What is Express.js?

Express.js is a popular Node.js web application framework that simplifies the process of building robust, scalable, and efficient web applications. It provides a set of features that allow developers to easily manage middleware, routing, templates, and server-side logic.

What is middleware in Express.js?

Middleware in Express.js is a function that receives the HTTP request and response objects and performs some action on them before passing them on to the next middleware function in the chain. Middleware functions can be used for a wide range of tasks such as logging, authentication, handling errors, and more.

What is routing in Express.js?

Routing in Express.js is the process of defining the URL endpoints of an application and associating them with the appropriate server-side logic. This involves creating routes that correspond to specific HTTP methods and URLs, and defining route handlers that implement the required functionality for each route.

What is a route handler in Express.js?

A route handler in Express.js is a function that is invoked when a specific route is matched during an incoming request. It receives the request and response objects as arguments, and performs the necessary logic to generate the response, such as querying a database, rendering a template, or returning JSON data.

How do you set up a basic Express.js application?

Here is an example of setting up a basic Express.js application:

```
Copy code
const express = require('express')
const app = express()

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

app.listen(3000, () => {
console.log('Server listening on port 3000')
})
```

This creates an Express.js app, defines a single route that responds with "Hello World!" when accessed via the root URL, and starts the app listening on port 3000.

How do you serve static files in Express.js?

To serve static files in Express.js, you can use the built-in express.static() middleware function. This function takes a path to the directory containing the static files, and returns a middleware function that serves the files in that directory.

Here is an example:

```
Copy code
app.use(express.static('public'))
```

This tells Express.js to serve static files from the public directory.

How do you handle POST requests in Express.js?

To handle POST requests in Express.js, you can use the app.post() method to define a route handler that responds to POST requests. The route handler can access the data sent in the request body using the req.body property.

Here is an example:

```
Copy code
app.post('/login', (req, res) => {
const { username, password } = req.body
// Check if username and password are valid
// Generate JWT token and send it in response
})
```

This defines a route that responds to POST requests to the /login URL, extracts the username and password from the request body, and performs the required logic.

What is the purpose of the body-parser middleware in Express.js?

The body-parser middleware in Express.js is used to parse the request body and make it available on the req.body property. It supports parsing various types of request data, such as JSON, URL-encoded data, and raw text.

Here is an example of using the body-parser middleware:

```
Copy code
const bodyParser = require('body-parser')

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }))
```

This tells Express.js to use

What is the purpose of the cookie-parser middleware in Express.js?

The cookie-parser middleware in Express.js is used to parse cookies sent in the request and make them available on the req.cookies property. It supports signing cookies with a secret key to prevent tampering.

Here is an example of using the cookie-parser middleware:

```
Copy code
const cookieParser = require('cookie-parser')

app.use(cookieParser('my-secret-key'))
```

This tells Express.js to use the cookie-parser middleware and sign cookies with the secret key "my-secret-key".

What is the purpose of the cors middleware in Express.js?

The cors middleware in Express.js is used to enable Cross-Origin Resource Sharing (CORS) for an application. CORS is a security feature implemented by web browsers that restricts web pages from making requests to a different domain than the one that served the web page.

Here is an example of using the cors middleware:

```
Copy code
const cors = require('cors')

app.use(cors())
```

This tells Express.js to use the cors middleware and allow requests from any origin.

How do you handle errors in Express.js?

In Express.js, you can handle errors by defining an error-handling middleware function. This function takes four arguments: err, req, res, and next. It should check the type of error and return an appropriate response or pass the error to the next middleware function.

Here is an example of defining an error-handling middleware function:

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

This defines a middleware function that logs the error stack trace to the console and responds with a 500 status code and error message.

What is the purpose of the next function in Express.js middleware?

The next function in Express.js middleware is used to pass control to the next middleware function in the chain. It is typically called at the end of a middleware function to allow processing to continue.

Here is an example of using the next function in middleware:

```
```

```
Copy code
app.use((req, res, next) => {
console.log('Request received')
next()
})
```

This defines a middleware function that logs a message to the console and passes control to the next middleware function.

How do you handle asynchronous operations in Express.js?

In Express.js, you can handle asynchronous operations using callback functions, promises, or async/await syntax. The most common way is to use promises, which allow you to chain multiple asynchronous operations together.

Here is an example of using promises to handle asynchronous operations in Express.js:

```
Copy code
app.get('/users', (req, res) => {
User.find()
.then(users => {
res.json(users)
})
.catch(err => {
console.error(err)
res.status(500).send('Something went wrong!')
})
})
```

This defines a route that retrieves all users from a database using a User.find() function that returns a promise. If the promise resolves successfully, the route sends a JSON response with the users. If the promise is rejected, the route logs an error to the console and sends a 500 status code with an error message.

What is the purpose of the app.listen method in Express.js?

The app.listen method in Express.js is used to start the application server and listen for incoming HTTP requests on a specified port. It takes a port number and an optional callback function that is called when the server is ready to start accepting requests.

Here is an example of using the app.listen method:

```
Copy code
const port = process.env.PORT || 3000

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

This starts the application server and listens for incoming requests on port 3000 (or the port specified by the PORT environment variable). When the server is ready to start accepting requests, it logs a message to the console.

What is middleware in Express.js?

Middleware in Express.js is a function that can process incoming requests and perform actions before passing the request on to the next middleware function or final route handler. Middleware functions can modify the request and response objects, end the request-response cycle, or call the next middleware function in the chain.

Here is an example of defining and using middleware in Express.js:

```
Copy code
app.use((req, res, next) => {
console.log('Request received')
next()
})

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

This defines a middleware function that logs a message to the console and passes control to the next middleware function or final route handler. The route handler sends a "Hello World!" response to the client.

What is the purpose of the router object in Express.js?

The router object in Express.js is used to define routes for a specific path prefix. It allows you to group related routes together and define middleware that is specific to those routes.

Here is an example of defining and using a router object in Express.js:

```
Copy code
const router = express.Router()

router.use((req, res, next) => {
console.log('Request received')
next()
})

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

app.use('/api', router)
```

This defines a router object that logs a message to the console and sends a "Hello World!" response to the client when the client requests the /api/ route. The router object is mounted to the /api path prefix in the

Express.js application using the app.use method.

PROMISES Promises are a feature in JavaScript that allow you to handle asynchronous operations in a more organized and structured way. They are objects that represent the eventual completion (or failure) of an asynchronous operation, such as a network request or a database query.

A promise has three states: pending, fulfilled, or rejected. When a promise is in the pending state, it means that the operation has not yet completed. When a promise is fulfilled, it means that the operation was successful and the promise returns a value. When a promise is rejected, it means that the operation failed and the promise returns an error.

Promises can be chained together using the then() method, which takes two arguments: a success handler function and an error handler function. If the promise is fulfilled, the success handler function is called with the result of the operation. If the promise is rejected, the error handler function is called with the reason for the failure.

Promises provide a cleaner and more organized way to handle asynchronous operations compared to traditional callback functions, as they avoid the so-called "callback hell" problem and allow you to handle success and error cases separately.