

Express

Interview Questions

(Practice Project)



Interview Questions

Easy

1. What is the main difference between Express and Node.js?

Answer –

Node.js is a runtime environment that executes JavaScript on the server, while Express.js is a web application framework built on Node.js for simplifying web development.

2. What are the benefits of using Express.js?

Answer –

- It makes it easy to develop web applications.
- It is fast and efficient.
- It is scalable and can be used to develop high-traffic web applications.
- It has a large community and there are many resources available to help you learn and use Express.js.

3. What are some best practices for developing Express.js applications?

Answer –

Some best practices for developing Express.js applications include:

- Use middleware to organize your code and make it easier to maintain.
- Implement error handling to ensure that your application remains stable even when errors occur.
- Optimize your database queries to improve the performance of your application.
- Use a minifier to reduce the size of your JavaScript and CSS files.
- Test your code thoroughly to ensure that it is bug-free

4. What is the purpose of the next() function in Express middleware?

Answer –

The next() function is used in Express Middleware to pass control to the next middleware function or route handler in the stack.

5. Name some of the unique Express.js Features.

Answer –

- It allows developers to design single and multi-page applications using Express.js
- Express.js supports various databases such as RDBMS and NoSQL
- It allows developers to dynamically render HTML pages with arguments and templates

7. Name some built-in middleware and what are they used for?

Answer –

Some commonly used built-in middleware in Express are –

- 1. express.json()** – This middleware is used to parse JSON data in the request body. It makes the JSON data available as req.body
- 2. express.urlencoded()** – This middleware is used to parse URL-encoded data in the request body. It parses data sent via HTML forms
- 3. express.static()** – This middleware serves static files, such as HTML, CSS, and JavaScript, from a specified directory. It simplifies the serving of static assets.
- 4. express.raw()** – This middleware is used to parse raw request data and make it available in the req.body. It's often used for handling binary data
- 5. express.text()** – This middleware parses text data from the request body and makes it available in req.body.
- 6. express.Router()** – While not exactly middleware, it allows you to create modular route handlers and is often included in lists of built-in middleware. You can define routes and route handlers using the Router object.

Medium

1. What is middleware in Express.js Why is it essential for Express applications and What are the different types of middleware

Answer –

Middleware functions in Express.js are functions that have access to the request and response objects. They can perform tasks, modify data, and end the request-response cycle. Middleware is vital for tasks like authentication, logging, and handling common operations in web applications.

The different types of Middleware –

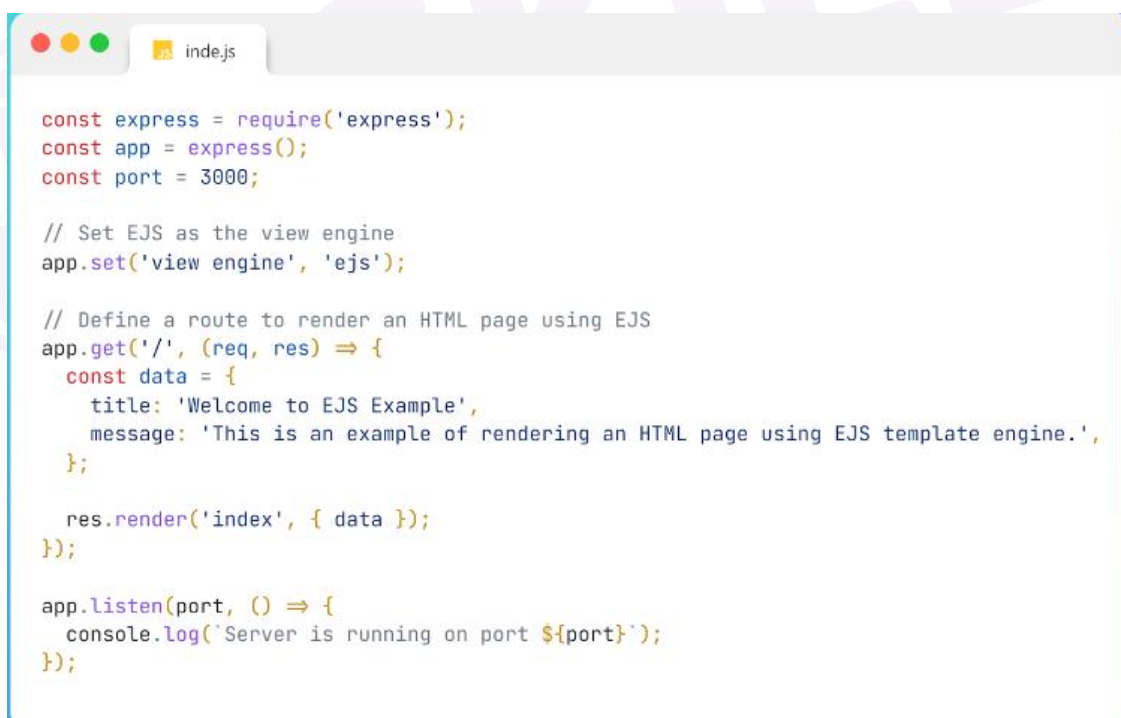
- Application-level Middleware
- Router-level Middleware
- Error-handling Middleware
- Built-in Middleware
- Third-party middleware

2. Explain the role of a templating engine in Express and provide an example of using a templating engine

Answer –

A templating engine in Express is used to generate dynamic HTML content. Example usage with the EJS templating engine:

Example of template engine



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

// Set EJS as the view engine
app.set('view engine', 'ejs');

// Define a route to render an HTML page using EJS
app.get('/', (req, res) => {
  const data = {
    title: 'Welcome to EJS Example',
    message: 'This is an example of rendering an HTML page using EJS template engine.',
  };

  res.render('index', { data });
});

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

3. How do you improve the performance of an Express.js application?

Answer –

Some of the ways to improve the Express.js application are –

- Use a caching middleware to cache frequently accessed data. This will reduce the number of database queries that need to be made, which can improve performance significantly.
- Use a load balancer to distribute traffic across multiple servers. This can help to improve the performance of your application by handling more requests simultaneously.
- Optimize your database queries. This can be done by using indexes and avoiding unnecessary queries.
- Use a minifier to reduce the size of your JavaScript and CSS files. This will reduce the amount of data that needs to be downloaded by the client, which can improve page load times.
- Use a content delivery network (CDN) to serve static files. This will offload the serving of static files from your servers, which can improve performance and scalability.
- Use a reverse proxy to handle SSL/TLS termination and caching. This can improve performance and security by offloading these tasks from your Express.js servers.

4. How does the `next()` function work in a sequence of middleware functions in Express? Can you provide an example of its usage in a typical middleware stack?

Answer -

The `next()` function is used to pass control from one middleware function to the next in the stack. Here's an example of a typical middleware stack:



```

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

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

app.get('/route', (req, res) => {
  // Route handler
  console.log('Route handler');
});
/*

```

5. What is Scaffolding in Express.js

Answer -

Scaffolding in Express.js is the process of creating a basic structure for your Express application. This can be done manually, but there are also a number of tools available to help you automate the process. One popular scaffolding tool for Express.js is `express-generator`. This tool allows you to quickly create a new Express application with a basic structure in place.

Here are some of the benefits of using scaffolding in Express.js:

- It can save you time and effort.
- It can help you to avoid making common mistakes.
- It can provide you with a basic structure for your application, which you can then customize to meet your specific needs.
- It can help you to learn the basics of Express.js more quickly.

Hard

1. Explain the concept of a router in Express.js and provide a use case for using routers.

Answer -

An Express router is a middleware that allows you to group and modularize route handlers. Use cases include creating separate routers for different parts of an application, like user authentication, and then mounting them in the main application for a more organized and maintainable codebase.

2. Describe the differences between the `app.all()`, `app.use()`, and `app.route()` methods in Express and provide scenarios for when to use each.

Answer –

`app.all()` matches all HTTP methods,
`app.use()` is for adding middleware, and
`app.route()` creates a chainable route for a specific endpoint.

For example, `app.all()` is suitable for global middleware, `app.use()` for general middleware, and `app.route()` for a specific route with multiple HTTP methods.

3. Explain how to create custom middleware that handles user authentication in an Express application. Provide a code example.

Answer –

To create custom authentication middleware, you can check user credentials in a middleware function and conditionally allow or deny access. Here's an example:



```
const authMiddleware = (req, res, next) => {
  if (req.isAuthenticated()) {
    next(); // User is authenticated, continue
  } else {
    res.status(401).send('Unauthorized');
  }
};

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

4. Explain the concept of "short-circuiting" in Express middleware. How can you use the `next()` function to implement it, and in what scenarios might it be beneficial?

Answer –

Short-circuiting in Express middleware refers to stopping the execution of middleware and route handlers based on certain conditions, without proceeding to subsequent functions in the stack. The `next()` function can be used to achieve this by not calling it when conditions for short-circuiting are met. This can be beneficial for scenarios where certain routes or middleware should only execute under specific conditions, improving efficiency and reducing unnecessary processing.

```

app.use((req, res, next) => {
  if (req.user) {
    // User is authenticated, proceed
    next();
  } else {
    // User is not authenticated, short-circuit
    res.status(401).send('Unauthorized');
  }
});

app.get('/secure', (req, res) => {
  // This route is only reached if the user is authenticated
  res.send('Access granted');
});

/*
In this example, if the user is not authenticated,
the route is short-circuited,
and the response is sent without further processing.
*/

```

5. Differentiate between built-in and third-party middleware provided with an example

Answer -

Built-In Middleware -

Built-in middleware in Express.js are pre-defined middleware functions that are included with the Express framework. They are part of the core Express package and can be used without installing additional packages.

Example -

`express.json()`: This middleware is used to parse incoming JSON data in request bodies. It automatically parses the JSON data and makes it accessible through `req.body`

```

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

app.use(express.json());

```

Third-Party Middleware -

Third-party middleware in Express.js are middleware functions or packages developed by third-party developers or the community. These middleware extend the functionality of Express and are not included in the core package. They need to be installed separately using NPM or Yarn.

Example

morgan: A third-party middleware for logging HTTP requests. It provides request and response logging, making it easier to debug and monitor requests.

```

const morgan = require('morgan');
app.use(morgan('dev'));

```