The top 100 interview questions on Express.js

1. What is Express.js?

Answer: Express.js is a popular web application framework for Node.js that provides a simple and flexible way to build web applications and APIs.

2. What are the key features of Express.js?

Answer: The key features of Express.js include routing, middleware support, template engine integration, error handling, and support for various HTTP methods.

3. How do you install Express.js?

Answer: You can install Express.js using npm (Node Package Manager) by running the command: `npm install express`.

4. How do you create a new Express.js application?

Answer: To create a new Express.js application, you need to require the express module and create an instance of the Express application using the following code:

javascript

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

const app = express();
```

5. How do you start a server using Express.js?

Answer: You can start a server using Express.js by calling the `listen` method on the Express application instance and specifying the port number, as shown below:

javascript

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

6. What is middleware in Express.js?

Answer: Middleware in Express.js refers to functions that have access to the request and response objects and can modify them or perform additional tasks. It sits between the initial request and the final response and can be used for tasks such as logging, authentication, and error handling.

7. How do you use middleware in Express.js?

Answer: You can use middleware in Express.js by calling the `use` method on the Express application instance and passing the middleware function as an argument. Middleware functions can be defined inline or in separate modules.

8. What are the different types of middleware in Express.js?

Answer: There are different types of middleware in Express.js:

- Application-level middleware: It is bound to the Express application and is used for tasks that are specific to the application, such as logging and error handling.

- Router-level middleware: It is bound to a specific router and is used for tasks related to that router, such as authentication and request preprocessing.

- Error-handling middleware: It is used to handle errors that occur during the request-response cycle.

- Third-party middleware: These are external middleware modules that can be used to add additional functionality to the application.

9. How do you handle routing in Express.js?

Answer: Routing in Express.js is handled using the `express.Router` class. You can create an instance of the Router class, define routes on it, and then mount the router on the main Express application using the `app.use` method.

10. How do you define a route in Express.js?

Answer: You can define a route in Express.js by calling the appropriate method on the Express application or router instance, specifying the HTTP method and the route path, and providing a callback function to handle the request.

Example:

javascript

```
app.get('/users', (req, res) => {
  res.send('GET request to /users');
});
```

11. How do you handle HTTP GET requests in Express.js?

Answer: You can handle HTTP GET requests in Express.js by using the `get` method on the Express application or router instance, specifying the route path and providing a callback function to handle the request.

12. How do you handle HTTP POST requests in Express.js?

Answer: You can handle HTTP POST requests in Express.js by using the `post` method on the Express application or router instance, specifying the route path and providing a callback function to handle the request.

13. How do you handle HTTP PUT requests in Express.js?

Answer: You can handle HTTP PUT requests in

Express.js by using the `put` method on the Express application or router instance, specifying the route path and providing a callback function to handle the request.

14. How do you handle HTTP DELETE requests in Express.js?

Answer: You can handle HTTP DELETE requests in Express.js by using the `delete` method on the Express application or router instance, specifying the route path and providing a callback function to handle the request.

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

Answer: The `next` function is used in Express.js middleware to pass control to the next middleware function in the chain. It is typically called inside a middleware function to allow the application to continue processing subsequent middleware or routes.

16. How do you handle errors in Express.js?

Answer: You can handle errors in Express.js by defining an error-handling middleware function with four parameters: `(err, req, res, next)`. This function is called whenever an error occurs in the request-response cycle. You can handle the error and send an appropriate response to the client.

Example:

javascript

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

17. What is the purpose of the "app.use()" method in Express.js?

Answer: The `app.use()` method is used to mount middleware functions on the Express application. It can be used to define application-level middleware or to mount routers or other middleware at a specified route.

18. How do you serve static files in Express.js?

Answer: You can serve static files in Express.js by using the `express.static` middleware function and specifying the directory from which to serve the files.

Example:

javascript

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

19. What is the purpose of the "req" object in Express.js?

Answer: The `req` object in Express.js represents the request made by the client to the server. It contains information about the request, such as the URL, HTTP headers, query parameters, request body, and more.

20. What is the purpose of the "res" object in Express.js?

Answer: The `res` object in Express.js represents the response that will be sent back to the client. It is used to send the response data, set response headers, and perform other operations related to the response.

21. What is the purpose of the "app.locals" object in Express.js?

Answer: The `app.locals` object in Express.js provides a way to pass data from the server to views or templates. It is an object that is available in all views rendered by the application.

Example:

javascript

app.locals.title = 'My Express App';

22. How do you access query parameters in Express.js?

Answer: You can access query parameters in Express.js through the `req.query` object. It contains key-value pairs of the query parameters sent with the request.

Example:

javascript

// GET /users?name=John&age=25

console.log(req.query.name); // Output: John

console.log(req.query.age); // Output: 25

23. How do you access request headers in Express.js?

Answer: You can access request headers in Express.js through the `req.headers` object. It contains key-value pairs of the headers sent with the request.

Example:

javascript

console.log(req.headers['content-type']);

24. How do you access request body data in Express.js?

Answer: To access request body data in Express.js, you need to use middleware such as `body-parser` or the built-in `express.json` middleware. Once the middleware is set up, you can access the parsed request body through the `

req.body` object.

Example with `body-parser` middleware:

javascript

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

app.post('/users', (req, res) => {
  console.log(req.body);
});
```

25. How do you set response headers in Express.js?

Answer: You can set response headers in Express.js by using the `res.set()` method or by directly assigning values to the `res.headers` object. The headers must be set before sending the response.

Example:

javascript

```
res.set('Content-Type', 'application/json');
```

26. How do you redirect a request in Express.js?

Answer: You can redirect a request in Express.js by using the `res.redirect()` method and specifying the URL to redirect to. The status code for the redirect is set to 302 by default.

Example:

javascript

```
res.redirect('/new-page');
```

27. What is Express Router and how do you use it?

Answer: Express Router is a class provided by Express.js that allows you to create modular, mountable route handlers. You can create an instance of the Router class, define routes on it, and then mount it on the main Express application using the `app.use()` method.

Example:

javascript

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

const router = express.Router();


router.get('/users', (req, res) => {
  // Handle GET request for /users
});


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

28. How do you implement sessions in Express.js?

Answer: You can implement sessions in Express.js by using middleware such as `express-session`. This middleware creates and manages a session object for each client and allows you to store session data and access it across requests.

Example with `express-session` middleware:

javascript

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

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


const app = express();

app.use(session({
  secret: 'my-secret-key',
  resave: false,
  saveUninitialized: true
}));
```

```
app.get('/counter', (req, res) => {
  if (req.session.views) {
    req.session.views++;
  } else {
    req.session.views = 1;
  }
  res.send(`Views: ${req.session.views}`);
});
```

29. How do you implement authentication in Express.js?

Answer: Authentication in Express.js can be implemented by using middleware such as Passport.js. Passport.js provides a flexible authentication framework that supports various authentication strategies like local, OAuth, and JWT.

Example with Passport.js:

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

const app = express();
app.use(passport.initialize());

app.get('/login', passport.authenticate('local'), (req, res) => {
  res.send('Logged in successfully');
});
```

30. How do you implement authorization in Express.js?

Answer: Authorization in Express.js can be implemented by using middleware functions that check whether the current user has the required permissions to access a specific resource. These middleware functions are typically added to the routes or route handlers that require authorization.

Example:

javascript

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

const app = express();

const checkAdmin = (req, res, next) => {
  if (req.user.role === 'admin') {
    next();
  } else {
    res.status(403).send('Forbidden');
  }
};

app.get('/admin', checkAdmin, (req, res) => {
  res.send('Welcome, admin');
});
```

31. How do you handle file uploads in Express.js?

Answer: You can handle file uploads in Express.js by using middleware such as `multer`. Multer is a popular middleware that provides support

for handling multipart/form-data, which is used for file uploads.

Example with Multer:

javascript

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

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


const app = express();

const upload = multer({ dest: 'uploads/' });


app.post('/upload', upload.single('file'), (req, res) => {

  console.log(req.file);

});
```

32. What is Express generator and how do you use it?

Answer: Express generator is a command-line tool that helps in generating the basic structure and files for an Express.js application. It sets up the application with default configurations, including the file structure, routing, middleware, and package.json.

To use Express generator, you need to install it globally using npm: `npm install -g express-generator`. Then, you can create a new Express application using the `express` command followed by the application name.

Example:

bash

```
express myapp

cd myapp

npm install
```

33. What are route handlers in Express.js?

Answer: Route handlers in Express.js are callback functions that are executed when a specific route is matched. They handle the logic for processing the request and sending the response.

Example:

javascript

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

```
  // Handle GET request for /users
});
```

34. How do you pass data to route handlers in Express.js?

Answer: You can pass data to route handlers in Express.js by using route parameters or query parameters.

Example with route parameters:

javascript

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

Example with query parameters:

javascript

```
app.get('/users', (req, res) => {
  const sortBy = req.query.sortBy;
  // Handle request with sorting parameter
});
```

35. How do you implement pagination in Express.js?

Answer: Pagination in Express.js can be implemented by using query parameters to control the number of items per page and the page number. In the route handler, you can use these parameters to retrieve the appropriate subset of data to send back as the response.

Example:

javascript

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

```
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;

  // Retrieve users based on pagination parameters
  const startIndex = (page - 1) * limit;
  const endIndex = page * limit;
  const users = data.slice(startIndex, endIndex);

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

36. How do you implement caching in Express.js?

Answer: Caching in Express.js can be implemented by using middleware functions such as `response-time` and `express-cache-headers`. These middleware functions add caching headers to the response, allowing the client or intermediate caches to cache the response and serve it for subsequent requests.

Example with `express-cache-headers` middleware:

javascript

```
const express = require('express');
const cacheHeaders = require('express-cache-headers');

const app = express();
app.use(cacheHeaders());

app.get('/users', (req, res) => {
  // Retrieve users from the database
  // Set cache headers using res.setCacheHeaders() method
  res.setCacheHeaders({
    maxAge: 60,
    private: true
```

```
  });

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

37. How do you handle CORS in Express.js?

Answer: You can handle CORS (Cross-Origin Resource Sharing) in Express.js by using middleware such as `cors`. CORS middleware adds the necessary headers to allow cross-origin requests from specified domains.

Example with `cors` middleware:

```
javascript

const express

 = require('express');

const cors = require('cors');

const app = express();

app.use(cors());

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

  // Handle the request

});
```

38. How do you handle WebSocket connections in Express.js?

Answer: Express.js does not provide built-in support for WebSocket connections. However, you can use external libraries such as `socket.io` to handle WebSocket connections alongside your Express.js application.

Example with `socket.io`:

javascript

```
const express = require('express');
const app = express();
const server = require('http').createServer(app);
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  console.log('A user connected');

  socket.on('chat message', (msg) => {
    console.log('Received message:', msg);
    io.emit('chat message', msg);
  });

  socket.on('disconnect', () => {
    console.log('A user disconnected');
  });
});

server.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

39. How do you handle cookies in Express.js?

Answer: You can handle cookies in Express.js by using the `cookie-parser` middleware. This middleware parses the cookies sent by the client and makes them available in the `req.cookies` object.

Example with `cookie-parser` middleware:

javascript

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

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

const app = express();
app.use(cookieParser());

app.get('/visit', (req, res) => {
  const visits = parseInt(req.cookies.visits) || 0;
  res.cookie('visits', visits + 1);
  res.send(`Total visits: ${visits + 1}`);
});
```

40. How do you handle route-specific middleware in Express.js?

Answer: You can define route-specific middleware in Express.js by adding the middleware functions as additional arguments before the route handler function in the route definition.

Example:

javascript

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

const logMiddleware = (req, res, next) => {
  console.log('Log middleware');
  next();
};

const routeMiddleware = (req, res, next) => {
  console.log('Route middleware');
  next();
};
```

```
app.get('/users', logMiddleware, routeMiddleware, (req, res) => {
  res.send('Users');
});
```

51. How do you set response headers in Express.js?

Answer: You can set response headers in Express.js using the `res.set()` or `res.header()` methods. These methods allow you to specify the headers and their values to be included in the response.

Example:

```
javascript
app.get('/users', (req, res) => {
  res.set('Content-Type', 'application/json');
  res.set({
    'Cache-Control': 'no-cache',
    'X-Custom-Header': 'Custom Value'
  });

  res.send('Response with custom headers');
});
```

52. How do you handle JSON data in the request body in Express.js?

Answer: You can handle JSON data in the request body in Express.js by using the `express.json()` middleware. This middleware automatically parses JSON data sent in the request body and makes it available in `req.body`.

Example:

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

app.post('/users', (req, res) => {
  const userData = req.body;
```

```
  // Handle user creation with JSON data
});
```

53. How do you handle URL-encoded form data in the request body in Express.js?

Answer: You can handle URL-encoded form data in the request body in Express.js by using the `express.urlencoded()` middleware. This middleware parses the URL-encoded data sent in the request body and makes it available in `req.body`.

Example:

javascript

```
app.use(express.urlencoded({ extended: true }));

app.post('/users', (req, res) => {
  const formData = req.body;
  // Handle form data
});
```

54. How do you handle request validation in Express.js?

Answer: You can handle request validation in Express.js by using validation middleware such as `express-validator`. This middleware allows you to define validation rules for the request body, query parameters, route parameters, and headers.

Example with `express-validator`:

javascript

```
const { body, validationResult } = require('express-validator');

app.post('/users', [
  body('name').notEmpty().withMessage('Name is required'),
  body('email').isEmail().withMessage('Invalid email'),
], (req, res) => {
```

```
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  // Handle valid request
});
```

55. How do you handle authentication and authorization in Express.js?

Answer: Authentication and authorization in Express.js can be implemented using middleware functions. Authentication middleware validates the user's credentials and sets the user information in `req.user`. Authorization middleware checks if the user has the necessary permissions to access a specific resource.

Example:

```
javascript
const authenticateUser = (req, res, next) => {
  // Check user credentials and set req.user
  next();
};

const authorizeUser = (req, res, next) => {
  // Check user permissions
  if (req.user.isAdmin) {
    next();
  } else {
    res.status(403).send('Forbidden');
  }
};

app.get('/admin/dashboard', authenticateUser, authorizeUser, (req, res) => {
```

```
  // Handle authorized request
});
```

56. How do you handle sessions and cookies in Express.js?

Answer: You can handle sessions and cookies in Express.js using middleware such as `express-session` and `cookie-parser`. The `express-session` middleware manages session data and assigns a unique session ID to each client. The `cookie-parser` middleware parses cookies sent by the client and makes them available in `req.cookies`.

Example:

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

app.use(cookieParser());
app.use(session({
  secret: 'secret-key',


  resave: false,
  saveUninitialized: true,
}));

app.get('/dashboard', (req, res) => {
  const sessionData = req.session;
  const cookieData = req.cookies;
  // Handle session and cookie data
});
```

57. How do you handle file uploads in Express.js?

Answer: You can handle file uploads in Express.js using middleware such as `multer`. This middleware provides a flexible way to handle file uploads, including storing files on the server, validating file types, and renaming files.

Example with `multer`:

javascript

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


const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    cb(null, file.originalname);
  },
});


const upload = multer({ storage });


app.post('/upload', upload.single('file'), (req, res) => {
  // Handle uploaded file
});
```

58. How do you handle cross-origin resource sharing (CORS) in Express.js?

Answer: You can handle Cross-Origin Resource Sharing (CORS) in Express.js using the `cors` middleware. This middleware adds the necessary headers to enable cross-origin requests from client applications hosted on different domains.

Example with `cors`:

javascript

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

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


app.get('/api/data', (req, res) => {
  // Handle the request
});
```

59. How do you handle server-side rendering (SSR) with Express.js?

Answer: Server-side rendering (SSR) with Express.js involves rendering the initial HTML on the server and sending it to the client. You can use templating engines like `EJS` or `Pug` to generate dynamic HTML on the server.

Example with `EJS`:

```
javascript

app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));


app.get('/users', (req, res) => {
  const users = ['John', 'Jane', 'Bob'];
  res.render('users', { users });
});
```

60. How do you implement pagination in Express.js?

Answer: Pagination in Express.js involves handling requests for specific pages of data. You can use query parameters to specify the page number and limit, and then apply appropriate logic to retrieve and display the requested data.

Example:

```
javascript

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

```
 const page = parseInt(req.query.page) || 1;
 const limit = parseInt(req.query.limit) || 10;

 // Apply pagination logic (e.g., retrieve users for the specified page and limit)

 res.json({
  page,
  limit,
  users: /* Retrieved users */,
 });
});
```

61. How do you handle request logging in Express.js?

Answer: You can handle request logging in Express.js by creating a middleware function that logs information about each incoming request. This middleware can log details such as the request method, URL, headers, and timestamps.

Example:

```
javascript
const requestLogger = (req, res, next) => {
 console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
 next();
};

app.use(requestLogger);

app.get('/users', (req, res) => {
 // Handle GET request for /users
});
```

62. How do you handle response compression in Express.js?

Answer: You can handle response compression in Express.js using middleware such as `compression`. This middleware automatically compresses the response body using gzip or deflate encoding, reducing the size of the data sent over the network.

Example with `compression`:

javascript

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


app.use(compression());


app.get('/users', (req, res) => {
  // Handle GET request for /users
});
```

63. How do you handle rate limiting in Express.js?

Answer: You can handle rate limiting in Express.js using middleware such as `express-rate-limit`. This middleware allows you to set limits on the number of requests a client can make within a specified time period, preventing abuse or excessive usage.

Example with `express-rate-limit`:

javascript

```
const rateLimit = require('express-rate-limit');


const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Maximum 100 requests per windowMs
});


app.use(limiter);


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

```
  // Handle GET request for /users
});
```

64. How do you handle CSRF protection in Express.js?

Answer: You can handle CSRF (Cross-Site Request Forgery) protection in Express.js by using middleware such as `csurf`. This middleware adds a CSRF token to each form or API request, and validates the token on subsequent requests to prevent CSRF attacks.

Example with `csurf`:

javascript

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

app.use(csurf());

app.get('/form', (req, res) => {
  const csrfToken = req.csrfToken();
  res.render('form', { csrfToken });
});

app.post('/submit', (req, res) => {
  // Validate CSRF token and process the submitted form
});
```

65. How do you handle WebSocket connections in Express.js?

Answer: Express.js does not provide built-in support for WebSocket connections. However, you can use external libraries such as `socket.io` to handle WebSocket connections alongside your Express.js application.

Example with `socket.io`:

javascript

```
const express = require('express');
const app = express();
const server = require('http').createServer(app);
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  console.log('A user connected');

  socket.on('chat message', (msg) => {
    console.log('Received message:', msg);
    io.emit('chat message', msg);
  });

  socket.on('disconnect', () => {
    console.log('A user disconnected');
  });
});

server.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

66. How do you handle server-sent events (SSE) in Express.js?

Answer: You can handle server-sent events (SSE) in Express.js by using the `res.sse()` method provided by the `express-sse` library. This method allows you to send SSE data to

clients, enabling real-time server-to-client communication.

Example with `express-sse`:

javascript

```
const express = require('express');
const SSE = require('express-sse');

const app = express();
const sse = new SSE();

app.get('/stream', sse.init);

app.post('/data', (req, res) => {
  const newData = req.body; // New data to be sent

  sse.send(newData); // Send the data to connected clients

  res.sendStatus(200);
});
```

67. How do you handle route prefixing in Express.js?

Answer: You can handle route prefixing in Express.js by using the `express.Router` middleware. This middleware allows you to create modular route handlers that can be mounted to a specific path prefix.

Example:

javascript

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

router.get('/users', (req, res) => {
  // Handle GET request for /api/users
});

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

```
  // Handle POST request for /api/users
});

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

68. How do you handle custom 404 errors in Express.js?

Answer: You can handle custom 404 errors in Express.js by defining a middleware function that runs after all other route handlers. This middleware function can be used to handle requests that don't match any route, returning a custom 404 error message.

Example:

javascript

```
app.use((req, res) => {
  res.status(404).send('Page not found');
});
```

69. How do you handle long-polling in Express.js?

Answer: Long-polling is a technique used for real-time communication between a client and a server. While Express.js does not provide built-in support for long-polling, you can implement it by combining techniques such as event emitters and timeouts.

Example:

javascript

```
app.get('/data', (req, res) => {
  // Perform initial data retrieval

  // Long-polling implementation
  const intervalId = setInterval(() => {
    if (/* New data available */) {
      clearInterval(intervalId);
```

```
    res.json({ newData: true });
  }
 }, 1000);
});
```

70. How do you handle multiple callback functions in a route handler in Express.js?

Answer: You can handle multiple callback functions in a route handler in Express.js by passing an array of callback functions as the second argument to the route handler. Each callback function is executed in the order they appear in the array.

Example:

javascript

```
const authenticate = (req, res, next) => {
 // Authenticate the user
 next();
};

const authorize = (req, res, next) => {
 // Authorize the user
 next();
};

app.get('/users', [authenticate, authorize], (req, res) => {
 // Handle GET request for /users
});
```

Certainly! Here are the next 10 interview questions on Express.js:

71. How do you implement method overriding in Express.js?

Answer: Express.js does not provide built-in support for method overriding. However, you can use middleware such as `method-override` to enable method overriding by utilizing query parameters or headers to specify the desired HTTP method.

Example with `method-override`:

javascript

```
const methodOverride = require('method-override');


app.use(methodOverride('_method'));


app.put('/users/:id', (req, res) => {
  // Handle PUT request for updating a user
});
```

72. How do you handle server-side validation in Express.js?

Answer: Server-side validation in Express.js involves validating the data received from the client to ensure it meets certain criteria or business rules. You can use validation libraries such as `express-validator` or custom validation logic to perform server-side validation.

Example with `express-validator`:

javascript

```
const { body, validationResult } = require('express-validator');


app.post('/users', [
  body('username').notEmpty().withMessage('Username is required'),
  body('email').isEmail().withMessage('Invalid email address'),
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  // Handle valid request
});
```

73. How do you handle file downloads in Express.js?

Answer: You can handle file downloads in Express.js by setting the appropriate headers and sending the file as a response using the `res.download()` method. This method automatically sets the headers to trigger a file download in the client's browser.

Example:

javascript

```
app.get('/download', (req, res) => {
  const filePath = '/path/to/file';
  res.download(filePath);
});
```

74. How do you handle route-specific middleware in Express.js?

Answer: You can apply middleware to specific routes in Express.js by including them as additional arguments in the route handler function. These middleware functions will be executed only for the specific route they are applied to.

Example:

javascript

```
const authenticate = (req, res, next) => {
  // Authenticate user
  next();
};

const authorize = (req, res, next) => {
  // Authorize user
  next();
};
```

```
app.get('/users', authenticate, authorize, (req, res) => {
  // Handle GET request for /users with authentication and authorization
});
```

75. How do you handle environment-specific configuration in Express.js?

Answer: You can handle environment-specific configuration in Express.js by utilizing the `process.env` object and setting different configuration values based on the current environment. You can use tools like `dotenv` to load environment variables from a `.env` file.

Example:

```
javascript
const dotenv = require('dotenv');
dotenv.config();

const dbConfig = {
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  username: process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_DATABASE,
};
```

76. How do you handle cross-site scripting (XSS) prevention in Express.js?

Answer: Express.js helps prevent cross-site scripting (XSS) attacks by default through template engines such as `EJS` or `Pug`, which automatically escape HTML entities by default. It is important to properly handle user input and sanitize data when displaying it in views.

Example with `EJS`:

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

```
app.get('/users/:id', (req, res) => {
  const userData = getUserData(req.params.id);
  res.render('user', { userData });
});
```

77. How do you handle HTTPS and SSL certificates in Express.js?

Answer: To handle HTTPS and SSL certificates in Express.js, you need to create an HTTPS server using the `https` module and provide the necessary SSL certificate files. You can obtain SSL certificates from certificate authorities like Let's Encrypt.

Example:

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

const options = {
  key: fs.readFileSync('path/to/privateKey.pem'),
  cert: fs.readFileSync('path/to/certificate.pem'),
};

const server = https.createServer(options, app);

server.listen(443, () => {
  console.log('HTTPS server started on port 443');
});
```

78. How do you handle request timeouts in Express.js?

Answer: You can handle request timeouts in Express.js by using middleware such as `connect-timeout`. This middleware allows you to set a timeout duration, and if a request exceeds that duration, it will automatically return an error response.

Example with `connect-timeout`:

javascript

```
const timeout = require('connect-timeout');


app.use(timeout('5s')); // Set a timeout of 5 seconds


app.get('/users', (req, res) => {
  // Handle GET request for /users within the timeout duration
});
```

79. How do you handle handling large request bodies in Express.js?

Answer: Express.js has a default request size limit of 100kb for request bodies. To handle large request bodies, you can use middleware such as `body-parser` or `multer` to increase the size limit or handle streaming large files.

Example with `body-parser`:

javascript

```
const bodyParser = require('body-parser');


app.use(bodyParser.json({ limit: '10mb' })); // Increase the limit to 10mb


app.post('/upload', (req, res) => {
  // Handle the request with large request body
});
```

80. How do you handle response caching in Express.js?

Answer: You can handle response caching in Express.js by using middleware such as `express-cache-controller`. This middleware allows you to set cache-related headers, such as `Cache-Control` and `Expires`, to control how responses are cached by clients and intermediate proxies.

Example with `express-cache-controller`:

javascript

```
const cacheController = require('express-cache-controller');

app.use(cacheController({ maxAge: 3600 }));

app.get('/users', (req, res) => {
  // Handle GET request for /users with response caching
});
```

Certainly! Here are the next 10 interview questions on Express.js:

91. How do you handle route-specific error handling in Express.js?

Answer: You can handle route-specific error handling in Express.js by defining error-handling middleware functions that are specific to certain routes. You can use the `app.use()` method to apply these middleware functions after the route handlers for the specific routes.

Example:

javascript

```
app.get('/users', (req, res, next) => {
  // Route handler for /users
  next(new Error('An error occurred'));
});

app.use('/users', (err, req, res, next) => {
  // Error handling middleware specific to /users route
  res.status(500).json({ error: 'Internal server error' });
});
```

92. How do you handle request validation using middleware in Express.js?

Answer: You can handle request validation using middleware in Express.js by creating custom middleware functions that validate the request data. You can then apply these middleware functions to the specific routes that require validation.

Example:

javascript

```
const validateUser = (req, res, next) => {
  // Validate user data
  if (!req.body.username || !req.body.email) {
    return res.status(400).json({ error: 'Invalid user data' });
  }
  next();
};

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

93. How do you handle JSON web tokens (JWT) in Express.js?

Answer: You can handle JSON web tokens (JWT) in Express.js by using middleware such as `jsonwebtoken`. This middleware allows you to generate and verify JWTs for authentication and authorization purposes.

Example with `jsonwebtoken`:

javascript

```
const jwt = require('jsonwebtoken');

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

```
  // Authenticate user
  const token = jwt.sign({ userId: '123' }, 'secret-key');
  res.json({ token });
});

app.get('/protected', (req, res) => {
  const token = req.headers.authorization.split(' ')[1];
  const decoded = jwt.verify(token, 'secret-key');
  const userId = decoded.userId;
  // Handle authorized request
});
```

94. How do you handle rate limiting in Express.js?

Answer: You can handle rate limiting in Express.js by using middleware such as `express-rate-limit`. This middleware allows you to limit the number of requests a client can make within a certain time period.

Example with `express-rate-limit`:

javascript

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 60 * 1000, // 1 minute
  max: 100, // Maximum 100 requests per minute
});

app.use(limiter);

app.get('/users', (req, res) => {
  // Handle GET request for /users with rate limiting
});
```

95. How do you handle input sanitization and data validation in Express.js?

Answer: You can handle input sanitization and data validation in Express.js by using middleware such as `express-validator`. This middleware provides validation and sanitization functions that can be applied to request parameters, body, or query.

Example with `express-validator`:

javascript

```
const { body, validationResult } = require('express-validator');

app.post('/users', [
  body('username').notEmpty().trim().escape(),
  body('email').isEmail().normalizeEmail(),
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  // Handle valid request
});
```

96

. How do you handle file uploads in Express.js?

Answer: You can handle file uploads in Express.js by using middleware such as `multer`. This middleware allows you to handle multipart/form-data requests and store uploaded files on the server.

Example with `multer`:

javascript

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

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    cb(null, file.originalname);
  },
});

const upload = multer({ storage });

app.post('/upload', upload.single('file'), (req, res) => {
  // Handle file upload
});
```

97. How do you handle serving static files in Express.js?

Answer: You can handle serving static files in Express.js by using the `express.static` middleware. This middleware allows you to specify a directory containing static files that should be served directly to the client.

Example:

javascript

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

// Now, files in the "public" directory can be accessed directly
// e.g., http://example.com/images/logo.png
```

98. How do you handle URL parameters in Express.js?

Answer: You can handle URL parameters in Express.js by defining routes with parameters using a colon (`:`) prefix. The parameter values can be accessed in the route handler using the `req.params` object.

Example:

javascript

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

99. How do you handle query parameters in Express.js?

Answer: You can handle query parameters in Express.js by accessing them through the `req.query` object. Query parameters are the key-value pairs that follow the `?` in the URL.

Example:

javascript

```
app.get('/users', (req, res) => {
  const name = req.query.name;
  const age = req.query.age;
  // Handle GET request for /users?name=John&age=25
});
```

100. How do you handle environment-specific configuration in Express.js?

Answer: You can handle environment-specific configuration in Express.js by using environment variables. You can set different values for your configuration options based on the environment (e.g., development, production) and access them in your Express.js application.

Example:

javascript

```
const port = process.env.PORT || 3000;
```

```
const databaseUrl = process.env.DATABASE_URL || 'mongodb://localhost/myapp';


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