# Node.js Backend Developer Interview Notes (Beginner to Senior)

## 🔹 1. What is Node.js?

Node.js is a **JavaScript runtime** built on **Chrome's V8 engine** that lets you run JavaScript on the server. It's **event-driven**, **non-blocking**, and perfect for **scalable backend applications**.

---

## 🔹 2. Express.js Basics

Express.js is a **minimal and flexible web framework** for Node.js to build APIs quickly.

### ✅ Example: Basic REST API with Express

```
const express = require('express');
const app = express();
app.use(express.json());

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

app.listen(3000, () => console.log('Server is running'));
```

---

## 🔹 3. MongoDB with Mongoose

MongoDB is a NoSQL database. Mongoose is an ODM for MongoDB in Node.js.

### ✅ Connect to MongoDB

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myapp');
```

### ✅ Create a Schema & Model

```
const UserSchema = new mongoose.Schema({ name: String });
const User = mongoose.model('User', UserSchema);
```

---

## 🔹 4. CRUD API Example (Full)

```
// models/user.js
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({ name: String });
module.exports = mongoose.model('User', UserSchema);

// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const User = require('../models/user');

router.post('/', async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});

router.get('/', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

module.exports = router;

// app.js
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const userRoutes = require('./routes/userRoutes');

mongoose.connect('mongodb://localhost:27017/myapp');
app.use(express.json());
app.use('/users', userRoutes);

app.listen(3000);
```

---

## 🔹 5. Async/Await and Error Handling

```
app.get('/data', async (req, res) => {
  try {
    const data = await SomeModel.find();
    res.json(data);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

---

## 🔹 6. Redis Caching Example

```
const redis = require('redis');
const client = redis.createClient();
const { promisify } = require('util');
const getAsync = promisify(client.get).bind(client);
const setAsync = promisify(client.set).bind(client);

app.get('/cache-example', async (req, res) => {
  const key = 'some_key';
  const cached = await getAsync(key);
  if (cached) return res.json(JSON.parse(cached));

  const data = await fetchFromDB();
  await setAsync(key, JSON.stringify(data), 'EX', 60);
  res.json(data);
});
```

---

## 🔹 7. MongoDB Indexing

### ✅ Create Index

```
db.users.createIndex({ email: 1 });
```

### ✅ Check Index Usage

```
db.users.find({ email: 'test@example.com' }).explain("executionStats")
```

---

## 🔹 8. Scalability in Node.js

### ✅ Techniques:

- Use `cluster` or PM2 to scale across CPU cores.
- Use Redis to cache frequent DB queries.
- Use MongoDB sharding and replica sets.

---

## 🔹 9. Performance Tuning

- Avoid blocking code (heavy loops)
- Use async/await
- Optimize DB queries
- Use `.lean()` in Mongoose for read-only data

---

## 🔹 10. Microservices vs Tiered Architecture

| Topic | Microservices | Tiered Architecture |
|---|---|---|
| Deployment | Independently per service | As a single unit (monolith) |
| Scaling | Individual services | Entire app together |
| Database | Each service can have its own | Usually one shared DB |
| Communication | HTTP or message queue (Kafka, RabbitMQ) | Function calls |

---

## 🔹 11. Deployment Notes

### ✅ Using Docker

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "app.js"]
```

### ✅ Common Deployment Options:

- **Heroku**: Easy but limited
- **Render**: Free for testing

- **VPS (e.g., DigitalOcean)**: Full control
- **CI/CD**: GitHub Actions, Jenkins, etc.

---

## 🔹 12. Bonus Tools & Tips

- Use **Postman** or **Thunder Client** to test APIs
- Use **dotenv** to manage environment variables
- Use **helmet** for security headers
- Use **morgan** for logging

---

## ✅ Final Tip:

Practice building 1–2 full-stack apps with:

- Node.js + Express
- MongoDB + Mongoose
- Redis for caching
- JWT for authentication
- Docker for deployment

# Node.js Backend Developer Interview Notes (Beginner to Senior)

## 🔹 1. What is Node.js?

Node.js is a **JavaScript runtime** built on **Chrome's V8 engine** that lets you run JavaScript on the server. It's **event-driven**, **non-blocking**, and perfect for **scalable backend applications**.

---

## 🔹 2. Express.js Basics

Express.js is a **minimal and flexible web framework** for Node.js to build APIs quickly.

### ✅ Example: Basic REST API with Express

```
const express = require('express');
const app = express();
app.use(express.json());

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

app.listen(3000, () => console.log('Server is running'));
```

---

## 🔹 3. MongoDB with Mongoose

MongoDB is a NoSQL database. Mongoose is an ODM for MongoDB in Node.js.

### ✅ Connect to MongoDB

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myapp');
```

### ✅ Create a Schema & Model

```
const UserSchema = new mongoose.Schema({ name: String });
const User = mongoose.model('User', UserSchema);
```

---

## 🔹 4. CRUD API Example (Full)

```
// models/user.js
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({ name: String });
module.exports = mongoose.model('User', UserSchema);

// routes/userRoutes.js
const express = require('express');
```

```
const router = express.Router();
const User = require('../models/user');

router.post('/', async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});

router.get('/', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

module.exports = router;

// app.js
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const userRoutes = require('./routes/userRoutes');

mongoose.connect('mongodb://localhost:27017/myapp');
app.use(express.json());
app.use('/users', userRoutes);

app.listen(3000);
```

---

## 🔹 5. Async/Await and Error Handling

```
app.get('/data', async (req, res) => {
  try {
    const data = await SomeModel.find();
    res.json(data);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

---

## 🔹 6. Redis Caching Example

```
const redis = require('redis');
const client = redis.createClient();
const { promisify } = require('util');
```

```
const getAsync = promisify(client.get).bind(client);
const setAsync = promisify(client.set).bind(client);

app.get('/cache-example', async (req, res) => {
  const key = 'some_key';
  const cached = await getAsync(key);
  if (cached) return res.json(JSON.parse(cached));

  const data = await fetchFromDB();
  await setAsync(key, JSON.stringify(data), 'EX', 60);
  res.json(data);
});
```

---

## 🔹 7. MongoDB Indexing

### ✅ Create Index

```
db.users.createIndex({ email: 1 });
```

### ✅ Check Index Usage

```
db.users.find({ email: 'test@example.com' }).explain("executionStats")
```

---

## 🔹 8. Schema Design Best Practices

- Keep schema flat (avoid deep nesting)
- Use references (`ObjectId`) for relationships
- Use `enum` for fixed values
- Use `timestamps` to track creation/update

### ✅ Example

```
const PostSchema = new mongoose.Schema({
  title: String,
  content: String,
  status: { type: String, enum: ['draft', 'published'], default: 'draft' },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
}, { timestamps: true });
```

---

## 🔹 9. Scalability in Node.js

### ✅ Techniques:

- Use `cluster` or PM2 to scale across CPU cores.
- Use Redis to cache frequent DB queries.
- Use MongoDB sharding and replica sets.

---

## 🔹 10. MongoDB Sharding

Sharding splits large collections across multiple machines to scale reads/writes.

### ✅ Key Concepts

- Shard Key: Determines how data is distributed.
- Config Server: Stores metadata.
- Mongos: Query router.

### ✅ Command to Enable Sharding

```
sh.enableSharding("myDatabase")
sh.shardCollection("myDatabase.myCollection", { userId: 1 })
```

---

## 🔹 11. Performance Tuning

- Avoid blocking code (heavy loops)
- Use async/await
- Optimize DB queries
- Use `.lean()` in Mongoose for read-only data

---

## 🔹 12. Microservices vs Tiered Architecture

| Topic | Microservices | Tiered Architecture |
|---|---|---|
| Deployment | Independently per service | As a single unit (monolith) |
| Scaling | Individual services | Entire app together |
| Database | Each service can have its own | Usually one shared DB |
| Communication | HTTP or message queue (Kafka, RabbitMQ) | Function calls |

---

## 🔹 13. Deployment Notes

### ✅ Using Docker

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "app.js"]
```

### ✅ Common Deployment Options:

- **Heroku**: Easy but limited
- **Render**: Free for testing
- **VPS (e.g., DigitalOcean)**: Full control
- **CI/CD**: GitHub Actions, Jenkins, etc.

### ✅ PM2 for Process Management

```
npm install -g pm2
pm2 start app.js -i max  # Cluster mode
```

---

## 🔹 14. Bonus Tools & Tips

- Use **Postman** or **Thunder Client** to test APIs
- Use **dotenv** to manage environment variables
- Use **helmet** for security headers
- Use **morgan** for logging

---

## ✅ Final Tip:

Practice building 1–2 full-stack apps with:

- Node.js + Express
- MongoDB + Mongoose
- Redis for caching
- JWT for authentication
- Docker for deployment

```
// ----------------------------
// MongoDB Setup (Connection)
// ----------------------------
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
.catch(err => console.error('Connection error:', err));

// ----------------------------
// JWT Middleware
// ----------------------------
const jwt = require('jsonwebtoken');

const authMiddleware = (req, res, next) => {
```

```
  // Get token from headers
  const token = req.headers.authorization?.split(" ")[1];

  // Fallback (if no token found)
  if (!token) return res.status(401).json({ message: 'No token provided' });

  try {
    // Verify token
    const decoded = jwt.verify(token, 'your_jwt_secret');
    req.user = decoded; // Attach user info
    next();
  } catch (err) {
    res.status(403).json({ message: 'Invalid or expired token' });
  }
};

// ----------------------------
// Redis Setup and Example
// ----------------------------
const redis = require('redis');
const client = redis.createClient();

client.connect();

// Set and get data (CRUD - Read/Write)
// Cache user data
const cacheUser = async (userId, data) => {
  await client.set(`user:${userId}`, JSON.stringify(data));
};

const getCachedUser = async (userId) => {
  const data = await client.get(`user:${userId}`);
  return JSON.parse(data);
};

// ----------------------------
// Mongoose Models Example
// ----------------------------
const CategorySchema = new mongoose.Schema({
  name: String
});
const SubCategorySchema = new mongoose.Schema({
  name: String,
  categoryId: { type: mongoose.Schema.Types.ObjectId, ref: 'Category' }
});

const Category = mongoose.model('Category', CategorySchema);
const SubCategory = mongoose.model('SubCategory', SubCategorySchema);
```

```
// ----------------------------
// Fetching Subcategories by Category ID
// ----------------------------
const getSubcategories = async (categoryId) => {
  const subcategories = await SubCategory.find({ categoryId });
  return subcategories;
};

// ----------------------------
// Async/Await, Promise, Callback, Fallback Concepts
// ----------------------------

// Async/Await (waits for promises to resolve)
async function getUserData(id) {
  try {
    const user = await User.findById(id); // Promise-based
    return user;
  } catch (error) {
    console.error(error);
  }
}

// Promise (then/catch chaining)
User.findById('123')
  .then(user => console.log(user))
  .catch(err => console.error(err));

// Callback (older way)
User.findById('123', (err, user) => {
  if (err) return console.error(err);
  console.log(user);
});

// Fallback (backup method)
const getUsername = (user) => {
  return user?.name || 'Guest'; // fallback to 'Guest' if name is missing
};
```

Here's your full **interview-ready backend development notes** with concepts, meanings in brackets, and implementation code:

---

## 🔹 1. Node.js Architecture

- **Single-threaded** (runs one operation at a time in the main thread)
- **Non-blocking** (doesn't wait for an operation to complete)
- **Event-driven** (executes logic based on events like HTTP requests)
- Uses the **event loop** to handle multiple tasks asynchronously.

### Example:

```
// Non-blocking
setTimeout(() => {
  console.log("Runs after 2 seconds");
}, 2000);
console.log("Runs first");
```

---

## 🔹 2. async/await

### Concepts:

- `async`: (declares a function will return a **promise**)
- `await`: (waits for a **promise** to resolve/reject)
- Replaces **callback hell** or `.then()` chains.
- Still **non-blocking** — handled through event loop.

### Example:

```
async function getUser() {
  try {
    const res = await fetch('https://api.example.com/user');
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error(err); // fallback (backup logic on failure)
  }
}
getUser();
```

---

## 🔹 3. Promises, Callbacks, Fallback

- **Promise**: (a placeholder for a future result)
- **Callback**: (a function passed as argument to be executed later)
- **Fallback**: (an alternative if the main option fails)

```
// Callback
fs.readFile('file.txt', (err, data) => {
  if (err) {
    // fallback
    console.log('Read failed, trying backup');
  } else {
    console.log(data);
  }
});
```

---

## 🔹 4. JWT (JSON Web Token)

### Concept:

- Used for **authentication**
- Server gives signed token on login.
- Client sends token in headers → server verifies.

### Flow:

1. User logs in
2. Server signs JWT with secret key
3. JWT sent to client
4. Client sends token with every request in headers

### Example:

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

```
// Sign
const token = jwt.sign({ userId: 123 }, 'secretkey', { expiresIn: '1h' });

// Middleware
function auth(req, res, next) {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).send('No token');

  try {
    const user = jwt.verify(token, 'secretkey');
    req.user = user;
    next();
  } catch (err) {
    res.status(401).send('Invalid token');
  }
}
```

---

## 🔹 5. Redis Caching

### Concept:

- Redis = in-memory key-value store
- Caches frequent or expensive queries
- Speeds up performance, reduces DB load

### When to use:

- Expensive DB queries
- Rate limiting
- Session storage

### Example:

```
const redis = require('redis');
const client = redis.createClient();

app.get('/product/:id', async (req, res) => {
  const key = `product:${req.params.id}`;
  client.get(key, async (err, data) => {
```

```
    if (data) return res.json(JSON.parse(data));

    const product = await Product.findById(req.params.id);
    client.setex(key, 3600, JSON.stringify(product));
    res.json(product);
  });
});
```

---

## 🔹 6. MongoDB Indexing

### Concept:

- Index = special data structure to speed up queries
- Works like a book index

### Use cases:

- Searching frequently used fields
- Sorting or filtering

### Types:

- Single field
- Compound (multi-field)
- Text index (for searching text)

### Example:

```
// Create index
db.users.createIndex({ email: 1 }); // ascending

// In Mongoose
UserSchema.index({ email: 1 });
```

---

## 🔹 7. Schema Design (MongoDB)

### Best Practices:

- Use ObjectId for relations (references)
- Normalize if data is shared (like users)
- Embed only when data is tightly coupled

### Example:

```
// Users collection
{
  _id: ObjectId,
  name: "John"
}

// Posts collection
{
  title: "Post",
  userId: ObjectId("...")
}
```

---

## 🔹 8. Sharding (MongoDB)

### Concept:

- Splits data across **multiple servers**
- Improves **scalability** and **performance**

### When to use:

- Very large datasets
- High read/write volume

### Example:

```
# Enable sharding
sh.enableSharding("myDatabase")

# Shard a collection
sh.shardCollection("myDatabase.myCollection", { _id: "hashed" })
```

---

## 🔹 9. Microservices vs Tiered Architecture

| Feature | Microservices | Tiered (Monolith) |
|---|---|---|
| Structure | Small, independent services | Single app with layers |
| Scalability | Scalable per service | Whole app needs scaling |
| Deployment | Each service separately | One big deployment |
| Complexity | Higher | Lower |
| Communication | Over HTTP (API) or message queue | Internal function calls |

---

## 🔹 10. Deployment Steps (Node.js + Express)

Use **PM2** to run app

```
 npm install -g pm2
pm2 start app.js
```

1. 
2. Reverse proxy with **Nginx**
3. Set up SSL using **Let's Encrypt**
4. Environment variables: `.env` + `dotenv`
5. Use **Docker** for containerization (optional)
6. Push code to server via Git or CI/CD

---

Would you like me to:

✅ Turn this into a downloadable **PDF**
📘 Add **mock interview questions** for each section
🎯 Generate **flashcards or quiz** for fast revision?

# Node.js Backend Developer Interview Notes (Beginner to Senior)

## 🔹 1. What is Node.js?

Node.js is a **JavaScript runtime** built on **Chrome's V8 engine** that lets you run JavaScript on the server. It's **event-driven**, **non-blocking**, and perfect for **scalable backend applications**.

---

## 🔹 2. Express.js Basics

Express.js is a **minimal and flexible web framework** for Node.js to build APIs quickly.

### ✅ Example: Basic REST API with Express

```
const express = require('express');
const app = express();
app.use(express.json());

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

app.listen(3000, () => console.log('Server is running'));
```

---

## 🔹 3. MongoDB with Mongoose

MongoDB is a NoSQL database. Mongoose is an ODM for MongoDB in Node.js.

### ✅ Connect to MongoDB

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myapp');
```

### ✅ Create a Schema & Model

```
const UserSchema = new mongoose.Schema({ name: String });
const User = mongoose.model('User', UserSchema);
```

---

## 🔹 4. CRUD API Example (Full)

```
// models/user.js
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({ name: String });
module.exports = mongoose.model('User', UserSchema);

// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const User = require('../models/user');

router.post('/', async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});

router.get('/', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

module.exports = router;

// app.js
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const userRoutes = require('./routes/userRoutes');

mongoose.connect('mongodb://localhost:27017/myapp');
app.use(express.json());
app.use('/users', userRoutes);

app.listen(3000);
```

---

## 🔹 5. Async/Await and Error Handling

```
app.get('/data', async (req, res) => {
  try {
    const data = await SomeModel.find();
    res.json(data);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

---

## 🔹 6. Redis Caching Example

```
const redis = require('redis');
const client = redis.createClient();
const { promisify } = require('util');
const getAsync = promisify(client.get).bind(client);
const setAsync = promisify(client.set).bind(client);

app.get('/cache-example', async (req, res) => {
  const key = 'some_key';
  const cached = await getAsync(key);
  if (cached) return res.json(JSON.parse(cached));

  const data = await fetchFromDB();
  await setAsync(key, JSON.stringify(data), 'EX', 60);
  res.json(data);
});
```

---

## 🔹 7. MongoDB Indexing

### ✅ Create Index

```
db.users.createIndex({ email: 1 });
```

### ✅ Check Index Usage

```
db.users.find({ email: 'test@example.com' }).explain("executionStats")
```

---

## 🔹 8. Schema Design Best Practices

- Keep schema flat (avoid deep nesting)
- Use references (`ObjectId`) for relationships
- Use `enum` for fixed values
- Use `timestamps` to track creation/update

### ✅ Example

```
const PostSchema = new mongoose.Schema({
  title: String,
```

```
  content: String,
  status: { type: String, enum: ['draft', 'published'], default: 'draft' },
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
}, { timestamps: true });
```

---

## 🔹 9. Scalability in Node.js

### ✅ Techniques:

- Use `cluster` or PM2 to scale across CPU cores.
- Use Redis to cache frequent DB queries.
- Use MongoDB sharding and replica sets.

---

## 🔹 10. MongoDB Sharding

Sharding splits large collections across multiple machines to scale reads/writes.

### ✅ Key Concepts

- Shard Key: Determines how data is distributed.
- Config Server: Stores metadata.
- Mongos: Query router.

### ✅ Command to Enable Sharding

```
sh.enableSharding("myDatabase")
sh.shardCollection("myDatabase.myCollection", { userId: 1 })
```

---

## 🔹 11. Performance Tuning

- Avoid blocking code (heavy loops)
- Use async/await

- Optimize DB queries
- Use `.lean()` in Mongoose for read-only data

---

## 🔹 12. Microservices vs Tiered Architecture

| Topic | Microservices | Tiered Architecture |
|---|---|---|
| Deployment | Independently per service | As a single unit (monolith) |
| Scaling | Individual services | Entire app together |
| Database | Each service can have its own | Usually one shared DB |
| Communication | HTTP or message queue (Kafka, RabbitMQ) | Function calls |

---

## 🔹 13. Deployment Notes

### ✅ Using Docker

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "app.js"]
```

### ✅ Common Deployment Options:

- **Heroku**: Easy but limited
- **Render**: Free for testing
- **VPS (e.g., DigitalOcean)**: Full control
- **CI/CD**: GitHub Actions, Jenkins, etc.

### ✅ PM2 for Process Management

```
npm install -g pm2
pm2 start app.js -i max  # Cluster mode
```

---

## 🔹 14. Bonus Tools & Tips

- Use **Postman** or **Thunder Client** to test APIs
- Use **dotenv** to manage environment variables
- Use **helmet** for security headers
- Use **morgan** for logging

---

## ✅ Final Tip:

Practice building 1–2 full-stack apps with:

- Node.js + Express
- MongoDB + Mongoose
- Redis for caching
- JWT for authentication
- Docker for deployment