code to 500 (Internal Server Error).
* If the requested resource isn't found, set the status code to 404 (Not Found).

**Handling Different Paths**

* Use request.url to handle different paths and serve various resources based on the URL.
* If the URL doesn't match any defined paths, serve a 404 page and set the status code to 404.

**Conclusion**

Creating a Node server and managing HTTP responses and status codes are essential skills for server development. By understanding the fundamentals and applying them effectively, you can build efficient and robust server applications with Node.js.

These concepts provide a strong foundation for more advanced topics such as API development and middleware integration in the Node.js ecosystem. Explore further to expand your skills in these areas and enhance your backend development capabilities.

# MERN Stack: Node.js, Postman, and Creating a Calculator Application

In this session, we delve into creating a calculator application using Node.js and Postman. We start by creating a server with Node.js, then use Postman to test our APIs.

## Postman

Postman is a tool used by backend developers for testing APIs. It allows you to send requests to your server and view the responses. You can download Postman from their official website and install it on your device. Once installed, you can input your API's URL, select the request method (GET, POST, etc.), and send the request. Postman will then display the server's response.

## Creating a Calculator Application

Our calculator application will expose APIs for different mathematical operations. We'll start by creating a server with Node.js. We'll then use Postman to send requests to our server and view the responses.

We'll create a POST request that takes in two numbers and an operation (add, subtract, multiply, or divide) as input. The server will perform the operation and return the result.

Here's a basic outline of the code:

**const** http = require('http');

**const** url = require('url');

**const** querystring = require('querystring');

http.**createServer**((req, res) => {

**if** (req.method === 'POST') {

**let** body = '';

req.**on**('data', chunk => {

body += chunk.**toString**();

});

req.**on**('end', () => {

**const** data = querystring.**parse**(body);

**switch** (data.operation) {

**case** 'add':

res.**end**(`Sum is ${data.num1 + data.num2}`);

**break**;

*// Add cases for subtract, multiply, and divide*

default:

res.statusCode = 404;

res.**end**('Event not found');

}

});

} **else** {

res.statusCode = 400;

res.**end**('Invalid request');

}

}).**listen**(3000);

This code creates a server that listens for POST requests. When a request is received, it reads the data from the request body, performs the specified operation, and sends the result back in the response.

## Next Steps

In the next session, we'll handle edge cases in our calculator application and discuss the Node Package Manager (NPM). We'll see how to create a Node.js project with NPM and how the package.json file created by NPM can help us manage our project.

**MERN Stack Development: Node.js, Express, and NPM**

**Overview**

This lecture covers the development of APIs using Node.js and Express, and the use of NPM (Node Package Manager) for project setup and management. It provides an overview of API development, including error handling and how to use Express.js for building robust, efficient APIs.

**Node.js API Development**

* The lecture starts with a review of a previously created API that performs basic arithmetic operations such as addition, subtraction, multiplication, and division using two numbers and an event type.
* The lecturer addresses how to handle potential edge cases, such as subtracting a larger number from a smaller one and handling division by zero.

**NPM (Node Package Manager)**

* NPM is a package manager for the JavaScript runtime environment Node.js. It simplifies the process of managing project dependencies and packages.
* The lecture demonstrates how to initialize a new Node.js project using the npm init command, which creates a package.json file to manage the project's settings and dependencies.
* NPM also provides commands for installing packages (npm install <package-name>) and managing project dependencies.

**Express.js**

* Express.js is a back-end web application framework for Node.js, commonly used for building APIs.
* Express simplifies API development by providing a robust set of features and middleware, reducing the amount of code developers need to write and enhancing performance.
* The lecture demonstrates how to install Express.js using NPM and how to create a basic Express server that listens on a specified port and exposes an endpoint for arithmetic operations.

**Middleware and Utilities**

* Body Parser is a middleware used to parse incoming request bodies in JSON format before handling them in route handlers. This helps to facilitate JSON data processing in APIs.
* NodeMon is a utility that monitors source code changes and automatically restarts the server. It is often installed globally using NPM to streamline the development workflow.

**Conclusion**

* The lecture concludes with a demonstration of the Express.js application performing arithmetic operations based on API requests.
* The instructor emphasizes the importance of using Express.js for efficient and organized API development.
* In future lectures, the instructor will explore advanced Express.js features such as routing, middleware, and error handling, providing deeper insights into the development of scalable and maintainable server-side applications.

**MERN Stack: Express.js and Middleware**

In this session, we delve deeper into Express.js, a popular Node.js framework, and explore the concept of middleware. We'll also discuss how to create a simple to-do application using Express.js.

**Express.js**

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It simplifies the process of writing server-side applications by providing a simple and flexible API to build robust websites and web apps.

**Middleware**

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. Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware function in the stack.

Middleware functions are used to modify req and res objects for tasks like parsing request bodies, adding response headers, etc.

**Creating a To-Do Application**

We'll create a simple to-do application with the following endpoints:

* Get all to-dos: GET /todos
* Get a single to-do: GET /todo/:id
* Add a to-do: POST /todo
* Delete a to-do: DELETE /todo/:id
* Update a to-do: PUT /todo/:id

Each endpoint corresponds to a specific operation in the application. The HTTP method (GET, POST, DELETE, PUT) indicates the type of operation.

**Conclusion**

Understanding Express.js and middleware is crucial for building robust Node.js applications. In the next session, we'll start building our to-do application.

**MERN Stack Development: Node.js, Express.js, and Data Storage**

This lecture focuses on creating a to-do application using Node.js and Express.js. The application allows users to add, delete, and update to-do items while exploring data storage considerations and the use of databases for future improvements.

**Node.js and Express.js Overview**

* Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, which allows developers to use JavaScript for server-side scripting and command-line tools.
* Express.js is a web application framework for Node.js that simplifies the process of writing server-side code and provides a range of features for building web and mobile applications.

**To-Do Application Structure**

The to-do application consists of:

* todos.json: A file used to store to-do items.
* index.js: The main file that handles server-side logic and routes.

Each to-do item in todos.json is an object with a unique id and a text property that contains the to-do item's content. The index.js file uses Express.js to create the server, handle HTTP requests, and leverage the fs (file system) module to interact with the todos.json file.

**HTTP Requests Handling**

The application supports the following types of HTTP requests:

* GET requests: Retrieve all to-do items.
* POST requests: Add a new to-do item.
* PUT requests: Update an existing to-do item.
* DELETE requests: Remove an existing to-do item.

Each request type has a corresponding function in index.js, which uses the fs module to interact with todos.json and the express.json() middleware to parse incoming request bodies.

**Error Handling**

The application is designed to handle errors by checking specific conditions and sending appropriate HTTP status codes and messages:

* For instance, if a PUT or DELETE request is made for a non-existent to-do item, the application responds with a 404 status code and an error message.

**Data Storage**

While the application uses a JSON file for data storage, this approach may pose challenges as the application grows. As such, future improvements may include:

* Transitioning to a database management system (DBMS) for efficient data storage and retrieval.
* Using a database such as MongoDB, which is well-suited for MERN stack development and scalable applications.

**Conclusion**

The lecture provided insights into using Node.js and Express.js to create a simple to-do application and addressed key aspects such as handling different HTTP requests, working with a JSON file, and managing errors.

**MERN Stack: Introduction to MongoDB**

In our development of the to-do application using Express, we've been storing data in a JSON file. However, as the application evolves, using a file system for data storage becomes less ideal due to challenges such as scalability, performance, and the lack of query language.

**Why Use MongoDB?**

Switching to a database offers several benefits:

* Data Persistence: Ensures data survives application restarts and system failures, improving the reliability of the application.
* Concurrent Access: Efficiently handles multiple simultaneous requests without impacting data integrity or application performance.
* Query Language: Provides structured languages like SQL or MongoDB's query language, making data manipulation and extraction straightforward.

**Types of Databases**

There are two primary types of databases:

* SQL (Relational): Uses a tabular structure and is suitable for transactional data with relationships.
* NoSQL (Non-Relational): Offers flexibility in data storage formats such as document, key-value, column-family, and graph stores. Ideal for scenarios where data structure is dynamic and evolving.

**Focus on MongoDB**

In this course, we'll use MongoDB, a leading NoSQL database. MongoDB offers flexibility and scalability by storing data in collections of JSON-like documents. This approach aligns well with our to-do application's needs:

* Collections and Documents: MongoDB's collections are similar to tables, and each document within a collection represents an item (e.g., a to-do).
* Dynamic Schema: MongoDB allows varying fields across documents, supporting the flexible and evolving nature of our data.

**Interacting with MongoDB using Mongoose**

To interface with MongoDB, we'll use Mongoose, an Object Data Modeling (ODM) library for MongoDB and Node.js. Mongoose simplifies the interaction between our Node.js application and MongoDB by providing:

* Schemas: Define the structure and constraints of data in collections.
* Models: Represent collections and provide methods for data manipulation.
* Validation: Helps ensure data consistency by enforcing data rules defined in schemas.

**Next Steps**

In the upcoming part of the course, we will:

* Set up MongoDB: Learn how to install and configure MongoDB for use in our application.
* Migrate Data: Transition from file-based storage to MongoDB, ensuring data integrity and continuity.
* Use Mongoose: Explore Mongoose's features, including creating schemas, models, and performing CRUD (Create, Read, Update, Delete) operations.

By the end of this part of the course, you'll have a solid understanding of how to integrate MongoDB and Mongoose into your MERN stack application, enhancing its scalability and data handling capabilities.

**MERN Stack: MongoDB Integration**

In the MERN stack, MongoDB serves as the NoSQL database that stores data in a JSON-like format. It is an integral part of the stack, which includes MongoDB, Express.js, React.js, and Node.js, providing an efficient way to manage data in our application.

**MongoDB Overview**

* Flexible Schema: Unlike traditional relational databases, MongoDB allows for flexible and dynamic data modeling with its schema-less document-based approach.
* Scalability: MongoDB supports sharding, enabling horizontal scaling for handling large datasets and high throughput.

**Setting Up MongoDB**

* Register on mongodb.com: Begin by signing up on the MongoDB website.
* Create a Database: Use the dashboard to set up a new database, which includes a unique connection URL for your application.

**Connecting MongoDB to Node.js**

* Mongoose Package: Mongoose is an Object Data Modeling (ODM) library that simplifies the process of connecting and interacting with MongoDB from a Node.js application.
* Schema-based Solution: Mongoose enables the definition of schemas, which help model your application data while providing built-in type casting, validation, and query building.

**Creating a Model in Mongoose**

* Mongoose Models: Models are classes that represent collections in your database, with each model containing methods for working with documents.
* Defining Models: Use the Mongoose.Schema interface to create a model, specifying the structure and properties of documents.

**Connecting to MongoDB**

* mongoose.connect(): Use the mongoose.connect() method to establish a connection with your MongoDB database using the provided connection URL.
* Promise-based Connection: The method returns a promise, allowing you to handle asynchronous operations with callbacks for success and error handling.

**Performing CRUD Operations**

* Create, Read, Update, Delete: Once the connection is established, use Mongoose's methods (save(), find(), findOne(), update(), and remove()) to perform CRUD operations on your data.
* Data Manipulation: Mongoose's methods make data manipulation straightforward and efficient.

**Error Handling in Mongoose**

* Catch Errors: When working with promises, use the .catch() method to handle errors gracefully during asynchronous operations.
* Informative Responses: Provide clear error messages to help with debugging and user experience.

**Future Topics and Enhancements**

* Advanced MongoDB Integration: In the upcoming session, we'll dive deeper into integrating MongoDB with delete and update endpoints.
* Authentication: We'll begin exploring authentication strategies, a crucial feature in modern web applications for secure access control and data protection.

As you continue your journey in MERN stack development, mastering MongoDB integration will enhance your ability to build efficient and scalable applications.

# MERN Stack: Node.js, MongoDB, and Authentication

This lecture focuses on integrating MongoDB with Node.js, building an authentication system, and implementing CRUD operations (Create, Read, Update, Delete) for a website. Let's explore these concepts in detail.

## MongoDB Integration

### Endpoints

* The lecture begins with the integration of MongoDB into the application, focusing on delete and update endpoints.

### Delete Operation

* The delete operation is handled using MongoDB's findOneAndRemove method. This method takes an object parameter containing the ID of the item to be deleted and removes it from the database.

### Update Operation

* The update operation is implemented using MongoDB's findOneAndUpdate method, which takes two parameters: the condition to find the item (e.g., its ID) and the new data to update the item with.

## Authentication System

### Basics

* The lecture progresses to the creation of an authentication system for the website. A login system allows users to enter their credentials (ID and password) to log in.

### Registration

* During the registration process, users provide information such as name, email, and password. Passwords are hashed for security before being stored in the database.

### Login

* The login process involves comparing the entered user credentials with the data stored in the database. If the credentials match, the user is authenticated and allowed to proceed.

## Hashing Passwords

* MD5 Hashing: Passwords are hashed using the MD5 algorithm to convert them into non-human-readable format. This hashed password is stored in the database, and during login, the entered password is hashed again and compared with the stored hash.

## User Registration and Login APIs

### User Registration API

* The registration API handles user creation with provided name, email, and hashed password, saving the data to MongoDB.

### User Login API

* The login API checks if the provided email exists in the database and then compares the hashed password. If the credentials match, the user is successfully authenticated.

## Error Handling

### Try-Catch

* Error handling is done using try-catch blocks to capture exceptions in the code, allowing for graceful handling of errors and user-friendly error messages.

### Server Errors

* When errors occur on the server side (e.g., a database connection failure), a 500 internal server error is returned to the client, indicating the problem is server-side.

## Best Practices and Future Considerations

* Secure Authentication: Consider modern hashing algorithms such as bcrypt for password hashing for added security.
* Use SSL/TLS: Secure data transmission between the client and server by implementing SSL/TLS certificates.
* Token-based Authentication: Explore token-based authentication (e.g., JSON Web Tokens) for secure user sessions.
* User Authorization: Implement user authorization to control access to various resources based on user roles.

In the next session, we'll delve into advanced topics such as token-based authentication and further explore securing the application. Continue to build on the foundational knowledge you've gained to create more robust and secure web applications.

**MERN Stack: Advanced User Authentication and Data Management**

This session covers advanced concepts in user authentication, hashing, encryption, and data management within the MERN stack. We focus on practical applications and advanced data management techniques to improve the overall security and efficiency of web applications.

**User Authentication**

* Token Generation: Upon a successful user login, a unique token is generated for the user. This token is used for authenticating the user in subsequent API calls.
* Token Usage: Tokens are typically passed in the headers of API calls. The server decodes the token to validate the user's identity.

**Hashing and Encryption**

* Password Hashing: Libraries such as bcrypt are used to hash user passwords before storing them in the database. During login, the provided password is hashed and compared with the stored hash for verification.
* Data Encryption: While hashing is one-way, encryption allows data to be encrypted and decrypted using keys. Base64 encoding can be used for data transfer and temporary storage.

**Data Management**

* CRUD Operations: Data management in the MERN stack involves creating, reading, updating, and deleting data using HTTP methods (GET, POST, PUT, DELETE) and MongoDB.
* Advanced Techniques: Techniques such as pagination and sorting help manage large datasets effectively. Pagination breaks data into discrete pages, while sorting organizes data in a specific order.

**Practical Application**

* Token-Based Authentication: Tokens generated upon login often contain user information and permissions. This information can include the user's email and ID, and is used to authorize subsequent requests.
* Secure Token Storage: Tokens are stored on the client-side in local storage, session storage, or cookies for future use in authentication. Storing tokens securely is essential for protecting user data.
* API Integration: During API calls, the token is sent in the request headers. The server validates the token, ensuring the user is authorized to access the requested resources.

**Best Practices for Secure Development**

* Secure Password Storage: Utilize modern hashing algorithms such as bcrypt for password hashing, and avoid using weaker methods such as MD5.
* Token Security: Consider using JWT (JSON Web Tokens) for secure token-based authentication. This provides more control over token contents and expiration.
* Data Protection: Implement SSL/TLS certificates for encrypted data transmission, ensuring secure data transfer between the client and server.
* Access Control: Implement proper access controls based on user roles to restrict access to resources and endpoints.

Understanding these advanced user authentication and data management techniques is crucial for building secure and efficient web applications. Libraries such as bcrypt and JSON Web Tokens (JWT) enable you to enhance your application's security and protect user data.

**MERN Stack: Understanding View Engines and EJS**

This course introduces essential concepts of View Engines and Embedded JavaScript (EJS), which play a vital role in the MERN stack, especially for server-side rendering.

**View Engines**

* Functionality: View Engines enable you to inject dynamic content into your HTML templates. They offer more functionality than static HTML files, including conditional rendering and looping constructs.
* Usage in Express: To set a view engine in an Express application, use the app.set('view engine', 'ejs') command. This specifies EJS as the view engine for rendering views.

**Embedded JavaScript (EJS)**

* EJS Overview: EJS is a simple and efficient templating language that allows you to generate HTML using plain JavaScript. It supports features like includes, variables, and control flow statements.
* Creating EJS Files: To use EJS, rename your HTML files with the .ejs extension. These files can then be rendered in your Express application using the res.render() function.

**Using EJS in Express Applications**

* Rendering Views: Use the res.render() function to render EJS templates. Pass the file path and any data required by the template as parameters.
* Data Binding: Data can be passed to EJS templates as a second parameter in the res.render() function. EJS syntax allows easy access to this data within the template.

**Using Partials in EJS**

* Partials Overview: Partials are reusable pieces of code such as headers, footers, or navigation bars that can be included in multiple EJS templates.
* Syntax: To include a partial in an EJS template, use the <%- include('partial\_name') %> syntax.

**Creating a Blogging Application**

* Blogging App Features: The course includes a project to build a blogging application using the MERN stack, providing hands-on experience with EJS.
* Functionalities: The application features user signup, login, and the ability to create, read, update, and delete blogs.
* Advanced Techniques: Learn to implement pagination, filtering, sorting, and other functionalities. The course also covers how to store a copy of the last update to track changes in your blog.

**Best Practices for EJS and Express**

* File Organization: Organize your EJS files into directories, such as 'views' for templates and 'partials' for reusable components.
* Security: Avoid exposing sensitive data in EJS files. Use appropriate server-side data validation and sanitization to prevent cross-site scripting (XSS).
* Error Handling: Implement proper error handling to gracefully handle issues during rendering and data fetching.

**Conclusion**

Understanding View Engines and EJS is crucial for building dynamic, efficient, and maintainable web applications in the MERN stack. By mastering EJS and Express, you can create interactive, responsive user interfaces and improve the overall user experience.