Node.js part 3

7. Validation in Node.js

Why is data validation important in Node.js applications?

Data validation ensures that the input data is correct, secure and in the expected format before processing. It helps prevent invalid data from corrupting the system, reduces bugs & improves security by preventing injection attacks (eg :- SQL injection, XSS)

What are the different types of validation techniques in Node.js?

1) Manual validation - Using JavaScript conditions to check input.

2) Regular expressions - Checking patterns like email, phone number.

3) Schema-based validation - Using libraries like Joi, Zod or Yup.

4) Middleware-based validation - Using express-validator for request validation.

---

3) How can you validate user input using the express-validator package?

```
const { body, validationResult } = require('express-validator');
app.post('/register', [ body('email').isEmail(),
  body('password').isLength({ min: 6 }),
], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
```

```
    res.send('User registered successfully');
});
```

4. How do you handle form validation on both the client and server side?

Client-side: Use JavaScript/HTML5 (required, pattern etc.).

Server side: Use libraries like express-validator or Joi to validate input.

5. What are the common security risks related to invalid input in Node.js?

- SQL Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Buffer Overflow Attacks.

6. How can you perform custom validations in Express.js?

```
app.post('/signup', [body('username').custom(value => {
    if (value.includes('admin')) {
        throw new Error('Username cannot contain "admin"');
    }
    return true;
}),
], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors:
    errors.array() });
    res.send('User registered');
});
```

7. What is the role of middleware in request validation?

- Middleware intercepts request before they reach the controller.
- It can validate, sanitize or modify input
- Eg: express.

8. Error Handling in Node.js

1. What is error handling and why is it important in Node.js?

Error handling refers to detecting managing and responding to errors in a program. In Node.js proper error handling is more important because:

+ It prevents application crashes.
+ It improves user experience by providing meaning error messages.

2. What are synchronous and asynchronous errors in Node.js?

* Synchronous errors occur immediately and can be caught using try-catch.
- Asynchronous errors occur in callbacks, promises or async function and must be handled using .catch() or error-first callback.

3. How does the try-catch block work in Node.js?

try-catch is used to handle synchronous errors:

```
try{
    let result = JSON.parse('invalid json');
}
```

```
catch (error) {
  console.error('Caught an error:', error.message);
}
```

4. What is the purpose of error handling middleware in express.js?

Error-handling middleware in Express is used to centralize error management. It has four parameters:

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

5. How do you handle errors in asynchronous operations?

- For callbacks: Use error-first callback convention ((err, result) => {}).
- For promises: Use .catch().
- For async/await: Use try-catch.

```
async function fetchData() {
  try {
    let data = await fetch('https://api.example.com');
    return await data.json();
  } catch (error) {
    console.error('Error:', error.message);
  }
}
```

6. What is the difference between operational error and programmer error?

- Operational errors: Errors caused by external factors. These should be handled gracefully.
- Programmer errors: Bugs in the code. These should be fixed in the codebase.

7. How can you define a global error handler in an Express application?

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

9. File Handling in Node.js

1. What are the different ways to handle file in Node.js?

- Using the fs module.
- Using streams for large files.
- Using third-party libraries like fs-extra.

2. What is the role of the fs module in Node.js?

The fs module (File System) provides methods to read, write, update & delete files in Node.js.

3. How do you read a file asynchronously in Node.js?

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

```
fs.readFile('file.txt', 'utf-8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File content:', data);
});
```

4. What is the difference between synchronous and asynchronous file operations?

Synchronous (fs.readFileSync) blocks execution until the file operation completes.

Asynchronous (fs.readFile) does not block execution and uses a callback or promise.

5. How can you create and write data to a file in Node.js?

```
fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {
  if (err) throw err;
  console.log('File');
});
```

6. What is the purpose of file streaming in Node.js?

File Streaming allows handling large files efficiently without loading them into memory at once.

```
const readStream = fs.createReadStream('largeFile.txt');
readStream.on('data', (chunk) => console.log('Received
```

7- How can you delete a file in Node.js using fs module?

```
fs.unlink('file.txt', (err) => {
    if (err) throw err;
    console.log('File deleted');
});
```

---

10. Callback in Node.js

1- What is a callback function in Node.js?

A callback is a function passed as an argument to another function, which is executed later.

---

2. Why are callbacks commonly used in Node.js applications?

Node.js uses asynchronous, non-blocking I/O, so callbacks help execute tasks after completion without blocking execution.

---

3. What is the callback hell problem, and how can it be avoided?

Callback hell occurs when multiple nested callbacks make code unreadable:

```
fs.readFile('file.txt', (err, data) => {
    if (err) return console.error(err);
    fs.writeFile('output.txt', data, (err) => {
        if (err) return console.error(err);
        console.log('Done');
    });
});
```

Solutions:

- Use Promises
- Use async/await
- Use modular functions.

4. How do you convert a callback based function into a promise in Node.js?

```
const { promisify } = require('util');
const readFile = promisify(fs.readFile);
readFileAsync('file.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

5. What is the role of util.promisify() function?

It converts callback-based functions into Promise-based functions, allowing usage of .then() or async/await.

6. How does the event loop manage callback execution in Node.js?

- Callbacks are added to the event loop queue when asynchronous operations complete.
  - Timers
  - Pending callbacks
  - I/O callbacks
  - Check phase
  - Close callbacks.

7. How do callbacks differ from promises and async/wait in handling asynchronous operation?

Feature	Callbacks	Promises	Async/Await
Readability	Hard to read	Better	Best
Error Handling	Manual (error-first)	.catch()	try-catch
Flexibility	Low	Medium	High
Chainability	No	Yes	Yes

Eg :-

```
async function readFile() {
  try {
    let data = await fs.promises.readFile('file.txt', 'utf
    console.log(data);
  } catch (error) {
    console.error('Error:', error.message);
  }
}
```