Great 👍 Let’s cover **Topic 20: Error Handling & Debugging in Node.js** in the same **detailed format** as the previous notes.

---

20. Error Handling & Debugging in Node.js

Error handling is critical in Node.js because of its asynchronous, event-driven nature. Poor handling can crash the application or leave unhandled rejections. Debugging tools help diagnose these issues effectively.

---

1. Types of Errors in Node.js

- **Synchronous Errors**
  - Occur immediately in code execution.
  - Example: `JSON.parse("invalid")`.
- **Asynchronous Errors**
  - Occur in callbacks, promises, or async functions.
  - Example: failed DB query, failed HTTP request.

---

2. Error Handling Techniques

a) Try/Catch (Synchronous & Async/Await)

```
try {
  let data = JSON.parse('{ "name": "Shubham" }');
  console.log(data.name);
} catch (err) {
  console.error("Invalid JSON:", err.message);
}
```

For **async/await**:

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

b) Callbacks (Error-First Pattern)

- Convention: `(err, result)` as arguments.

```
fs.readFile("file.txt", (err, data) => {
  if (err) return console.error("File read error:", err.message);
  console.log(data.toString());
});
```

c) Promises (then/catch)

```
fetch("https://api.example.com")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error("Promise error:", err.message));
```

d) Centralized Error Handling in Express

- Middleware to catch errors:

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

- Async route wrapper:

```
const asyncHandler = fn => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};
```

3. Handling Uncaught Errors

a) Uncaught Exceptions

```
process.on("uncaughtException", err => {
  console.error("Uncaught Exception:", err);
  process.exit(1); // restart app
});
```

b) Unhandled Promise Rejections

```
process.on("unhandledRejection", reason => {
  console.error("Unhandled Rejection:", reason);
});
```

⚠ Best practice: **restart the app** using PM2/Docker if a fatal error occurs.

---

4. Logging Errors

- **Winston** (structured logging)

```
const { createLogger, transports } = require("winston");

const logger = createLogger({
  transports: [
    new transports.Console(),
    new transports.File({ filename: "errors.log" })
  ]
});

logger.error("Critical error occurred!");
```

- **Morgan** (HTTP request logging)

```
const morgan = require("morgan");
app.use(morgan("combined"));
```

---

5. Debugging Tools

a) Node.js Inspector

- Run app with:

```
node --inspect app.js
```

- Open `chrome://inspect` in Chrome.

b) VS Code Debugger

- Use `launch.json` to set breakpoints.
- Step through code execution.

c) Console Utilities

- `console.log()`, `console.error()`
- `console.table()` for arrays/objects
- `console.time()` / `console.timeEnd()` for performance tracking

---

6. Best Practices

- Always validate inputs (Joi, express-validator).
- Use **centralized error middleware** in Express.
- Separate **operational errors** (e.g., DB connection failure) from **programmer errors** (e.g., null pointer).
- Log errors with **context** (timestamp, request details).
- Gracefully shut down the app on fatal errors & restart with PM2/Docker.
- Never expose stack traces to clients in production.

☑ This completes **Error Handling & Debugging in Node.js** with examples, tools, and best practices.

Do you want me to move on to **Topic 21: Deployment & DevOps in Node.js** next?