💡 These test your ability to build, maintain, and secure server-side applications.

- What is the difference between **monolithic and microservices** architecture?
- How do you handle **authentication and authorization** in a web application? (Explain **JWT, OAuth, and Session-based auth**)
- Explain the **Node.js event loop** and how it handles asynchronous operations.
- How do you manage **state in a distributed system**?
- What are **ORMs (Object-Relational Mappers)**, and have you used one (e.g., Sequelize, TypeORM, Prisma)?
- Explain **SOLID principles** in software development.
- What is **middleware** in Express.js, and how do you use it?
- How do you handle **background jobs** in a Node.js application? (e.g., using BullMQ or Celery)
- How do you **secure an API** against attacks such as **SQL injection, CSRF, and DDoS**?

1. What is the difference between monolithic and microservices architecture?

Feature	Monolithic Architecture	Microservices Architecture
Structure	Single, unified codebase	Multiple small, independent services
Scalability	Harder to scale; scales as a whole	Easily scalable; scales individual services
Development	Simpler for small projects	Requires more infrastructure and management
Deployment	Entire application deployed at once	Independent services can be deployed separately
Communication	Direct function calls within the app	Uses APIs (e.g., REST, gRPC, GraphQL)
Fault Tolerance	Failure in one part can break the whole system	Failure in one service does not affect others

🔹 **Use Monolithic** for small applications, quick development, and easier management.

🔹 **Use Microservices** for large-scale applications needing scalability, fault tolerance, and independent deployments.

---

2. How do you handle authentication and authorization in a web application?

Authentication verifies identity; authorization controls access.

Common Authentication Methods:

١. **JWT (JSON Web Tokens)**

- Stateless (no need to store sessions on the server).
- Encodes user data inside a signed token.
- Ideal for APIs, mobile apps, and single-page applications.

Example:

javascript
CopyEdit

```javascript
const token = jwt.sign({ userId: user.id }, "secretKey", { expiresIn: "1h" });
```

٢. **OAuth 2.0 & OpenID Connect**

- Used for third-party logins (Google, Facebook, GitHub).
- **OAuth** provides authorization, **OpenID Connect (OIDC)** extends it to authentication.
- Uses access tokens & refresh tokens.

٣. **Session-Based Authentication**

- Stores session ID in a cookie; server keeps session data.
- Requires session management (e.g., Redis for scaling).

Example using Express-session:

javascript
CopyEdit

```javascript
app.use(session({ secret: "secretKey", resave: false, saveUninitialized: true }));
```

◆ **Use JWT** for scalable APIs.
◆ **Use OAuth** for third-party logins.
◆ **Use Sessions** for traditional web applications.

---

3. Explain the Node.js event loop and how it handles asynchronous operations.

Node.js uses a **single-threaded event loop** with **non-blocking I/O** to handle concurrent tasks efficiently.

١. **Phases of the Event Loop:**
   - **Timers:** Executes setTimeout and setInterval.
   - **I/O callbacks:** Handles I/O operations (network, file system).
   - **Idle & Prepare:** Internal tasks.
   - **Poll:** Retrieves new I/O events.
   - **Check:** Executes setImmediate().
   - **Close Callbacks:** Handles socket.on("close").

٢. **Handling Asynchronous Tasks:**
   - **Callbacks:** Traditional approach, but causes callback hell.
   - **Promises:** Provides chaining with .then().
   - **Async/Await:** Cleaner, avoids callback hell.

   Example:

   javascript
   CopyEdit

```
async function fetchData() {
  const response = await fetch("https://api.example.com/data");
  const data = await response.json();
  console.log(data);
}
fetchData();
```

---

4. How do you manage state in a distributed system?

State management in distributed systems ensures **consistency, availability, and fault tolerance**.

Approaches:

١. **Database Replication:**
   - Synchronizes data across multiple databases (e.g., PostgreSQL replication).

٢. **Cache Systems:**
   - **Redis** or **Memcached** for fast access to frequently used data.

٣. **Event-Driven Architecture:**
   - Uses message brokers (e.g., **Kafka, RabbitMQ**) to manage state asynchronously.

٤. **Service Coordination (Consensus Algorithms):**
    - Uses **Zookeeper, etcd, or Consul** for distributed consensus.

◆ **Use caching** for fast data retrieval.
◆ **Use event-driven architecture** to handle distributed state changes efficiently.

---

5. What are ORMs (Object-Relational Mappers)? Have you used one?

ORMs map database tables to objects in programming languages.

Popular ORMs in Node.js:

- **Sequelize (for SQL databases like PostgreSQL, MySQL):**

  javascript
  CopyEdit

  ```
  const User = sequelize.define("User", { name: Sequelize.STRING });
  ```

- **Prisma (Next-gen ORM, type-safe, for SQL databases):**

  javascript
  CopyEdit

  ```
  const users = await prisma.user.findMany();
  ```

- **TypeORM (for TypeScript, supports Active Record & Data Mapper patterns):**

  typescript
  CopyEdit

  ```
  @Entity() class User { @PrimaryGeneratedColumn() id: number; }
  ```

◆ **Use ORMs** to simplify database interactions, but raw queries may be better for performance-critical operations.

---

6. Explain SOLID principles in software development.

SOLID principles improve software maintainability and scalability.

١. **S**ingle Responsibility Principle (SRP) → One class should have one responsibility.
٢. **O**pen/Closed Principle (OCP) → Open for extension, closed for modification.
٣. **L**iskov Substitution Principle (LSP) → Derived classes should be substitutable for base classes.
٤. **I**nterface Segregation Principle (ISP) → No large interfaces; instead, create multiple small ones.
٥. **D**ependency Inversion Principle (DIP) → Depend on abstractions, not concrete implementations.

🔹 Applying SOLID ensures clean, scalable, and testable code.

---

7. What is middleware in Express.js, and how do you use it?

Middleware functions in Express.js execute **before the final request handler**.

Types of Middleware:

١. **Application-Level Middleware:**

javascript
CopyEdit

```
app.use((req, res, next) => {
  console.log("Request received");
  next();
});
```

٢. **Router-Level Middleware:**

javascript
CopyEdit

```
const router = express.Router();
router.use(authMiddleware);
```

٣. **Error-Handling Middleware:**

javascript
CopyEdit

```
app.use((err, req, res, next) => {
  res.status(500).send("Something went wrong!");
});
```

◆ **Middleware** helps with **logging, authentication, validation, and error handling**.

---

8. How do you handle background jobs in a Node.js application?

Background jobs allow time-consuming tasks to run asynchronously.

Tools for Background Jobs:

١. **BullMQ (Redis-based job queue)**:

```
javascript
CopyEdit
const Queue = require("bull");
const myQueue = new Queue("emailQueue");
myQueue.add({ email: "user@example.com" });
```

٢. **Agenda (MongoDB-based job scheduling)**:

```
javascript
CopyEdit
const agenda = new Agenda({ db: { address: "mongodb://localhost/jobs" } });
```

٣. **Node-cron (for scheduled tasks like cron jobs)**:

```
javascript
CopyEdit
cron.schedule("*/5 * * * *", () => console.log("Runs every 5 minutes"));
```

◆ **Use BullMQ for scalable, Redis-based task queues.**

---

9. How do you secure an API against attacks like SQL injection, CSRF, and DDoS?

Attack	Prevention
SQL Injection	Use **parameterized queries** (? in SQL, ORM escape methods)

Attack	Prevention
CSRF (Cross-Site Request Forgery)	Use **CSRF tokens** (`csrf` package), **SameSite cookies**
DDoS (Distributed Denial of Service)	Use **rate limiting** (`express-rate-limit`), **CDN (Cloudflare, AWS WAF)**
XSS (Cross-Site Scripting)	Escape user input (`DOMPurify`), enable **CSP (Content Security Policy)**

🔹 Security is **critical**—always **sanitize inputs, use HTTPS, and implement proper authentication mechanisms**.