MEAN STACK UNIT-03

NODE.js

What is Node.js:

Node.js is an **open-source**, **cross-platform JavaScript runtime environment built on Chrome's V8 JavaScript engine**. It **allows developers to run JavaScript on the server-side and build fast, scalable, and efficient network applications**. Node.js has a rich set of built-in modules and packages available in the **NPM** (**Node Package Manager**) repository, making it easy for developers to quickly build web applications, APIs, and microservices.

Node.js was **created in 2009 by Ryan Dahl** and has since gained widespread popularity and adoption in the development community. It is **commonly used for building real-time web applications**, such as chat applications and multiplayer games, as well as server-side applications that need to handle large amounts of data and traffic.

One of the key features of Node.js is its non-blocking I/O model, which allows it to handle multiple requests concurrently without blocking the event loop. This makes it particularly well-suited for building applications that require high levels of concurrency and low latency.

Node.js is **also used for serverless computing** and **deploying microservices**, as it can be easily containerized and **run on cloud platforms** such as **AWS Lambda**, **Azure Functions**, and **Google Cloud Functions**.

Why Node.js:

There are several reasons why Node.js is a popular choice for building web applications and server-side services:

- **Speed and scalability**: Node.js is built on Chrome's V8 JavaScript engine, which compiles JavaScript to native machine code, making it incredibly fast and efficient. It is also designed to handle large volumes of traffic and data, and can scale horizontally across multiple servers.
- **Non-blocking I/O**: Node.js uses an event-driven, non-blocking I/O model, which allows it to handle multiple requests concurrently without blocking the event loop.

This makes it particularly well-suited for building real-time applications, such as chat applications and multiplayer games.

- **JavaScript**: Node.js allows developers to use JavaScript on the server-side, which can make it easier to share code between the front-end and back-end of an application. It also means that developers who are familiar with JavaScript can easily transition to building server-side applications with Node.js.
- **Rich ecosystem**: Node.js has a vast ecosystem of open-source modules and packages available in the npm repository, which can significantly speed up development time and make it easier to build complex applications.
- **Serverless computing**: Node.js is well-suited for serverless computing and deploying microservices, as it can be easily containerized and run on cloud platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions.

Overall, Node.js offers a combination of speed, scalability, and flexibility that makes it a popular choice for building modern web applications and server-side services.

How to Use Node.js:

To use Node.js, you need to follow these steps:

- **Install Node.js**: You can download the latest version of Node.js from the official website and install it on your computer.
- **Create a new Node.js project**: Once you have installed Node.js, you can create a new project by creating a new folder and opening it in a terminal or command prompt. Then, run the command "npm init" to initialize a new Node.js project.
- **Install dependencies**: You can use the npm package manager to install dependencies that your project requires. For example, if you want to use the Express web framework, you can run the command "npm install express" to install it.
- **Write code**: You can write your Node.js code in a JavaScript file with a .js extension. You can use any code editor or IDE that supports JavaScript, such as Visual Studio Code, Atom, or Sublime Text.
- **Run your Node.js application**: Once you have written your code, you can run your Node.js application by running the command "node app.js" (assuming your code is in a file named "app.js"). This will start your Node.js application, and you can access it in a web browser at http://localhost:3000 (assuming your application is running on port 3000).

- **Deploy your application**: Once your Node.js application is ready to be deployed, you can deploy it to a hosting service or cloud provider. There are many options available, such as Heroku, AWS, and Google Cloud Platform.

These are the basic steps to use Node.js, but there is much more to learn, such as how to use different Node.js modules, how to write asynchronous code, and how to use databases with Node.js. You can find many resources online, such as tutorials, documentation, and online courses, to help you learn more about Node.js.

Create a web server in Node.js:

To create a web server in Node.js, you can use the built-in HTTP module. Here's an example code to create a basic web server in Node.js:

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

// create a server object

const server = http.createServer((req, res) => {

 // set the response header

 res.writeHead(200, {'Content-Type': 'text/html'});

 // send a response

 res.write('<h1>Hello World!</h1>');

 res.end();

});

// start the server

server.listen(3000, () => {

 console.log('Server started on port 3000');

});
```

Let's break down the code:

- We first import the **http** module using the **require** function.
- We then create a server object using the **createServer** method of the **http** module. This method takes a callback function as an argument, which is executed every time a request is received by the server.
- Inside the callback function, we set the response header using the **writeHead** method of the response object. In this case, we set the status code to 200 (which means the request was successful) and the content type to **text/html**.
- We then send a response to the client using the **write** method of the response object. In this case, we send a simple HTML message saying "Hello World!".
- Finally, we end the response using the **end** method of the response object.
- We start the server using the **listen** method of the server object. In this case, we listen on port 3000 and log a message to the console when the server starts.

You can run this code by saving it to a file named **server.js** and then running **node server.js** in the terminal. You should then be able to access the server in a web browser at http://localhost:3000.

Node Package Manager-NPM:

Node Package Manager (npm) is the default package manager for Node.js. It is used to install and manage packages, which are collections of code and dependencies that can be shared and reused in different Node.js projects.

Here are some key features of npm:

- **Package installation**: You can use npm to install packages from the npm registry or from a local directory. For example, you can run **npm install express** to install the Express web framework.
- **Dependency management**: npm allows you to specify dependencies for your Node.js project in a **package.json** file, which lists all the required packages and their versions. This makes it easy to manage and share your project's dependencies with others.
- **Package publishing**: You can publish your own packages to the npm registry, which can be used by others in their own projects.

- **Versioning**: npm uses SemVer (Semantic Versioning) to manage package versions, which ensures that different versions of the same package can be used in different projects without conflicts.
- **Scripts**: npm allows you to define custom scripts in the **package.json** file, which can be used to automate tasks such as building, testing, and deploying your Node.js application.
- **Security**: npm has built-in security features, such as scanning packages for vulnerabilities and enforcing two-factor authentication for package publishing.

Overall, npm is an essential tool for Node.js developers, as it makes it easy to manage dependencies, reuse code, and collaborate with other developers. You can learn more about npm by reading the official documentation on the npm website.

Modular programming in Node.js:

Modular programming is an important concept in Node.js, as it allows you to break down a complex Node.js application into smaller, more manageable modules that can be developed and tested independently.

In Node.js, you can create a module by creating a new JavaScript file and exporting its functionality using the **module.exports** object. Here's an example of a simple module that exports a function:

```
// myModule.js

function myFunction() {

  console.log('Hello World!');

}

module.exports = myFunction;
```

To use this module in another file, you can use the **require** function to import it:

```
// main.js

const myFunction = require('./myModule');

myFunction(); // logs 'Hello World!'
```

In this example, we define a function **myFunction** in a separate file named **myModule.js**, and export it using the **module.exports** object. We then import this function in another file named **main.js** using the **require** function, and call it to log 'Hello World!' to the console.

You can also export multiple functions or objects from a module using the **module.exports** object. Here's an example of a module that exports two functions:

```
// myModule.js

function function1() {

 console.log('Function 1');

}

function function2() {

 console.log('Function 2');

}

module.exports = {

 function1,

 function2

};
```

To use this module in another file, you can import it like this:

```
// main.js

const { function1, function2 } = require('./myModule');

function1(); // logs 'Function 1'

function2(); // logs 'Function 2'
```

In this example, we define two functions in a separate file named **myModule.js**, and export them as an object using the **module.exports** object. We then import this object and its properties in another file named **main.js**, and call the functions to log their messages to the console.

Modular programming allows you to organize your Node.js application into smaller, reusable components, making it easier to develop, test, and maintain your code.

Restarting Node Application:

When you are developing a Node.js application, you may need to restart the application multiple times to see the changes you've made. There are a few ways to restart a Node.js application:

- **Manually**: You can simply stop the running Node.js process in the terminal by pressing **Ctrl + C**, and then start it again using the **node** command. This method is suitable for small applications or for development purposes.
- **Using nodemon**: nodemon is a tool that monitors changes in your Node.js application and automatically restarts the server when changes are detected. To use nodemon, you need to install it globally using npm:

npm install -g nodemon

Once installed, you can use the **nodemon** command to start your application:

nodemon app.js

Now, whenever you make changes to your application, nodemon will detect them and automatically restart the server.

- **Using PM2**: PM2 is a process manager for Node.js applications that provides advanced features such as automatic restart, monitoring, and logging. To use PM2, you need to install it globally using npm:

npm install -g pm2

Once installed, you can use the **pm2 start** command to start your application:

```
pm2 start app.js
```

PM2 will monitor your application and automatically restart it if it crashes or if changes are detected.

You can also use the **pm2 list** command to see a list of all running applications, and the **pm2 logs** command to view the logs for a specific application.

Overall, there are multiple ways to restart a Node.js application, depending on your needs and preferences. Choose the method that works best for your situation and development workflow.

File Operations:

Node.js provides a built-in module called **fs** (short for file system) that allows you to perform file operations such as reading and writing files. Here are some common file operations that you can perform using the **fs** module in Node.js:

Reading a file:

You can use the **fs.readFile** method to read the contents of a file asynchronously. Here's an example:

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

fs.readFile('file.txt', 'utf8', (err, data) => {

  if (err) throw err;

  console.log(data);

});
```

In this example, we use the **fs.readFile** method to read the contents of a file named **file.txt**. The second argument specifies the encoding of the file (in this case, **utf8**). The callback function is called when the file is read, and it logs the contents of the file to the console.

Writing to a file:

You can use the **fs**.**writeFile** method to write data to a file asynchronously. Here's an example:

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

fs.writeFile('file.txt', 'Hello World!', (err) => {
  if (err) throw err;
  console.log('Data written to file');
});
```

In this example, we use the **fs.writeFile** method to write the string "Hello World!" to a file named **file.txt**. The callback function is called when the data is written to the file, and it logs a message to the console.

Appending to a file:

You can use the **fs.appendFile** method to append data to a file asynchronously. Here's an example:

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

fs.appendFile('file.txt', 'Hello again!', (err) => {
  if (err) throw err;
  console.log('Data appended to file');
});
```

In this example, we use the **fs.appendFile** method to append the string "Hello again!" to a file named **file.txt**. The callback function is called when the data is appended to the file, and it logs a message to the console.

Renaming a file:

You can use the **fs.rename** method to rename a file asynchronously. Here's an example:

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

fs.rename('file.txt', 'newfile.txt', (err) => {
  if (err) throw err;
  console.log('File renamed');
});
```

In this example, we use the **fs.rename** method to rename a file named **file.txt** to **newfile.txt**. The callback function is called when the file is renamed, and it logs a message to the console.

These are just a few examples of file operations that you can perform using the **fs** module in Node.js. The **fs** module provides many other methods that you can use to work with files, such as creating directories, deleting files, and more.

EXPRESS.js

Express.js is a **popular Node.js framework for building web applications** and **APIs**. It **provides a set of features** and **tools to make it easier to build** and **manage web applications**, such as:

- **Routing**: Express provides a simple and flexible way to define routes for your application, allowing you to handle HTTP requests for different URLs and HTTP methods.
- **Middleware**: Express allows you to define middleware functions that can modify request and response objects, such as adding headers, logging, parsing request bodies, and more.
- **Templating**: Express supports a variety of templating engines that allow you to dynamically generate HTML or other types of content based on data from your application.
- **Error handling**: Express provides a robust error handling system that allows you to define error-handling middleware functions to handle errors in a consistent and customizable way.

- **Static file serving**: Express makes it easy to serve static files such as HTML, CSS, and JavaScript files.
- **Database integration**: Express can be easily integrated with various databases, such as MongoDB, MySQL, PostgreSQL, and more, using third-party libraries.

To use Express in your Node.js application, you first need to install it using npm:

```
npm install express
```

Once installed, you can create an Express application by creating an instance of the **express** module:

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

const app = express();
```

You can then define routes for your application using the **app** object:

```
app.get('/', (req, res) => {

  res.send('Hello World!');

});
```

This code defines a route for the root URL (**'/'**) that sends the string "Hello World!" as the response.

Express provides a rich API for defining routes, middleware, and other features. You can learn more about Express and its features in the official documentation: https://expressjs.com/

Express Development Environment:

To set up a development environment for Express.js, you will need to have Node.js and npm (Node Package Manager) installed on your machine. Once you have installed these tools, you can follow these steps to create an Express.js application:

- **Create a new directory for your application and navigate to it in your terminal**:

mkdir my-express-app

cd my-express-app

- **Initialize a new Node.js project in the directory**:

npm init

This will prompt you to answer a few questions about your project and create a **package.json** file in the directory.

- **Install the Express.js package as a dependency of your project**:

npm install express

- **Create a new file called index.js in the directory, and add the following code to it**:

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

const app = express();

app.get('/', (req, res) => {

  res.send('Hello World!');

});

app.listen(3000, () => {

  console.log('Server started on port 3000');

});
```

This code creates a new Express.js application, defines a route for the root URL, and starts the server on port 3000.

- **Start the server by running the following command**:

node index.js

This will start the server and output a message in the console indicating that the server has started.

You can now view your application by opening a web browser and navigating to http://localhost:3000. This will display the "Hello World!" message that you defined in the route.

As you develop your Express.js application, you can use tools like nodemon to automatically restart the server when changes are made to your code. To install nodemon, run the following command:

npm install nodemon --save-dev

You can then start the server with nodemon using the following command:

nodemon index.js

This will start the server and automatically restart it whenever changes are made to your code.

Defining a route:

Defining routes is a core feature of Express.js, and it allows you to handle incoming requests to your application based on the URL and HTTP method of the request. In Express.js, you can define routes using the **app** object, which is an instance of the **express** module.

To define a route in Express.js, you can use the **app.METHOD()** methods, where **METHOD** is the HTTP method of the request (e.g. **GET**, **POST**, **PUT**, **DELETE**, etc.). The basic syntax for defining a route is as follows:

app.METHOD(path, callback);

Here, path is the URL path for the route (e.g. **/users**, **/login**, etc.), and **callback** is a function that is called when the route is matched. The function takes two arguments: **req** (the request object) and **res** (the response object).

Here's an example of how to define a route that handles a **GET** request to the root URL (**/**):

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

In this example, the **app.get()** method is used to define a route for **GET** requests to the root URL (**/**). When a request to the root URL is received, the callback function is called with the **req** and **res** objects, and the response is sent back to the client with the **res.send()** method, which sends the string "Hello World!" as the response.

You can define routes for other HTTP methods using the same syntax. For example, to define a route for a **POST** request to the **/users** URL, you can use the following code:

```
app.post('/users', (req, res) => {
  // Handle POST request to /users
});
```

By defining routes in your Express.js application, you can handle incoming requests and send responses to the client based on the URL and HTTP method of the request.

Handling Routes:

In Express.js, you can define routes using the **app.METHOD()** methods, where **METHOD** is the HTTP method of the request (e.g. **GET**, **POST**, **PUT**, **DELETE**, etc.). Once you have defined a route, you can handle it by defining a callback function that takes two arguments: **req** (the request object) and **res** (the response object).

The **req** object contains information about the incoming request, such as the URL, HTTP headers, and any data sent in the request body. The **res** object is used to send a response back to the client, and it contains methods for setting response headers, sending response data, and more.

Here's an example of how to handle a **GET** request to the root URL (**/**) in Express.js:

```
app.get('/', (req, res) => {

res.send('Hello World!');

});
```

In this example, the **app.get()** method is used to define a route for **GET** requests to the root URL (**/**). When a request to the root URL is received, the callback function is called with the **req** and **res** objects, and the response is sent back to the client with the **res.send()** method, which sends the string "Hello World!" as the response.

You can handle other HTTP methods in the same way. For example, to handle a **POST** request to the **/users** URL, you can use the following code:

```
app.post('/users', (req, res) => {

 // Handle POST request to /users

});
```

In this example, the **app.post()** method is used to define a route for **POST** requests to the **/users** URL. When a **POST** request to the **/users** URL is received, the callback function is called with the **req** and **res** objects, and you can handle the request by adding code inside the callback function.

You can also define routes with URL parameters, which allow you to extract variable data from the URL. For example, to define a route for a **GET** request to a URL with a user ID parameter, you can use the following code:

```
app.get('/users/:id', (req, res) => {

 const userId = req.params.id;

 // Handle GET request to /users/:id

});
```

In this example, the **:id** portion of the URL is a parameter that can be accessed in the callback function using **req.params.id**. This allows you to extract the user ID from the URL and use it in your code to handle the request.

By handling routes in your Express.js application, you can define how your application responds to incoming requests based on the URL and HTTP method of the request.

Route and Query Parameters:

In Express.js, you can define routes with URL parameters and query parameters. URL parameters are defined as part of the URL path and are used to identify a specific resource or data. Query parameters are passed in the URL after the **?** symbol and are used to filter, sort or paginate data.

To define a route with URL parameters in Express.js, you can use the **:** symbol followed by the parameter name in the URL path. For example:

```
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  // Handle request for user with ID userId
});
```

In this example, the **:userId** portion of the URL path is a parameter that can be accessed using **req.params.userId** in the callback function. This allows you to extract the user ID from the URL and use it in your code to handle the request.

To define a route with query parameters, you can use the **req.query** object in the callback function. For example:

```
app.get('/users', (req, res) => {
  const { name, age } = req.query;
  // Handle request with name and/or age query parameters
});
```

In this example, the **req.query** object contains the query parameters passed in the URL. You can extract the values of the **name** and **age** parameters using destructuring, and use them in your code to filter or sort data.

You can also combine URL parameters and query parameters in the same route. For example:

```
app.get('/users/:userId/posts', (req, res) => {
  const { userId } = req.params;
  const { sortBy, limit } = req.query;
  // Handle request for posts by user with ID userId,
  // sorted by sortBy and limited to limit
});
```

In this example, the route is defined with a URL parameter for the **userId** and query parameters for **sortBy** and **limit**. You can extract the values of the parameters using **req.params** and **req.query**, and use them in your code to handle the request.

By using URL parameters and query parameters in your Express.js application, you can define flexible routes that allow users to filter and sort data based on their needs.

How Middleware works:

Middleware functions in Express.js are **functions that have access to the request (req)** and **response (res) objects** and the **next** function in the application's request-response cycle. Middleware functions can execute any code, modify the request and response objects, and end the response cycle. They can also call the **next** function to pass control to the next middleware function in the stack.

Middleware functions are executed in the order they are defined in the application code. When a request is made to the server, the middleware functions are executed one by one in the order they are defined until one of them ends the response cycle by sending a response to the client or the **next** function is called without any argument.

Here is an example of a middleware function that logs the request method and URL to the console:

```
const logMiddleware = (req, res, next) => {
 console.log(`${req.method} ${req.url}`);
 next();
}
```

In this example, **logMiddleware** is a middleware function that logs the request method and URL to the console and calls the **next** function to pass control to the next middleware function in the stack.

You can use middleware functions in your Express.js application to perform a variety of tasks such as logging, authentication, error handling, and more. Middleware functions can be defined globally for the entire application or locally for specific routes or groups of routes.

To define a middleware function globally, you can use the **app.use()** method with the middleware function as an argument. For example:

```
app.use(logMiddleware);
```

In this example, **logMiddleware** is a middleware function that is executed for every request made to the server.

To define a middleware function locally for specific routes or groups of routes, you can use the **app.use()** method with the path and middleware function as arguments. For example:

```
app.use('/users', logMiddleware);
```

In this example, **logMiddleware** is a middleware function that is executed only for requests made to routes that start with **/users**.

By using middleware functions in your Express.js application, you can add custom functionality to the request-response cycle and keep your code organized and modular.

Chaining of Middlewares:

In Express.js, you can chain multiple middleware functions together using the app.use() method to execute them sequentially for a specific route or group of routes. This allows you to create a pipeline of middleware functions that process the request and response objects in a specific order.

To chain middleware functions in Express.js, you can call the app.use() method multiple times with different middleware functions as arguments. For example:

```
app.use((req, res, next) => {
  // First middleware function
  next();
});
app.use((req, res, next) => {
  // Second middleware function
  next();
});
app.use((req, res, next) => {
  // Third middleware function
  res.send('Hello, world!');
});
```

In this example, three middleware functions are defined and chained together using the app.use() method. The first and second middleware functions call the next() function to pass control to the next middleware function in the stack. The third middleware function sends a response to the client using the res.send() method.

You can also chain middleware functions for a specific route or group of routes by passing the path as the first argument to the app.use() method. For example:

```
app.use('/users', (req, res, next) => {
  // First middleware function for '/users' route
  next();
});
app.use('/users', (req, res, next) => {
  // Second middleware function for '/users' route
  next();
});
app.use('/users', (req, res, next) => {
  // Third middleware function for '/users' route
  res.send('Hello, users!');
});
```

In this example, the middleware functions are chained together only for the '/users' route. The request must match the path '/users' for these middleware functions to be executed.

By chaining middleware functions in Express.js, you can create a modular and organized code structure that makes it easy to add or remove functionality to your application.

Types of Middlewares:

In Express.js, there are three types of middleware functions that can be used in an application:

- **Application-level middleware**: This type of middleware is bound to the entire application and is executed for every request made to the server. These middleware

functions can be used for tasks such as logging, authentication, and error handling. Application-level middleware can be defined using the app.use() method or any of the HTTP method-specific routing methods such as app.get(), app.post(), etc.

- **Router-level middleware**: This type of middleware is bound to a specific router instance and is executed only when a request matches the defined router's path. Router-level middleware can be defined using the router.use() method or any of the HTTP method-specific routing methods such as router.get(), router.post(), etc.
- **Error handling middleware**: This type of middleware is used to handle errors that occur during the request-response cycle. Error handling middleware should be defined after all other middleware functions in the stack. These middleware functions can be defined using the app.use() method with four arguments (err, req, res, next).

Here is an example of each type of middleware in Express.js:

```
// Application-level middleware

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

 console.log('Middleware executed for every request.');

 next();

});

// Router-level middleware

const router = express.Router();

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

 console.log('Middleware executed for this router only.');

 next();

});

router.get('/', (req, res) => {

 res.send('Hello from router!');
```

```
});

app.use('/router', router);

// Error handling middleware

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

 console.error(err.stack);

 res.status(500).send('Internal server error.');

});
```

In this example, an application-level middleware logs a message to the console for every request made to the server. A router-level middleware logs a message to the console only when a request matches the defined router's path. An error handling middleware logs the error stack trace to the console and sends an error response to the client.

By using these types of middleware functions in your Express.js application, you can add custom functionality to the request-response cycle and keep your code organized and modular.

Connecting to MongoDB with Mongoose:

To connect to MongoDB with Mongoose in an Express.js application, you first need to install the mongoose package using npm:

```
npm install mongoose
```

Once you have installed mongoose, you can create a connection to your MongoDB database by calling the connect() method on the mongoose object, passing in the connection string and any required options:

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

mongoose.connect('mongodb://localhost/my_database', {

 useNewUrlParser: true,
```

```
  useUnifiedTopology: true
});
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  console.log('Connected to MongoDB database!');
});
```

In this example, we are connecting to a local MongoDB database named "my_database" using the connection string mongodb://localhost/my_database. We are also passing in two options: useNewUrlParser and useUnifiedTopology. These options are required for the latest version of mongoose and will ensure that we use the latest MongoDB driver and avoid deprecation warnings.

After connecting to the database, we are setting up event listeners to handle errors and the successful connection. Once we receive the open event, we log a message to the console indicating that we have successfully connected to the MongoDB database.

Once you have established a connection to your MongoDB database using Mongoose, you can define your data models and use Mongoose to perform CRUD operations on your data.

Validation Types and Defaults:

In Express.js, you can use the express-validator package to handle validation of incoming requests. This package provides various validation types and options that you can use to validate user input.

Here's an example of how to use express-validator to validate a user registration form:

```
const { body, validationResult } = require('express-validator');
app.post('/register', [
```

```
  body('username')
    .notEmpty()
    .withMessage('Username is required.')
    .isLength({ min: 5 })
    .withMessage('Username must be at least 5 characters long.'),
  body('email')
    .notEmpty()
    .withMessage('Email is required.')
    .isEmail()
    .withMessage('Invalid email address.'),
  body('password')
    .notEmpty()
    .withMessage('Password is required.')
    .isLength({ min: 8 })
    .withMessage('Password must be at least 8 characters long.')
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  // Code to create user account  });
```

In this example, we are using the body() method from express-validator to define validation rules for each field in the user registration form. We are using various validation methods such as notEmpty(), isLength(), and isEmail() to ensure that the user input meets our requirements. If any validation errors occur, we return a 400 Bad Request response with the validation errors in the response body.

In addition to validation types, express-validator also provides options for defining default values for fields and sanitizing user input. For example, you can use the default() method to set a default value for a field if it is not provided in the request:

```
body('language').default('en')
```

You can also use the trim() method to remove whitespace from user input, and the escape() method to escape HTML characters to prevent XSS attacks:

```
body('username').trim().escape()
```

By using express-validator and its various validation types and options, you can ensure that your application is handling user input securely and reliably.

Models:

In an Express.js application, a model represents a structured definition of a data entity that you can use to interact with a database. A model can define properties and methods that represent the data and behavior of the entity.

There are several popular Node.js libraries that you can use to define models in your Express.js application, such as mongoose and sequelize.

Here's an example of how to define a model using mongoose in an Express.js application:

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

const userSchema = new mongoose.Schema({

  name: { type: String, required: true },

  email: { type: String, required: true, unique: true },
```

```
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});
const User = mongoose.model('User', userSchema);
module.exports = User;
```

In this example, we are defining a User model using mongoose. The userSchema defines the properties of a user entity, including the name, email, password, and createdAt fields. We are also using mongoose validators to enforce required fields and unique email addresses.

After defining the schema, we create a User model using mongoose.model(). The first argument to this method is the name of the model, and the second argument is the schema that we defined earlier.

Finally, we export the User model so that it can be used in other parts of our application.

With the User model defined, we can use mongoose methods to perform CRUD operations on the User collection in the database. For example, to create a new user, we can use the following code:

```
const user = new User({
  name: 'John Doe',
  email: 'john.doe@example.com',
  password: 'mysecretpassword'
});
user.save((err, user) => {
  if (err) {
 console.error(err);
```

```
  return;
 }
 console.log(user);
});
```

In this example, we create a new User instance and call the save() method to insert the user into the database. If an error occurs, we log the error to the console, and if the user is saved successfully, we log the saved user to the console.

CRUD Operations:

CRUD operations (Create, Read, Update, Delete) are common database operations that are used in web applications. In an Express.js application, you can perform CRUD operations using various database libraries, such as mongoose or sequelize.

Here's an example of how to perform CRUD operations using mongoose in an Express.js application:

Create

To create a new document in the database, you can create a new instance of the User model and call the save() method:

```
const User = require('./models/user');
const user = new User({
 name: 'John Doe',
 email: 'john.doe@example.com',
 password: 'mysecretpassword'
});
user.save((err, user) => {
```

```
  if (err) {
    console.error(err);
    return;
  }
  console.log(user);
});
```

Read

To read data from the database, you can use the find() method on the User model:

```
const User = require('./models/user');
User.find((err, users) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(users);
});
```

This will return all documents in the users collection. You can also use the findOne() method to find a single document:

```
const User = require('./models/user');
User.findOne({ email: 'john.doe@example.com' }, (err, user) => {
  if (err) {
```

```
    console.error(err);
    return;
  }
  console.log(user);
});
```

Update

To update an existing document in the database, you can use the updateOne() method on the User model:

```
const User = require('./models/user');

User.updateOne({ email: 'john.doe@example.com' }, { name: 'Jane Doe' }, (err, result) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(result);
});
```

This will update the name property of the document with the email address john.doe@example.com.

Delete

To delete a document from the database, you can use the deleteOne() method on the User model:

```
const User = require('./models/user');

User.deleteOne({ email: 'john.doe@example.com' }, (err, result) => {
 if (err) {
  console.error(err);
  return;
 }
 console.log(result);
});
```

This will delete the document with the email address john.doe@example.com.

API Development:

API development is a common use case for Express.js. In this context, the application acts as a server that provides access to a collection of resources via HTTP endpoints. Here's an example of how to develop an API in Express.js:

Defining endpoints

To define an endpoint, you need to specify a URL pattern and an HTTP method that the endpoint should respond to. You can do this using the app.get(), app.post(), app.put(), app.patch(), and app.delete() methods, depending on the HTTP method you want to use.

For example, to define an endpoint that responds to a GET request to the /api/users URL, you can use the following code:

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

const app = express();

app.get('/api/users', (req, res) => {
 // Get all users from the database and send them as a JSON response
```

```
  res.json(users);
});
```

Handling requests

When a request is received at an endpoint, you need to process the request and generate a response. You can do this using middleware functions.

For example, to add a middleware function that logs incoming requests, you can use the following code:

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

This middleware function logs the HTTP method and URL of incoming requests and passes control to the next middleware function using the next() function.

Parsing request data

To parse data from a request body or query string, you can use middleware functions such as express.json() and express.urlencoded().

For example, to parse JSON data in the request body, you can use the following code:

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

Sending responses

To send a response back to the client, you can use the res.send(), res.json(), and res.status() methods.

For example, to send a JSON response with a status code of 200 (OK), you can use the following code:

```
app.get('/api/users', (req, res) => {
  // Get all users from the database
  const users = getUsers();
  // Send the users as a JSON response with a status code of 200
  res.status(200).json(users);
});
```

Error handling

To handle errors that occur during request processing, you can define error handling middleware functions using the app.use() method.

For example, to define an error handling middleware function that logs errors to the console and sends an error response to the client, you can use the following code:

```
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('An error occurred');
});
```

This middleware function logs the error to the console and sends a 500 (Internal Server Error) response to the client.

Why Session management:

Session management is an important aspect of web development, especially in the context of user authentication and authorization. In Express.js, session management is used to maintain user state across requests and to implement features such as user login, user logout, and user-specific content.

When a user logs in to a web application, a session is created that identifies the user and stores their session data, such as their user ID and authentication status. The session data is

stored on the server and is associated with a session ID, which is sent to the client in a cookie. The cookie is sent with every subsequent request from the client, allowing the server to identify the user and retrieve their session data.

There are several benefits to using session management in Express.js:

- **User authentication**: Session management is used to authenticate users and to restrict access to protected resources based on user permissions.
- **User-specific content**: Session management is used to display user-specific content, such as a user's profile or their order history.
- **Session timeouts**: Session management can be used to enforce session timeouts, which automatically logs a user out after a period of inactivity to prevent unauthorized access.
- **Security**: Session management is used to prevent session hijacking and other security threats by encrypting session data and using secure cookies.

In Express.js, session management can be implemented using middleware functions such as express-session and cookie-parser. These middleware functions handle the creation and management of sessions, including session data storage, session ID generation, and cookie handling.

Cookies:

Cookies are small pieces of data that are stored on the client-side (usually in the browser) and are sent back to the server with each subsequent request. Cookies are often used for authentication and to store user preferences or session data. In Express.js, cookies can be set and retrieved using the cookie-parser middleware.

To use cookie-parser, you need to first install it using npm:

npm install cookie-parser

Then, in your Express.js application, you can use cookie-parser as middleware to parse cookies from the request and attach them to the request object:

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

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

```
const app = express();

app.use(cookieParser());
```

Once cookie-parser is set up as middleware, you can set cookies using the res.cookie() method. This method takes two arguments: the name of the cookie and its value. You can also provide additional options, such as the cookie's expiration date and whether it should be secure or not:

```
app.get('/set-cookie', (req, res) => {
  res.cookie('myCookie', 'hello world', { maxAge: 900000, httpOnly: true });
  res.send('Cookie set!');
});
```

In the example above, we set a cookie called myCookie with a value of hello world. We also set an expiration time of 900000 milliseconds (15 minutes) and the httpOnly flag to true, which means the cookie can only be accessed through HTTP(S) requests and not through JavaScript.

You can retrieve cookies from the request object using req.cookies. This property contains an object with key-value pairs for each cookie sent with the request:

```
app.get('/get-cookie', (req, res) => {
  const myCookie = req.cookies.myCookie;
  res.send(`Cookie value: ${myCookie}`);
});
```

In the example above, we retrieve the value of the myCookie cookie from the req.cookies object and send it as a response.

Overall, cookies are a useful tool for maintaining state and session data in Express.js applications. The cookie-parser middleware makes it easy to set and retrieve cookies from requests and responses.

Sessions:

Sessions are a way to store data on the server-side between requests. In Express.js, session management can be implemented using middleware such as express-session.

To use express-session, you need to first install it using npm:

```
npm install express-session
```

Then, in your Express.js application, you can use express-session as middleware to set up session management:

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

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

const app = express();

app.use(session({

 secret: 'mySecret',

 resave: false,

 saveUninitialized: false

}));
```

In the example above, we set up the express-session middleware with a secret key, mySecret, to sign the session ID cookie. We also set the resave and saveUninitialized options to false, which means the session data will not be saved to the session store unless there is a change to the data.

Once express-session is set up as middleware, you can store data in the session using the req.session object. This object is unique to each user's session and can be accessed and modified like a regular JavaScript object:

```
app.get('/set-session', (req, res) => {

 req.session.myData = 'hello world';
```

```
  res.send('Session data set!');

});
```

In the example above, we set a value called myData in the user's session using the req.session object.

You can retrieve data from the session object using dot notation, just like with any JavaScript object:

```
app.get('/get-session', (req, res) => {

  const myData = req.session.myData;

  res.send(`Session data: ${myData}`);

});
```

In the example above, we retrieve the value of myData from the user's session and send it as a response.

Overall, session management is an important part of many web applications, and express-session makes it easy to set up and use sessions in Express.js.

Why and What Security:

Security is a critical aspect of any web application, and Express.js provides several security features to help developers build secure applications.

Some of the reasons why security is important in Express.js are:

- **Protection against attacks**: Web applications can be vulnerable to various attacks such as cross-site scripting (XSS), SQL injection, cross-site request forgery (CSRF), etc. Express.js provides built-in security features to protect against these attacks.
- **Compliance**: Many web applications are required to comply with security regulations such as HIPAA, GDPR, etc. Express.js provides features that can help ensure compliance with these regulations.

- **Reputation**: A security breach can damage the reputation of an application and the organization behind it. It is important to ensure that the application is secure to maintain the reputation of the organization.

Some of the security features provided by Express.js are:

- **Helmet**: Helmet is a middleware that helps secure Express.js applications by setting various HTTP headers to protect against common web vulnerabilities such as XSS, clickjacking, etc.
- **Validation and sanitization**: Express.js provides several modules for validating and sanitizing user input to prevent attacks such as SQL injection, XSS, etc. These modules include express-validator, joi, and sanitize-html.
- **Authentication and authorization**: Express.js provides several modules for implementing authentication and authorization, such as passport, jsonwebtoken, etc. These modules can help protect against attacks such as CSRF, session hijacking, etc.
- **Rate limiting**: Express.js provides middleware such as express-rate-limit that can help protect against DDoS attacks by limiting the number of requests from a particular IP address.
- **Secure cookies**: Cookies can be vulnerable to attacks such as CSRF, XSS, etc. Express.js provides middleware such as cookie-parser and cookie-session to help secure cookies.

Overall, security is an essential aspect of any web application, and it is important to ensure that the application is secure to protect against attacks and comply with regulations. Express.js provides several security features to help developers build secure applications.

Helmet Middleware:

Helmet is a middleware for Express.js that helps secure web applications by setting various HTTP headers. It provides a suite of middleware functions that can be used to protect against common web vulnerabilities such as cross-site scripting (XSS), clickjacking, and cross-site request forgery (CSRF).

To use Helmet in your Express.js application, you first need to install it as a dependency:

npm install helmet

Then, you can use it as middleware in your application by adding the following line:

const helmet = require('helmet');

app.use(helmet());

This will set various HTTP headers to improve the security of your application. Here are some of the headers that Helmet can set:

- **X-XSS-Protection**: This header helps protect against cross-site scripting (XSS) attacks by enabling the browser's XSS filter.
- **X-Content-Type-Options**: This header helps protect against MIME-type sniffing attacks by disabling the browser's ability to guess the MIME type of a response.
- **X-Frame-Options**: This header helps protect against clickjacking attacks by preventing a page from being loaded in an iframe.
- **Content-Security-Policy**: This header helps protect against various attacks such as XSS, code injection, etc. by defining a whitelist of sources that are allowed to execute scripts, styles, etc.
- **Strict-Transport-Security**: This header helps protect against man-in-the-middle (MITM) attacks by enforcing the use of HTTPS.

By using Helmet in your Express.js application, you can improve the security of your application by setting these headers and protecting against common web vulnerabilities.

Using a Template Engine Middleware:

Express.js provides several template engines that you can use to generate dynamic HTML pages. Some popular template engines for Express.js include EJS, Pug (formerly Jade), Handlebars, and Mustache.

To use a template engine in your Express.js application, you first need to install it as a dependency:

npm install ejs

Then, you need to configure your application to use the template engine. Here's an example of how to configure EJS as the template engine for your Express.js application:

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

const app = express();

// set the view engine to ejs

app.set('view engine', 'ejs');

// define a route that renders an ejs template

app.get('/', function(req, res) {

  res.render('index', { title: 'Express.js with EJS' });

});

app.listen(3000, function() {

  console.log('Server started on port 3000');

});
```

In this example, we're setting the view engine to EJS using app.set('view engine', 'ejs'). Then, we're defining a route that renders an EJS template using res.render('index', { title: 'Express.js with EJS' }). The res.render method takes two arguments: the name of the template file to render (without the file extension), and an object containing data that will be passed to the template.

Once you have your template engine set up, you can create dynamic HTML pages by using template tags and logic. Here's an example of an EJS template that uses a variable passed in from the route handler:

```
<!DOCTYPE html>

<html>

<head>

  <title><%= title %></title>
```

```
</head>

<body>

  <h1><%= title %></h1>

  <p>Welcome to <%= title %></p>

</body>

</html>
```

In this template, we're using EJS template tags (<%= %>) to insert the value of the title variable into the HTML page. This will be rendered as "Express.js with EJS" when the page is displayed in the browser.

Using a template engine in your Express.js application can simplify the process of generating dynamic HTML pages and make your code more maintainable.

Stylus CSS Preprocessor:

Stylus is a popular CSS preprocessor that can be used with Express.js to simplify the process of writing CSS stylesheets. Stylus provides a range of features that can help you write cleaner, more organized CSS code, such as variables, mixins, and nested selectors.

To use Stylus in your Express.js application, you first need to install it as a dependency:

```
npm install stylus
```

Then, you need to configure your application to use Stylus as a middleware. Here's an example of how to do this:

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

const app = express();

// set up the stylus middleware

app.use(require('stylus').middleware(__dirname + '/public'));
```

```
// serve static files from the public directory
app.use(express.static(__dirname + '/public'));
// define a route that serves an HTML page with a Stylus stylesheet
app.get('/', function(req, res) {
  res.render('index', { title: 'Express.js with Stylus' });
});
app.listen(3000, function() {
  console.log('Server started on port 3000');
});
```

In this example, we're setting up the Stylus middleware using app.use(require('stylus').middleware(__dirname + '/public')). This tells Express.js to use Stylus to process any CSS files requested from the /public directory.

Then, we're serving static files from the public directory using express.static(__dirname + '/public'). This will allow us to reference the Stylus stylesheet in our HTML page.

Finally, we're defining a route that serves an HTML page with a Stylus stylesheet using res.render('index', { title: 'Express.js with Stylus' }). The res.render method takes two arguments: the name of the template file to render (without the file extension), and an object containing data that will be passed to the template.

Once you have Stylus set up in your application, you can write your stylesheets using Stylus syntax. Here's an example of a Stylus stylesheet that uses variables and nested selectors:

```
// define some variables
$primary-color = #0099ff
$secondary-color = #ff9900
// define some styles
```

```
body
  font-family Arial, sans-serif
  font-size 16px
header
  background-color $primary-color
  color white
  h1
    font-size 24px
button
  background-color $secondary-color
  color white
  padding 10px 20px
  border none
```

In this stylesheet, we're defining two variables ($primary-color and $secondary-color) that can be used throughout the stylesheet. We're also using nested selectors to apply styles to the h1 element inside the header element, and to the button element.

THANK YOU

HAPPY LEARNING