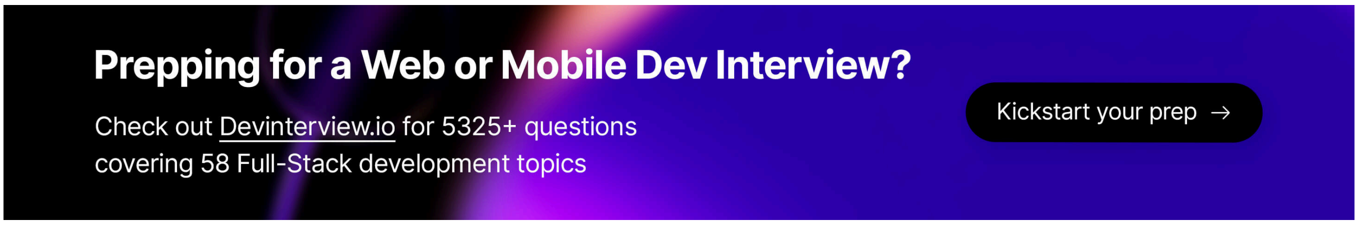
Top 58 Express.js Interview Questions in 2025.


Prepping for a Web or Mobile Dev Interview?

Check out Devinterview.io for 5325+ questions covering 58 Full-Stack development topics

Kickstart your prep →

You can also find all 58 answers here 👉 Devinterview.io - Express.js

1. What is *Express.js*, and how does it relate to *Node.js*?

Express.js is a web application framework that runs on **Node.js**. It simplifies the process of building web applications and APIs by providing a range of powerful features, including robust routing, middleware support, and HTTP utility methods. Thanks to its modular design, you can expand its functionality through additional libraries and Node.js modules.

Key Features

- **Middleware**: Express.js makes use of middleware functions that have access to the request-response cycle. This allows for a variety of operations such as logging, authentication, and data parsing.
- **Routing**: The framework offers a flexible and intuitive routing system, making it easy to handle different HTTP request methods on various URLs.
- **Templates**: Integrated support for template engines enables the dynamic rendering of HTML content.
- **HTTP Methods**: It provides built-in methods for all HTTP requests, such as `get` , `post` , `put` , `delete` , simplifying request handling.
- **Error Handling**: Express streamlines error management, and its middleware functions can specifically handle errors.
- **RESTful APIs**: Its features such as request and response object chaining, along with HTTP method support, make it ideal for creating RESTful APIs.

Relationship with Node.js

Express.js is a web application framework specifically designed to extend the capabilities of **Node.js** for web development. Node.js, on the other hand, is a cross-platform JavaScript runtime environment that allows developers to build server-side and networking applications.

Express.js accomplishes this through a layer of abstractions and a more structured approach, which Node.js, by itself, doesn't provide out of the box.

Code Example: Basic Express Server

Here is the Node.js code:

```javascript
// Import required modules
const express = require('express');

// Create an Express application
const app = express();
const port = 3000;

// Define a route and its callback function
app.get('/', (req, res) => {
  res.send('Hello World!');
});

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

2. Explain the concept of *middleware* in *Express.js*.

Middleware acts as a bridge between incoming HTTP requests and your Express.js application, allowing for a range of operations such as parsing request bodies, handling authentication, and even serving static files.

Middleware Functions

A middleware function in Express is a **handler invoked in sequence** when an HTTP request is received. It has access to the request and response objects, as well as the `next` function to trigger the next middleware in line.

Each middleware function typically follows this signature:

```javascript
function middlewareFunction(req, res, next) {
    // ...middleware logic
    next(); // or next(err); based on whether to proceed or handle an error
}
```

Note that the `next()` call is essential to move on to the next middleware.

Types of Middleware

Application-Level Middleware

Registered via `app.use(middlewareFunction)`, it's active for every incoming request, making it suitable for tasks like request logging or establishing cross-cutting concerns.

Router-Level Middleware

Operates on specific router paths and is defined using `router.use(middlewareFunction)`. It's useful for tasks related to particular sets of routes.

Error-Handling Middleware

Recognizable via its function signature `(err, req, res, next)`, this type of middleware specifically handles errors. In the middleware chain, it should be placed after regular middlewares and can be added using `app.use(function(err, req, res, next) { ... })`.

Built-In Middleware

Express offers ready-to-use middleware for tasks like serving static files or parsing the request body.

Middleware Chaining

By **sequentially** calling `next()` within each middleware, you form a chain, facilitating a cascade of operations for an incoming request.

Consider a multi-tiered security setup, for example, with authentication, authorization, and request validation. Only when a request passes through all three tiers will it be processed by the actual route handler.

Code Example: Middleware Chaining

Here is the code:

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

// Sample middleware functions
function authenticationMiddleware(req, res, next) {
    console.log('Authenticating...');
    next();
}

function authorizationMiddleware(req, res, next) {
    console.log('Authorizing...');
    next();
}

function requestValidationMiddleware(req, res, next) {
    console.log('Validating request...');
    next();
}

// The actual route handler
app.get('/my-secured-endpoint', authenticationMiddleware, authorizationMiddleware, requestValidationMiddleware, (req,
    res.send('Welcome! You are authorized.');
});

app.listen(3000);
```

3. How would you set up a basic *Express.js* application?

To set up a **basic Express.js** application, follow these steps:

1. Initialize the Project

Create a new directory for your project and run `npm init` to generate a `package.json` file.

2. Install Dependencies

Install **Express** as a dependency using the Node Package Manager (NPM):

```
npm install express
```

3. Create the Application

In your project directory, create a main file (usually named `app.js` or `index.js`) to set up the Express application.

Here is the JavaScript code:

```
// Import the Express module
const express = require('express');

// Create an Express application
const app = express();

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

// Start the server
const port = 3000;
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

4. Run the Application

You can start your Express server using Node.js:

```
node app.js
```

For convenience, you might consider using **Nodemon** as a development dependency which automatically restarts the server upon file changes.

4. What is the purpose of the `app.use()` function?

In Express.js, the `app.use()` function is a powerful tool for **middleware management**. It can handle HTTP requests and responses, as well as prepare data or execute processes in between.

Key Functions

- **Global Middleware**: Without a specified path, the middleware will process every request.
- **Route-specific Middleware**: When given a path, the middleware will only apply to the matched routes.

Common Use-Cases

- **Body Parsing**: To extract data from incoming requests, especially useful for POST and PUT requests.

```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

- **Handling CORS**: Useful in API applications to manage cross-origin requests.

```
app.use(function(req, res, next) {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
    next();
});
```

- **Static File Serving**: For serving files like images, CSS, or client-side JavaScript.

```
app.use(express.static('public'));
```

- **Logging**: To record request details for debugging or analytics.

```
app.use(function(req, res, next) {
    console.log(`${new Date().toUTCString()}: ${req.method} ${req.originalUrl}`);
    next();
});
```

- **Error Handling**: To manage and report errors during request processing.

```
app.use(function(err, req, res, next) {
    console.error(err);
    res.status(500).send('Internal Server Error');
});
```

Chaining Middleware

You can **stack multiple middleware** using `app.use()` in the order they need to execute. For a matched route, control can be passed to the next matching route or terminated early using `next()` .

5. How do you serve *static files* using *Express.js*?

In an Express.js **web application**, you often need to **serve static files** such as stylesheets, client-side JavaScript, and images. You can accomplish this using the `express.static` middleware.

Middleware for Serving Static Files

The `express.static` middleware function serves static files and is typically used to serve assets like images, **CSS**, and **client-side JavaScript**.

Here is the code example:

```
app.use(express.static('public'));
```

In this example, the folder named `public` will be used to serve the static assets.

Additional Configuration with Method Chaining

You can further configure the behavior of the `express.static` middleware by chaining methods.

For example, to set the cache-control header, the code looks like this:

```
app.use(express.static('public', {
    maxAge: '1d'
}));
```

Here, the `'1d'` ensures that caching is enabled for a day.

Using a Subdirectory

If you want to serve files from a subdirectory, you can specify it when using the `express.static` middleware.

Here is the code example:

```
app.use('/static', express.static('public'));
```

This serves the files from the `public` folder but any requests for these files should start with `/static` .

What `express.static` Serves

- **Images**: PNG, JPEG, GIF
- **Text Content**: HTML, CSS, JavaScript
- **Fonts**
- **JSON Data**

Not for dynamic content

While `express.static` is excellent for **static assets**, it's not suitable for dynamic content or data in **POST** requests.

6. Discuss the difference between `app.get()` and `app.post()` in *Express.js*.

In **Express.js**, `app.get()` and `app.post()` are two of the most commonly used HTTP method middleware. The choice between them (or using both) typically depends on whether you are **retrieving** or **submitting/persisting** data.

Key Distinctions

HTTP Verbs: External Visibility

- **app.get()**: Listens for GET requests. Designed for data retrieval. Visible URLs typically trigger such requests (e.g., links or direct URL entry in the browser).

- **app.post()**: Listens for POST requests. Intended for data submission. Typically not visible in the URL bar, commonly used for form submissions.

Data Transmission

- **app.get()**: Uses query parameters for data transmission, visible in the URL. Useful for simple, non-sensitive, read-only data (e.g., filtering or pagination).

- **app.post()**: Uses request body for data transmission, which can be in various formats (e.g., JSON, form data). Ideal for more complex data, file uploads, or sensitive information.

Using Both `app.get()` and `app.post()` for the Same Route

There are cases, especially for **RESTful** design, where a single URL needs to handle both data retrieval and data submission.

- **Resource Retrieval and Creation**:

- **Fetch a Form**: Use `app.get()` to return a form for users to fill out.
- **Form Submission**: Use `app.post()` to process and save the submitted form data.

- **Complete Entity Modification**: For a complete update (or replacement in REST), using `app.post()` ensures that the update action is triggered via a post request, not a get request. This distiction is important to obey the RESTful principles.

Code Example: Using both `app.get()` and `app.post()` for a single route

Here is the JavaScript code:

```
const userRecords = {}; // in-memory "database" for the sake of example

// Handle user registration form
app.get('/users/register', (req, res) => {
    res.send('Please register: <form method="POST"><input name="username"></form>');
});

// Process submitted registration form
app.post('/users/register', (req, res) => {
    userRecords[req.body.username] = req.body;
    res.send('Registration complete');
});
```

7. How do you retrieve the *URL parameters* from a *GET request* in *Express.js*?

In **Express.js**, you can extract **URL parameters** from a **GET** request using the `req.params` object. Here's a quick look at the steps and the code example:

Code Example: Retrieving URL Parameters

```
// Sample URL: http://example.com/users/123
// Relevant Route: /users/:id

// Define the endpoint/route
app.get('/users/:id', (req, res) => {
    // Retrieve the URL parameter
    const userId = req.params.id;
    // ... (rest of the code)
});
```

In this example, the URL parameter `id` is extracted and used to fetch the corresponding user data.

Additional Steps for Complex GET Requests

For simple and straightforward **GET** requests, supplying URL parameters directly works well. However, for more complex scenarios, such as parsing parameters from a URL with the help of `querystrings` or handling optional parameters, **Express.js** offers more advanced techniques which are outlined below:

Parsing Query Parameters

- **What It Is**: Additional data passed in a URL after the `?` character. Example: `http://example.com/resource?type=user&page=1`.

- **How to Access It**: Use `req.query`, an object that provides key-value pairs of the parsed query parameters.

Code Example: Parsing Query Parameters

```
app.get('/search', (req, res) => {
    const { q, category } = req.query;
    // ... (rest of the code)
});
```

Optional and Catch-All Segments

- **Optional Segments**: URL segments enclosed in parentheses are optional and can be accessed using `req.params`. Example: `/book(/:title)`

- **Catch-All Segments**: Captures the remainder of the URL and is useful in cases like URL rewriting. Denoted by an asterisk (`*`) or double asterisk (`**`). Accessed using `req.params` as well. Example: `/documents/*`

8. What are *route handlers*, and how would you implement them?

Route handlers in Express.js are middleware functions designed to manage specific paths in your application.

Depending on the HTTP method and endpoint, they can perform diverse tasks, such as data retrieval from a database, view rendering, or HTTP response management.

Code Example: Setting Up a Simple Route Handler

Here is the code:

```
// Responds with "Hello, World!" for GET requests to the root URL (/)
app.get('/', (req, res) => {
  res.send('Hello, World!');
});
```

In this example, the route handler is `(req, res) => { res.send('Hello, World!'); }`. It listens for GET requests on the root URL and responds with "Hello, World!".

What Are Route-Handler Chains?

You can associate numerous route-managing **middleware functions** to a single route. Every middleware function in the chain has to either proceed to the following function using `next()` or conclude the request-response cycle.

This allows for checks like user authentication before accessing a route.

HTTP Method Convenience Methods

Express.js offers specialized, highly-readable methods for the most common HTTP requests:

- `app.get()`
- `app.post()`
- `app.put()`
- `app.delete()`
- `app.use()`

These methods streamline route handling setup.

9. How do you enable *CORS* in an *Express.js* application?

Cross-Origin Resource Sharing (CORS) is a mechanism that allows web pages to make requests to a different domain. In Express.js, you can enable CORS using the `cors` package or by setting headers manually.

Using the `cors` Package

1. **Install `cors`**:

   Use npm or yarn to install the `cors` package.

   ```
   npm install cors
   ```

2. **Integrate with Your Express App**:

   Use the `app.use(cors())` middleware. You can also customize CORS behavior with options.

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

   // Enable CORS for all routes
   app.use(cors());

   // Example: Enable CORS only for a specific route
   app.get('/public-data', cors(), (req, res) => {
       // ...
   });
   ```

```
    // Example: Customize CORS options
    const customCorsOptions = {
        origin: 'https://example.com',
        optionsSuccessStatus: 200 // Some legacy browsers choke on 204
    };

    app.use(cors(customCorsOptions));
```

Manual CORS Setup

Use the following code example to **set CORS headers manually** in your Express app:

```
app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
    if (req.method === 'OPTIONS') {
        res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, PATCH, DELETE, OPTIONS');
        return res.status(200).json({});
    }
    next();
});
```

Make sure to place this middleware before your route definitions.

10. Explain the use of `next()` in *Express.js middleware*.

In Express.js, **middleware** functions are crucial for handling HTTP requests. A single request can pass through multiple middlewares before reaching its endpoint, providing opportunities for tasks like logging, data parsing, and error handling. The `next()` function is instrumental in this process, allowing for both regular middleware chaining and special error handling.

What is `next()`?

- `next()` : A callback function that, when called within a middleware, passes control to the next middleware in the stack.
- `next()` is typically invoked to signal that a middleware has completed its tasks and that the request should move on to the next middleware.
- If a middleware doesn't call `next()` , the request flow can get **stuck**, and the subsequent middlewares won't be executed.

Use-Cases

1. **Regular Flow**: Invoke `next()` to move the request and response objects through the middleware stack.
2. **Error Handling**: If a middleware detects an error, it can short-circuit the regular flow and jump directly to an error-handling middleware (defined with `app.use(function(err, req, res, next) {})`). This is achieved by calling `next(err)` , where `err` is the detected error.

Code Example: Logging Middleware

Here is the code:

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

// Sample middleware: logs the request method and URL
app.use((req, res, next) => {
    console.log(`${req.method} ${req.url}`);
    next(); // Move to the next middleware
});

// Sample middleware: logs the current UTC time
app.use((req, res, next) => {
    console.log(new Date().toUTCString());
    next(); // Move to the next middleware
});

app.listen(3000);
```

In this example, both middlewares call `next()` to allow the request to progress to the next logging middleware and eventually to the **endpoint** (not shown, but would be the next in the chain).

Without the `next()` calls, the request would get **stuck** after the first middleware.

11. What is the role of the `express.Router` class?

The `express.Router` is a powerful tool for **managing multiple route controllers**. It helps in organizing routes and their handling functions into modular, self-contained groups.

Key Features

- **Modularity**: Rely on separate route modules for improved code organization, maintainability, and collaboration.
- **Middlewares**: Like the main `express` app, the router can also use middlewares to process incoming requests.
- **HTTP Method Chaining**: Simplifies route handling by allowing method-specific routes to be defined using method names.

Example: Middleware and Route Handling

Here is the Node.js code:

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

// Logger Middleware
router.use((req, res, next) => {
  console.log('Router-specific Request Time:', Date.now());
  next();
});

// "GET" method route
router.get('/', (req, res) => {
  res.send('Router Home Page');
});

// "POST" method route
router.post('/', (req, res) => {
  res.send('Router Home Page - POST Request');
});

module.exports = router;
```

In this example, we:

- Utilize the built-in `express.Router`.
- Attach a general-purpose middleware and two different HTTP method-specific routes.
- The router is then integrated into the main `express` app using:

```
const app = express();
const router = require('./myRouterModule');

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

Here, `app.use('/routerExample', router);` assigns all routes defined in the router to `/routerExample`.

12. How do you handle *404 errors* in *Express.js*?

Handling 404 errors in Express is essential for capturing and responding to requests for non-existent resources. You typically use both **middleware** and **HTTP response** mechanisms for this purpose.

Middleware for 404s

1. Use `app.use` at the end of the middleware chain to capture unresolved routes.
2. Invoke the middleware with `next()` and an `Error` object to forward to the error-handling middleware.

Here is the Node.js code example:

```
app.use((req, res, next) => {
    const err = new Error(`Not Found: ${req.originalUrl}`);
    err.status = 404;
    next(err);
});
```

Error-Handling Middleware for 404s and Other Errors

1. Define an error-handling middleware with **four** arguments. The first one being the `error` object.
2. Check the error's status and respond accordingly. If it's a 404, handle it as a not-found error; otherwise, handle it as a server error.

Here is the Node.js code:

```
app.use((err, req, res, next) => {
    const status = err.status || 500;
    const message = err.message || "Internal Server Error";

    res.status(status).send(message);
});
```

Full Example:

Here is the complete Node.js application:

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

// Sample router for demonstration
const usersRouter = express.Router();
usersRouter.get('/profile', (req, res) => {
    res.send('User Profile');
});
app.use('/users', usersRouter);

// Capture 404s
app.use((req, res, next) => {
    const err = new Error(`Not Found: ${req.originalUrl}`);
    err.status = 404;
    next(err);
});

// Error-handling middleware
app.use((err, req, res, next) => {
    const status = err.status || 500;
    const message = err.message || "Internal Server Error";
    res.status(status).send(message);
});

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

13. What are the differences between `req.query` and `req.params`?

In Express.js, `req.query` is used to access **GET** request parameters, while `req.params` is used to capture parameters defined in the **URL path**.

Understanding Express.js Routing

Express.js uses **app.get()** and similar functions to handle different types of HTTP requests.

- **app.get('/users/:id')**: Matches GET requests to `/users/123` where `123` is the `:id` parameter in the path.

Accessing Request Data

- **req.query**: Utilized to extract query string parameters from the request URL. Example: For the URL `/route?id=123`, use `req.query.id` to obtain `123`.
- **req.params**: Used to retrieve parameters from the request URL path. For the route `/users/:id`, use `req.params.id` to capture the ID, such as for `/users/123`.

Code Example: Request Data

Here is the Express.js server setup:

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

// Endpoint to capture query string parameter
app.get('/query', (req, res) => {
  console.log(req.query);
  res.send('Received your query param!');
});

// Endpoint to capture URL parameter
app.get('/user/:id', (req, res) => {
  console.log(req.params);
  res.send('Received your URL param!');
});

app.listen(port, () => console.log(`Listening on port ${port}!`));
```

14. Describe the purpose of `req.body` and how you would access it.

In an Express.js application, `req.body` is a property of the **HTTP request object** that contains data submitted through an HTTP POST request.

The POST request might originate from an HTML form, a client-side JavaScript code, or another API client. The data in `req.body` is typically structured as a JSON object or a URL-encoded form.

Middleware and Parsing Request Body

The `express.json()` and `express.urlencoded()` middleware parse incoming `Request` objects before passing them on. These middlewares populate `req.body` with the parsed JSON and URL-encoded data, respectively.

Here is an example of how you might set up body parsing in an Express app:

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

// Parse JSON and URL-encoded data into req.body
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Accessing `req.body` Data

Once the body parsing middleware is in place, you can access the parsed data in your **route handling** functions:

- **POST** or **PUT** Requests: When a client submits a POST or PUT request with a JSON payload in the request body, you can access this data through `req.body`.

Here is an example:

Client-side JavaScript:

```
fetch('/example-route', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ key: 'value' })
});
```

Server-side Express route handler:

```
app.post('/example-route', (req, res) => {
  console.log(req.body); // Outputs: { key: 'value' }
});
```

- **HTML Forms**: When a form is submitted using `<form>` with `action` pointing to your Express route and `method` as POST or PUT, and the form fields are input elements within the form, `req.body` will contain these form field values.

README

HTML form:

```html
<form action="/form-endpoint" method="POST">
  <input type="text" name="username" />
  <input type="password" name="password" />
  <button type="submit">Submit</button>
</form>
```

Express route:

```javascript
app.post('/form-endpoint', (req, res) => {
  console.log(req.body.username, req.body.password);
});
```

A modern technique for sending form data using `fetch` is by setting the `Content-Type` header to `'application/x-www-form-urlencoded'` and using the `URLSearchParams` object:

```javascript
fetch('/form-endpoint', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: new URLSearchParams({ username: 'user', password: 'pass' })
});
```

- **Custom Parsers**: While Express provides built-in body parsers for JSON and URL-encoded data, you might receive data in another format. In such cases, you can create custom middleware to parse and shape the data as needed. This middleware should populate `req.body`.

15. How do you create a *middleware* that logs the *request method* and *URL* for every request?

In Express.js, **middlewares** allow you to handle HTTP requests. Here, you will learn how to create a simple **logging middleware** that records the request method and URL.

Setting Up the Express App

First, install Express via npm, and set up your `app.js` file:

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

Creating the Logging Middleware

Define a logging function that extracts the request method and URL, and then use `app.use()` to mount it as middleware.

```javascript
// Logging Middleware
const logRequest = (req, res, next) => {
  console.log(`Received ${req.method} request for: ${req.url}`);
  next(); // Call next to proceed to the next middleware
};

// Mount the middleware for all routes
app.use(logRequest);
```

Testing the Setup

Use `app.get()` to handle GET requests, and `app.listen()` to start the server.

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

```
// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

When you visit `http://localhost:3000/` in your browser and check the server console, you should see the request being logged.

Explore all 58 answers here 👉 Devinterview.io - Express.js

Releases

No releases published

Packages

No packages published