});

UNIT 3 – Advance Node.js Concepts Assignment 3

Q1. Explain how bi-directional communication is implemented in a simple chat application using Socket.io. What are the key steps involved in setting up and managing real-time communication between the server and clients?

Answer:
Bi-directional communication in Socket.io is implemented using WebSockets (and fallback mechanisms like long-polling). Socket.io allows the server and client to send messages to each other at any time using event-based communication.

Key Steps:

1. **Install Socket.io on server and client**
   - Server: `npm install socket.io`
   - Client: load `/socket.io/socket.io.js`

Create Node.js server and attach Socket.io

```
 const http = require('http');
const express = require('express');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server);
```

2.

Handle client connection event

```
 io.on('connection', (socket) => {
  console.log('User connected: ' + socket.id);
});
```

3.
4. **Send events from client → server and server → client**
   Example:

- Client: `socket.emit("message", "Hello");`

Server:

socket.on("message", msg => console.log(msg));

-

Broadcast events to all connected users

io.emit("chatMessage", msg);

5.

Handle disconnection

socket.on("disconnect", () => console.log("User disconnected"));

6.
7. **Manage mapping for private messages**
   Store username ↔ socket.id.

8. **Scaling with Redis Adapter**
   Needed when multiple Node.js instances run behind a load balancer.

Q2. Describe the essential steps to deploy a Node.js application to a production environment. What strategies would you use to ensure the scalability and reliability of the deployed application?

Answer:
Deploying a Node.js application to production involves preparing the app, setting up a server environment, securing the application, and ensuring continuous monitoring.

Essential Steps:

1. **Set environment variables**

   - Use `NODE_ENV=production`

    - Add `.env` for secrets

2. **Use a process manager (PM2 / systemd)**
    - Keeps app alive
    - Supports clustering

```
pm2 start app.js -i max
```

3.
4. **Set up a reverse proxy (Nginx)**
    - Handles HTTPS
    - Provides caching
    - Load balancing

5. **Containerization (optional but recommended)**
   Using Docker for consistent environments.

6. **Use CI/CD pipeline**
    - Automated testing
    - Auto-deployment
    - Rollback system

7. **Enable monitoring**
    - Tools: PM2 Monitor, Grafana, Prometheus, New Relic

Scalability & Reliability Strategies:

✔ **Horizontal Scaling** using multiple Node instances
✔ **Load Balancing** using Nginx / cloud load balancers
✔ **Redis for caching** to reduce DB load
✔ **Database optimization & indexing**
✔ **Autoscaling** (Kubernetes / AWS Elastic Beanstalk)
✔ **Graceful shutdown + health checks**
✔ **Fault tolerance using message queues**

Q3. Identify and explain two common security threats in Node.js applications, such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). How can these threats be mitigated through proper security measures?

Answer:
Node.js applications are often targeted by browser-based attacks. Two major threats are:

---

1. Cross-Site Scripting (XSS)

Meaning:
Attacker injects malicious JavaScript into a webpage, which runs on the victim's browser.

Mitigation:

- Escape and sanitize all user input
- Use templating engines that auto-escape
- Add Content Security Policy (CSP) headers

Use `helmet` middleware

```
 const helmet = require('helmet');
app.use(helmet());
```

-

---

2. Cross-Site Request Forgery (CSRF)

Meaning:
An attacker tricks a logged-in user into performing unintended actions (like money transfer).

Mitigation:

Use CSRF tokens

```
 const csurf = require('csurf');
app.use(csurf());
```

-
- Use SameSite cookies
- Validate request origin via CORS
- Prefer JWT-based authorization for APIs

Q4. Discuss the different methods of error handling in Node.js, including the use of callbacks, try/catch blocks, and error propagation. How can these methods be combined effectively to manage errors in a Node.js application, and what are the advantages and disadvantages of each approach?

Answer:

1. Callbacks (Old Method)

```
fs.readFile("file.txt", (err, data) => {
  if (err) return console.error(err);
});
```

- ✔ Works everywhere
- ✖ Leads to callback hell
- ✖ Hard to handle complex flows

---

2. Promises

```
doTask()
 .then(result => console.log(result))
 .catch(err => console.error(err));
```

- ✔ Chainable
- ✔ Cleaner than callbacks
- ✖ Must handle `.catch()` everywhere

3. Async/Await + try/catch

```
try {
  let data = await getData();
} catch (err) {
  console.error(err);
}
```

- ✔ Best for readability
- ✔ Synchronous-like
- ✖ Forgetting a `try/catch` allows unhandled rejections

4. Centralized Error Middleware (Express)

```
app.use((err, req, res, next) => {
  res.status(500).send(err.message);
});
```

- ✔ One place for all errors
- ✔ Best for API responses

Effective Combination

- Use **async/await** in routes
- Use **centralized middleware** for final error processing
- Use **try/catch** for known recoverable errors
- Throw errors to propagate them upwards

Q5. Analyze the techniques for identifying and resolving performance bottlenecks in a Node.js application. Discuss how caching and load balancing can be implemented to optimize the performance and scalability of a Node.js application.

Answer:

Identifying Performance Bottlenecks

1. **Profiling Tools**
    - Node.js built-in profiler
    - Chrome DevTools
    - Clinic.js / 0x
2. **Monitoring Event Loop Lag**
    - Detect CPU-heavy operations
3. **Load Testing**
    - Tools: JMeter, Artillery, k6
4. **Check External Dependencies**
    - Slow database queries
    - Slow API calls

Resolving Bottlenecks

- Move CPU-heavy tasks to worker threads
- Use connection pooling
- Add database indexes
- Avoid synchronous code in event loop
- Use pagination instead of loading large data

Caching

✔ Improve response speed
✔ Reduce DB load

Options:

Redis Cache (recommended)

```
redis.set("user:1", JSON.stringify(data));
```

1.
2. **In-memory Cache (LRU)**
   Only useful when running a single process

3. **HTTP Cache (ETag, Cache-Control)**

Load Balancing

Used when running multiple Node.js instances.

Methods:

- **Nginx Load Balancer**

- **Cloud LB (AWS, GCP, Azure)**

PM2 Cluster Mode

```
pm2 start app.js -i max
```

-

Benefits:

✔ Horizontal scaling
✔ High availability
✔ Failover
✔ Better distribution of traffic

Q6. Explore advanced concepts in Node.js development, such as microservices architecture, asynchronous programming patterns, or serverless computing. How do these advanced topics influence the design and development of robust Node.js applications?

Answer:

1. Microservices Architecture

- Breaks application into smaller services
- Independent deployment
- Better scalability
- Requires API gateway, service discovery, and distributed tracing

Influence:
✓ More modular design
✓ Fault isolation
✓ Easier scaling of specific services

---

2. Asynchronous Programming Patterns

- Promises, async/await
- Streams
- Message queues (RabbitMQ, Kafka)

Influence:
✓ Non-blocking design
✓ Better performance
✓ Requires careful error handling

---

3. Serverless Computing

- Use AWS Lambda, Google Cloud Functions

- Auto-scaling
- Pay per use
- No server maintenance

Influence:
✓ Event-driven architecture
✓ Stateless functions
✓ Requires breaking application logic into small units

Q7. Design and develop a real-time multi-user chat application using Socket.io in Node.js. The application should support features such as broadcasting messages to all connected clients, private messaging between users, and event handling for user connections and disconnections.

Answer:

Server Code (Node.js + Socket.io)

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server);

const users = new Map();

io.on('connection', (socket) => {
  console.log("User connected:", socket.id);

  socket.on('register', (username) => {
    users.set(socket.id, username);
    io.emit('userJoined', username);
  });

  socket.on('publicMessage', (msg) => {
    io.emit('publicMessage', {
      from: users.get(socket.id),
      text: msg
    });
  });

  socket.on('privateMessage', ({ to, text }) => {
    const targetSocket = [...users].find(([id, name]) => name === to)?.[0];
    if (targetSocket) {
      io.to(targetSocket).emit("privateMessage", {
        from: users.get(socket.id),
        text
      });
    }
  });

  socket.on('disconnect', () => {
    const name = users.get(socket.id);
    users.delete(socket.id);
    io.emit('userLeft', name);
  });
});

server.listen(3000, () => console.log("Server running"));
```

Features Included:

✔ Broadcasting messages
✔ Private messaging
✔ User join/leave events

Q8. Explain the importance of unit testing in Node.js applications. Demonstrate how to set up a testing environment using a framework like Mocha or Jest. Develop a suite of unit tests for a sample Node.js module, covering various test cases, including edge cases and error scenarios.

Answer:

Importance of Unit Testing

- Ensures code correctness
- Prevents future bugs

- Helps refactor safely
- Improves design
- Automates quality checks

Setting up Jest Testing Environment

Install Jest:

```
npm install --save-dev jest
```

1.

Add to package.json:

```
"scripts": { "test": "jest" }
```

2.

Sample Module: math.js

```
function add(a, b) {
  if (typeof a !== "number" || typeof b !== "number")
    throw new Error("Invalid numbers");
  return a + b;
}

function divide(a, b) {
  if (b === 0) throw new Error("Divide by zero");
  return a / b;
}

module.exports = { add, divide };
```

Unit Tests (math.test.js)

```
const { add, divide } = require('./math');
```

```
test("add(): valid numbers", () => {
  expect(add(2, 3)).toBe(5);
});

test("add(): invalid input", () => {
  expect(() => add(2, "x")).toThrow("Invalid numbers");
});

test("divide(): normal case", () => {
  expect(divide(6, 3)).toBe(2);
});

test("divide(): divide by zero", () => {
  expect(() => divide(4, 0)).toThrow("Divide by zero");
});
```

Run tests:

```
npm test
```