Node.js Advanced Concepts:

1) Middleware

- Definition- Functions that have access to the request object (req), the response object (res), & the next middleware function in application's request-response cycle.
- Usage- Commonly used in Express.js for tasks such as logging, authentication & error handling
- Example:

```
function logger(req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next(); }
  app.use(logger);
```

2) Asynchronous Programming

- Callbacks: Functions passed as arguments to other functions executed after an asynchronous operation completes.

e.g.
```
fs.readFile('file.txt', (err, data) => {
  if(err) throw err;
  console.log(data.toString()); });
```

- Promises: Objects representing the eventual completion (or failure) of an asynchronous operation.

e.g.
```
let promise = new Promise((resolve, reject) => {
  if(success) { resolve(result); }
  else { reject(error); } });
promise.then(result => console.log(result)).catch(error => console.
                                                   error(error));
```

Async Await: Syntatic sugar for promises, making asynchronous code look synchronous.

eg:
```
async function fetchData(){
  try {
    let data = await fetch(url);
    console.log(data);
  } catch (error){
    console.error(error); }}
  fetchData();
```

3) Event-Driven Architecture:-

- Events → events are emitted & listeners are executed.
- Event Emitter: Core module for working with events.

eg
```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('event', ()=>{
  console.log('An event occured!'); });
  emitter.emit('event');
```

4) Streams

- Definition- Objects used to handle reading or writing data continously.
- Types - Readable, writable, Duplex & Tranform streams
- e.g. →
```
const fs= require('fs');
const readableStream= fs.createReadStream('file.txt');
readableStream.on('data', (chunk) => {
  console.log('Received ${chunk.length} bytes of data.');
});
```

5) Buffer

- Definition - Raw memory allocation outside V8 heap, used to handle binary data.

- e.g
```
let buffer = Buffer.alloc(10);
buffer.write('Hello');
console.log(buffer.toString());
```

6) Cluster

- Definition: Node.js allows to create child processes that share the same server port to take advantage of multi-core systems.

- e.g →
```
const cluster = require('cluster');
const http = require('http');
if(cluster.isMaster){
 for(let i=0; i<numCPUs; i++){
  cluster.fork(); }}
 else{ http.createServer((req,res)=>{
  res.writeHead(200);
  res.end('Hello world\n'); }).listen(8000);
}
```

7) Error Handling

- Synchronous: Use try...catch blocks.
```
try{
 let data= fs.readFileSync('file.txt');
 console.log(data.toString()); }
catch(err){ console.error(err); }
```

- Asynchronous: Handle errors in callbacks or promise chains.
```
fs.readFile('file.txt', (err, data)=>{
 if(err){ console.error(err);
          return; }
 console.log(data.toString());
});
```

8) Testing & Debugging

- Testing: Use frameworks like Mocha, Jest for writing & running tests.

e.g.
```
const assert = require('assert');
describe('Array', function() {
  it('should return -1 when value is not present', function() {
    assert.equal([1,2,3].indexOf(4), -1);
  });
});
```

- Debugging: Use Node.js inspector, `node --inspect`, & debugging tools like VS code.

9) HTTP / HTTPS modules

- Creating a Server: Use http or https modules

e.g.
```
const http = require('http');
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(3000, () => {
  console.log('Server running at http://127.0.0.1:3000/');
});
```

10) Modules & NPM

- Modules: Use 'require' to include built-in, local or third party modules
- NPM: NodePackageManager for managing project dependencies:

```
npm init
npm install express
```