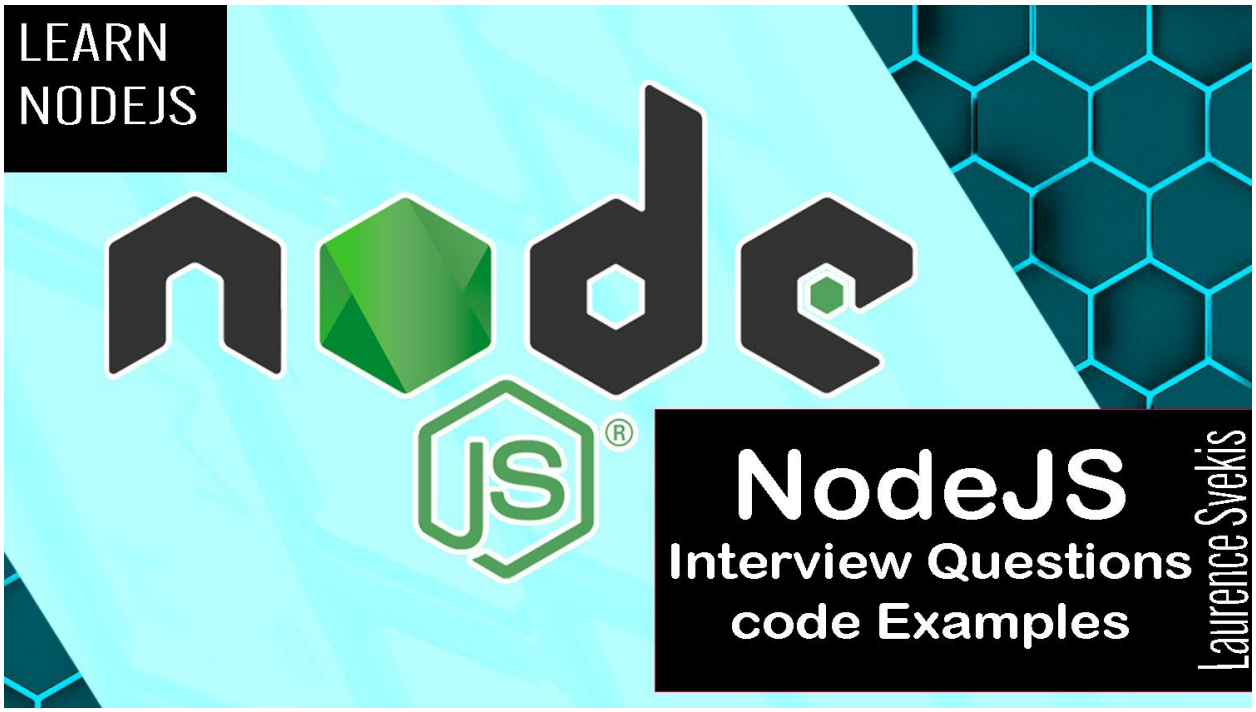
NodeJS Interview questions



What is Node.js and how does it work?

Node.js is an open-source, cross-platform JavaScript runtime environment built on the V8 JavaScript engine of Google Chrome. It allows developers to run JavaScript on the server-side to build scalable, high-performance applications.

Here's a simple example of using Node.js to create a server:

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!');
}).listen(3000);

console.log('Server running at http://localhost:3000/');
```

This code creates an HTTP server that listens on port 3000 and sends a response of 'Hello World!' to any incoming requests.

What is npm and how do you use it in Node.js?

npm (Node Package Manager) is a package manager for Node.js that allows developers to easily install and manage third-party packages or modules. It comes bundled with Node.js and can be accessed through the terminal or command prompt.

To install a package using npm, simply run the following command in your project directory:

```
npm install <package-name>
```

This will install the package and its dependencies, and add it to your project's package.json file.

How do you handle errors in Node.js?

Node.js provides a robust error handling system using try-catch blocks and the Error object. Here's an example of how to handle errors in Node.js:

```
try {
  // Code that may throw an error
} catch (error) {
  console.error('Error occurred: ', error);
```

```
}
```

In this example, we use a try-catch block to catch any errors that may occur in the code inside the try block. If an error is caught, we log it to the console using console.error().

What is middleware in Node.js and how do you use it?

Middleware in Node.js refers to a function that is called before the actual request handler. It can be used to perform operations such as logging, authentication, and validation.

Here's an example of how to use middleware in Node.js:

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

// middleware function
const logRequests = (req, res, next) => {
  console.log('Request received: ', req.url);
  next();
};

app.use(logRequests);
```

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

app.listen(3000, () => {
  console.log('Server running at
http://localhost:3000/');
});
```

In this example, we define a middleware function logRequests that logs the URL of incoming requests. We then use the app.use() method to register the middleware function. When a request is received, the middleware function is called before the actual request handler. The next() function is used to pass control to the next middleware function or request handler.

How do you read data from a file in Node.js?

Node.js provides the fs module for working with the file system. Here's an example of how to read data from a file in Node.js:

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

```
fs.readFile('example.txt', 'utf8', (error, data) => {
  if (error) {
    console.error('Error occurred: ', error);
  } else {
    console.log('Data read successfully: ', data);
  }
});
```

In this example, we use the fs.readFile() method to read data from a file named 'example.txt' in UTF-8 format. When the operation is complete, the callback function is called with an error object (if there is an error) and the data read from the file. We log the error or data to the console depending on the result.

What is callback hell and how do you avoid it in Node.js?

Callback hell is a common issue in Node.js when working with asynchronous code. It occurs when multiple asynchronous operations are nested within each other, resulting in a highly nested and hard-to-read code structure.

Here's an example of callback hell in Node.js:

```
getData((error, data) => {
```

```
  if (!error) {
    processData(data, (error, processedData) => {
      if (!error) {
        saveData(processedData, (error) => {
          if (!error) {
            console.log('Data saved successfully');
          } else {
            console.error('Error occurred: ', error);
          }
        });
      } else {
        console.error('Error occurred: ', error);
      }
    });
  } else {
    console.error('Error occurred: ', error);
  }
});
```

To avoid callback hell in Node.js, you can use several techniques such as promises, async/await, or using a module like async.js. Here's an example of using promises to simplify the above code:

```
getData()
  .then((data) => processData(data))
```

```
  .then((processedData) => saveData(processedData))
  .then(() => console.log('Data saved successfully'))
  .catch((error) => console.error('Error occurred: ',
error));
```

In this example, we use promises to chain the asynchronous operations and handle errors in a more readable and maintainable way.

How do you create a RESTful API in Node.js?

To create a RESTful API in Node.js, you can use a framework like Express.js. Here's an example of how to create a simple RESTful API using Express.js:

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

let todos = [
  { id: 1, text: 'Buy milk' },
  { id: 2, text: 'Walk the dog' },
];
```

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

// GET /todos
app.get('/todos', (req, res) => {
  res.json(todos);
});

// POST /todos
app.post('/todos', (req, res) => {
  const newTodo = { id: todos.length + 1, text:
req.body.text };
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

// PUT /todos/:id
app.put('/todos/:id', (req, res) => {
  const todo = todos.find((t) => t.id ===
parseInt(req.params.id));
  if (todo) {
    todo.text = req.body.text;
    res.json(todo);
  } else {
```

```
    res.status(404).json({ message: 'Todo not found'
});
  }
});

// DELETE /todos/:id
app.delete('/todos/:id', (req, res) => {
  todos = todos.filter((t) => t.id !==
parseInt(req.params.id));
  res.sendStatus(204);
});

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

In this example, we define four endpoints for the CRUD operations on a todo list. We use the express.json() middleware to parse the request body in JSON format. We use the res.json() method to send responses in JSON format and set appropriate HTTP status codes for success or error conditions.

How do you handle sessions in Node.js?

Node.js provides several modules and libraries for handling sessions, such as express-session and cookie-session. Here's an example of how to handle sessions using express-session:

```
const express = require('express');
const session = require('express-session');
const app = express();
const PORT = 3000;

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

app.get('/login', (req, res) => {
  // Check if the user is already logged in
  if (req.session.userId) {
    res.send('You are already logged in');
  } else {
    // Authenticate the user and set the session data
    const userId = 123;
    req.session.userId = userId;
```

```
    res.send('Logged in successfully');
  }
});

app.get('/logout', (req, res) => {
  // Destroy the session data to log the user out
  req.session.destroy();
  res.send('Logged out successfully');
});

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

In this example, we use the express-session middleware to store the session data. We set a secret key to sign the session ID, and configure the middleware to not save uninitialized sessions and not save unchanged sessions.

We define two endpoints, /login and /logout, to set and destroy the session data respectively. When the user logs in, we set the userId property in the session object. When the user logs out, we call the destroy() method on the session object to remove the session data.

How do you handle errors in Node.js?

In Node.js, you can handle errors using try/catch blocks, callbacks, or promises. Here's an example of using try/catch blocks to handle errors in Node.js:

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

try {
  const data = fs.readFileSync('file.txt', 'utf8');
  console.log(data);
} catch (error) {
  console.error('Error occurred: ', error);
}
```

In this example, we use the fs module to read data from a file synchronously. We wrap the code in a try/catch block to catch any errors that might occur during the file read operation. If an error occurs, we log it to the console.

You can also use callback functions to handle errors in Node.js. Here's an example:

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

fs.readFile('file.txt', 'utf8', (error, data) => {
  if (error) {
    console.error('Error occurred: ', error);
  } else {
    console.log(data);
  }
});
```

In this example, we use the fs module to read data from a file asynchronously. We pass a callback function as the last argument to the readFile() method, which is called with two arguments - the error (if any) and the data read from the file. We check for errors in the callback function and log them to the console.

What is event-driven programming in Node.js?

Event-driven programming is a programming paradigm in which the flow of execution is driven by events rather than a sequential flow of instructions. In Node.js, event-driven programming is

implemented using the EventEmitter class, which allows you to create and emit custom events.

Here's how you can use the EventEmitter class to create and emit a custom event in Node.js:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

// Event listener for the 'hello' event
myEmitter.on('hello', (name) => {
  console.log(`Hello, ${name}!`);
});

// Emit the 'hello' event with the name parameter
myEmitter.emit('hello', 'John');
```

In this example, we create a class MyEmitter that extends the EventEmitter class. We then create an instance of the class and define an event listener for the hello event using the on() method. The event listener logs a message to the console with the name parameter.

We then emit the hello event with the name parameter using the emit() method. When the hello event is emitted, the event listener is called and the message is logged to the console.

Event-driven programming is a powerful paradigm that allows you to write efficient and scalable code by handling events as they occur, rather than waiting for an external event to trigger the execution of your code. Node.js is well-suited for event-driven programming because of its non-blocking I/O model, which allows it to handle multiple requests and events simultaneously.