Day 4

Example 1 — Employee Portal JWT Authentication

Scenario

We have an **Employee API**:

- Employees must **login** to get a **JWT access token**.
- Token is required to access **protected routes** (`/employees`).
- Token expires after 15 minutes.

Project Structure

```
employee-jwt-api/
│── app.js
│── routes/
│     └── employee.js
│     └── auth.js
│── middleware/
│     └── authMiddleware.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express jsonwebtoken bcryptjs
```

Step 2 — Authentication Routes

routes/auth.js

```
const express = require("express");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const router = express.Router();
const users = [
  { id: 1, username: "alice", password:
bcrypt.hashSync("password123", 8) },
];

const ACCESS_TOKEN_SECRET = "access_secret";
const REFRESH_TOKEN_SECRET = "refresh_secret";
let refreshTokens = [];

// Login
router.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username);
  if (!user || !bcrypt.compareSync(password, user.password))
{
    return res.status(401).json({ error: "Invalid
credentials" });
  }

  const accessToken = jwt.sign({ id: user.id },
ACCESS_TOKEN_SECRET, { expiresIn: "15m" });
  const refreshToken = jwt.sign({ id: user.id },
REFRESH_TOKEN_SECRET);

  refreshTokens.push(refreshToken);
  res.json({ accessToken, refreshToken });
});

// Refresh token
router.post("/refresh", (req, res) => {
  const { token } = req.body;
  if (!token || !refreshTokens.includes(token)) return
res.status(403).json({ error: "Invalid refresh token" });

  jwt.verify(token, REFRESH_TOKEN_SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Token
verification failed" });
    const newAccessToken = jwt.sign({ id: user.id },
ACCESS_TOKEN_SECRET, { expiresIn: "15m" });
    res.json({ accessToken: newAccessToken });
  });
});

module.exports = router;
```

Step 3 — Protected Route Middleware

middleware/authMiddleware.js

```
const jwt = require("jsonwebtoken");
const ACCESS_TOKEN_SECRET = "access_secret";

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1];
  if (!token) return res.status(401).json({ error: "No token 
provided" });

  jwt.verify(token, ACCESS_TOKEN_SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Invalid 
token" });
    req.user = user;
    next();
  });
};

module.exports = authenticateToken;
```

Step 4 — Employee Routes

routes/employee.js

```
const express = require("express");
const router = express.Router();
const authenticateToken = require("../middleware/
authMiddleware");

let employees = [
  { id: 1, name: "Alice", department: "HR" },
  { id: 2, name: "Bob", department: "IT" },
];

router.get("/", authenticateToken, (req, res) => {
  res.json(employees);
});

module.exports = router;
```

Step 5 — App Setup

app.js

```
const express = require("express");
const authRoutes = require("./routes/auth");
const employeeRoutes = require("./routes/employee");

const app = express();
app.use(express.json());

app.use("/auth", authRoutes);
app.use("/employees", employeeRoutes);

app.listen(3000, () => console.log("Server running on port 3000"));
```

Explanation

1. `/auth/login` generates **access & refresh tokens**.
2. `/auth/refresh` issues a **new access token** using refresh token.
3. `authenticateToken` middleware protects `/employees`.
4. Passwords are hashed with **bcrypt**.

Sample Output

Login (POST /auth/login)

```
{
  "accessToken": "<JWT_ACCESS_TOKEN>",
  "refreshToken": "<JWT_REFRESH_TOKEN>"
}
```

Access protected route (GET /employees)

```
[
  { "id": 1, "name": "Alice", "department": "HR" },
  { "id": 2, "name": "Bob", "department": "IT" }
]
```

Refresh Token (POST /auth/refresh)

```
{
  "accessToken": "<NEW_JWT_ACCESS_TOKEN>"
}
```

Example 2 — Role-Based JWT Authentication with Admin & User

Scenario

- Two types of users: **admin** and **user**.
- Some routes are **admin-only** (`/admin`).
- JWT stores **role** in payload for **authorization**.

Project Structure

```
role-jwt-api/
│── app.js
│── routes/
│    └── auth.js
│    └── dashboard.js
│── middleware/
│    └── authMiddleware.js
│── package.json
```

Step 1 — Authentication and Role Middleware

routes/auth.js

```
const express = require("express");
const jwt = require("jsonwebtoken");
const router = express.Router();

const users = [
  { id: 1, username: "alice", password: "123", role:
"admin" },
  { id: 2, username: "bob", password: "123", role: "user" }
];
```

```
const SECRET = "jwt_secret";

router.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username &&
u.password === password);
  if (!user) return res.status(401).json({ error: "Invalid
credentials" });

  const token = jwt.sign({ id: user.id, role: user.role },
SECRET, { expiresIn: "30m" });
  res.json({ token });
});

module.exports = router;
```

middleware/authMiddleware.js

```
const jwt = require("jsonwebtoken");
const SECRET = "jwt_secret";

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1];
  if (!token) return res.status(401).json({ error: "No token
provided" });

  jwt.verify(token, SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Invalid
token" });
    req.user = user;
    next();
  });
};

const authorizeRole = (role) => (req, res, next) => {
  if (req.user.role !== role) return
res.status(403).json({ error: "Access denied" });
  next();
};

module.exports = { authenticateToken, authorizeRole };
```

Step 2 — Dashboard Routes

routes/dashboard.js

```
const express = require("express");
const { authenticateToken, authorizeRole } = require("../
middleware/authMiddleware");
const router = express.Router();

router.get("/admin", authenticateToken,
authorizeRole("admin"), (req, res) => {
  res.json({ message: "Welcome Admin!" });
});

router.get("/user", authenticateToken, (req, res) => {
  res.json({ message: `Welcome ${req.user.role}!` });
});

module.exports = router;
```

Step 3 — App Setup

app.js

```
const express = require("express");
const authRoutes = require("./routes/auth");
const dashboardRoutes = require("./routes/dashboard");

const app = express();
app.use(express.json());

app.use("/auth", authRoutes);
app.use("/dashboard", dashboardRoutes);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Explanation

1. JWT stores **user role** in payload.
2. `authorizeRole` middleware restricts access to certain routes.
3. `authenticateToken` ensures the user is logged in.

Sample Output

Login (POST /auth/login)

```
{
  "token": "<JWT_TOKEN_WITH_ROLE>"
}
```

Admin Route (GET /dashboard/admin)

- With Admin Token → 200 OK

```
{ "message": "Welcome Admin!" }
```

- With User Token → 403 Forbidden

```
{ "error": "Access denied" }
```

User Route (GET /dashboard/user)

- Accessible to both roles

```
{ "message": "Welcome user!" }
```

These two examples demonstrate **medium-level JWT authentication**, covering:

1. **Access & refresh tokens**
2. **Token verification**
3. **Role-based authorization**

Example 1 — Multi-Tenant API with JWT Authentication

Scenario

- We have a **multi-tenant SaaS API** where users belong to different companies (`tenantId`).
- JWT includes **tenantId** in the payload.
- Routes are protected and only accessible if the **tenantId in JWT matches the requested data**.

Project Structure

```
multi-tenant-api/
│── app.js
│── routes/
│     └── projects.js
│     └── auth.js
│── middleware/
│     └── authMiddleware.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express jsonwebtoken bcryptjs
```

Step 2 — Authentication Route

routes/auth.js

```
const express = require("express");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const router = express.Router();

const users = [
  { id: 1, username: "alice", password:
bcrypt.hashSync("pass123", 8), tenantId: 101 },
  { id: 2, username: "bob", password:
bcrypt.hashSync("pass123", 8), tenantId: 102 }
];

const SECRET = "multi_tenant_secret";

router.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username);
  if (!user || !bcrypt.compareSync(password, user.password))
    return res.status(401).json({ error: "Invalid
credentials" });
```

```
  const token = jwt.sign({ id: user.id, tenantId:
user.tenantId }, SECRET, { expiresIn: "30m" });
  res.json({ token });
});

module.exports = router;
```

Step 3 — Middleware for Authentication & Tenant Verification

middleware/authMiddleware.js

```
const jwt = require("jsonwebtoken");
const SECRET = "multi_tenant_secret";

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1];
  if (!token) return res.status(401).json({ error: "No token
provided" });

  jwt.verify(token, SECRET, (err, payload) => {
    if (err) return res.status(403).json({ error: "Invalid
token" });
    req.user = payload;
    next();
  });
};

const authorizeTenant = (req, res, next) => {
  const requestedTenantId = parseInt(req.params.tenantId);
  if (req.user.tenantId !== requestedTenantId)
    return res.status(403).json({ error: "Access denied for
this tenant" });
  next();
};

module.exports = { authenticateToken, authorizeTenant };
```

Step 4 — Project Routes

routes/projects.js

```
const express = require("express");
const { authenticateToken, authorizeTenant } = require("../
middleware/authMiddleware");
const router = express.Router();

const projects = [
  { id: 1, name: "Project A", tenantId: 101 },
  { id: 2, name: "Project B", tenantId: 102 }
];

router.get("/:tenantId", authenticateToken, authorizeTenant,
(req, res) => {
  const tenantProjects = projects.filter(p => p.tenantId ===
parseInt(req.params.tenantId));
  res.json(tenantProjects);
});

module.exports = router;
```

Step 5 — App Setup

app.js

```
const express = require("express");
const authRoutes = require("./routes/auth");
const projectRoutes = require("./routes/projects");

const app = express();
app.use(express.json());

app.use("/auth", authRoutes);
app.use("/projects", projectRoutes);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Sample Output

Login POST /auth/login

```
{
  "token": "<JWT_TOKEN_WITH_TENANT>"
}
```

Access Tenant Projects GET /projects/101

- With valid token tenantId=101 → 200 OK

```
[ { "id": 1, "name": "Project A", "tenantId": 101 } ]
```

- With token tenantId=102 → 403 Forbidden

```
{ "error": "Access denied for this tenant" }
```

Example 2 — JWT with Expired Token Handling and Refresh Strategy

Scenario

- Users log in to get **short-lived access tokens** (5 min) and **long-lived refresh tokens** (1 hour).
- When access token expires, they can **refresh** using the refresh token.
- Protects sensitive routes like `/salary`.

Project Structure

```
jwt-refresh-api/
│── app.js
│── routes/
│     └── auth.js
│     └── salary.js
│── middleware/
│     └── authMiddleware.js
│── package.json
```

Step 1 — Authentication Routes

routes/auth.js

```
const express = require("express");
const jwt = require("jsonwebtoken");
const router = express.Router();
const users = [
  { id: 1, username: "alice", password: "123" }
];

const ACCESS_SECRET = "access_secret";
const REFRESH_SECRET = "refresh_secret";
let refreshTokens = [];

router.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username &&
u.password === password);
  if (!user) return res.status(401).json({ error: "Invalid
credentials" });

  const accessToken = jwt.sign({ id: user.id },
ACCESS_SECRET, { expiresIn: "5m" });
  const refreshToken = jwt.sign({ id: user.id },
REFRESH_SECRET, { expiresIn: "1h" });

  refreshTokens.push(refreshToken);
  res.json({ accessToken, refreshToken });
});

router.post("/refresh", (req, res) => {
  const { token } = req.body;
  if (!token || !refreshTokens.includes(token)) return
res.status(403).json({ error: "Invalid refresh token" });

  jwt.verify(token, REFRESH_SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Refresh
token expired" });
    const newAccessToken = jwt.sign({ id: user.id },
ACCESS_SECRET, { expiresIn: "5m" });
    res.json({ accessToken: newAccessToken });
  });
});

module.exports = router;
```

Step 2 — Middleware for Protected Routes

middleware/authMiddleware.js

```
const jwt = require("jsonwebtoken");
const ACCESS_SECRET = "access_secret";

const authenticateToken = (req, res, next) => {
  const token = req.headers["authorization"]?.split(" ")[1];
  if (!token) return res.status(401).json({ error: "No token
provided" });

  jwt.verify(token, ACCESS_SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Access
token expired or invalid" });
    req.user = user;
    next();
  });
};

module.exports = authenticateToken;
```

Step 3 — Salary Route

routes/salary.js

```
const express = require("express");
const authenticateToken = require("../middleware/
authMiddleware");
const router = express.Router();

const salaries = [
  { id: 1, employee: "Alice", amount: 50000 },
  { id: 2, employee: "Bob", amount: 60000 }
];

router.get("/", authenticateToken, (req, res) => {
  res.json(salaries);
});

module.exports = router;
```

Step 4 — App Setup

app.js

```
const express = require("express");
const authRoutes = require("./routes/auth");
const salaryRoutes = require("./routes/salary");

const app = express();
app.use(express.json());

app.use("/auth", authRoutes);
app.use("/salary", salaryRoutes);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Sample Output

Login (POST /auth/login)

```
{
  "accessToken": "<ACCESS_TOKEN>",
  "refreshToken": "<REFRESH_TOKEN>"
}
```

Access protected route GET /salary

- Valid token → 200 OK

```
[
  { "id": 1, "employee": "Alice", "amount": 50000 },
  { "id": 2, "employee": "Bob", "amount": 60000 }
]
```

Access expired token → 403 Forbidden

```
{ "error": "Access token expired or invalid" }
```

Refresh token POST /auth/refresh

```
{ "accessToken": "<NEW_ACCESS_TOKEN>" }
```

These two examples focus on **different scenarios** from previous ones:

1. **Multi-tenant authorization using tenantId in JWT**
2. **Short-lived access tokens with refresh token strategy for sensitive routes**

Example 1 — RBAC with Admin/User Permissions

Scenario

- A company API has two roles: **admin** and **user**.
- Admins can **create, update, and delete employees**.
- Users can **view only**.
- JWT stores the **role**, and middleware enforces access control.

Project Structure

```
rbac-api/
│── app.js
│── routes/
│     └── auth.js
│     └── employees.js
│── middleware/
│     └── authMiddleware.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express jsonwebtoken bcryptjs
```

Step 2 — Authentication Routes

routes/auth.js

```
const express = require("express");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const router = express.Router();

const users = [
  { id: 1, username: "admin", password:
bcrypt.hashSync("admin123", 8), role: "admin" },
  { id: 2, username: "bob", password:
bcrypt.hashSync("user123", 8), role: "user" }
];

const SECRET = "rbac_secret";

router.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username);
  if (!user || !bcrypt.compareSync(password, user.password))
{
    return res.status(401).json({ error: "Invalid
credentials" });
  }

  const token = jwt.sign({ id: user.id, role: user.role },
SECRET, { expiresIn: "1h" });
  res.json({ token });
});

module.exports = router;
```

Step 3 — RBAC Middleware

middleware/authMiddleware.js

```
const jwt = require("jsonwebtoken");
const SECRET = "rbac_secret";

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1];
  if (!token) return res.status(401).json({ error: "No token
provided" });

  jwt.verify(token, SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Invalid
token" });
    req.user = user;
    next();
  });
};
```

```
const authorizeRole = (roles) => (req, res, next) => {
  if (!roles.includes(req.user.role)) return
res.status(403).json({ error: "Access denied" });
  next();
};

module.exports = { authenticateToken, authorizeRole };
```

Step 4 — Employee Routes

routes/employees.js

```
const express = require("express");
const { authenticateToken, authorizeRole } = require("../
middleware/authMiddleware");
const router = express.Router();

let employees = [
  { id: 1, name: "Alice", department: "HR" },
  { id: 2, name: "Bob", department: "IT" }
];

// GET all employees - user/admin
router.get("/", authenticateToken, (req, res) => {
  res.json(employees);
});

// POST employee - admin only
router.post("/", authenticateToken, authorizeRole(["admin"]),
(req, res) => {
  const emp = { id: employees.length + 1, ...req.body };
  employees.push(emp);
  res.json(emp);
});

// DELETE employee - admin only
router.delete("/:id", authenticateToken,
authorizeRole(["admin"]), (req, res) => {
  employees = employees.filter(e => e.id != req.params.id);
  res.json({ message: "Employee deleted" });
});

module.exports = router;
```

Step 5 — App Setup

app.js

```
const express = require("express");
const authRoutes = require("./routes/auth");
const employeeRoutes = require("./routes/employees");

const app = express();
app.use(express.json());

app.use("/auth", authRoutes);
app.use("/employees", employeeRoutes);

app.listen(3000, () => console.log("Server running on port 
3000"));
```

Sample Output

Login as Admin

```
{ "token": "<JWT_ADMIN_TOKEN>" }
```

POST /employees (Admin)

```
{ "id": 3, "name": "Charlie", "department": "Finance" }
```

POST /employees (User)

```
{ "error": "Access denied" }
```

GET /employees (User/Admin)

```
[
  { "id": 1, "name": "Alice", "department": "HR" },
  { "id": 2, "name": "Bob", "department": "IT" },
  { "id": 3, "name": "Charlie", "department": "Finance" }
]
```

Example 2 — Secure Password Storage, Rate Limiting, and CSRF/XSS Protection

Scenario

- A **user authentication system** with:
    1. **Bcrypt** for password hashing
    2. **Rate limiting** to prevent brute force login
    3. **Helmet** for basic CSRF/XSS/security headers

Project Structure

```
secure-auth-api/
│── app.js
│── routes/
│     └── auth.js
│── middleware/
│     └── rateLimiter.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express bcryptjs jsonwebtoken express-rate-limit
helmet
```

Step 2 — Rate Limiter Middleware

middleware/rateLimiter.js

```
const rateLimit = require("express-rate-limit");

const loginLimiter = rateLimit({
  windowMs: 10 * 60 * 1000, // 10 minutes
  max: 5, // max 5 attempts
  message: "Too many login attempts. Try again later."
});

module.exports = loginLimiter;
```

Step 3 — Authentication Routes

routes/auth.js

```
const express = require("express");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const router = express.Router();

const users = []; // store hashed passwords

const SECRET = "secure_secret";

// Register
router.post("/register", (req, res) => {
  const { username, password } = req.body;
  const hashed = bcrypt.hashSync(password, 10);
  users.push({ username, password: hashed });
  res.json({ message: "User registered" });
});

// Login
router.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username);
  if (!user || !bcrypt.compareSync(password, user.password))
{
    return res.status(401).json({ error: "Invalid
credentials" });
  }
  const token = jwt.sign({ username }, SECRET, { expiresIn:
"1h" });
  res.json({ token });
});

module.exports = router;
```

Step 4 — App Setup with Helmet and Rate Limiting

app.js

```
const express = require("express");
const helmet = require("helmet");
const authRoutes = require("./routes/auth");
const loginLimiter = require("./middleware/rateLimiter");
const app = express();
app.use(express.json());
app.use(helmet()); // security headers

app.use("/auth/login", loginLimiter);
app.use("/auth", authRoutes);

app.listen(3000, () => console.log("Server running on port
3000"));
```

Explanation

1. **Bcrypt** hashes passwords; plain-text passwords are never stored.
2. **Rate limiting** prevents brute force login attacks.
3. **Helmet** adds secure headers to prevent CSRF/XSS.

Sample Output

Register POST /auth/register

```
{ "message": "User registered" }
```

Login POST /auth/login

```
{ "token": "<JWT_TOKEN>" }
```

After 5 failed login attempts → Rate Limiter

```
{ "message": "Too many login attempts. Try again later." }
```

Headers added by Helmet

```
X-DNS-Prefetch-Control: off
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
...
```

Example 1 — Unit Testing with Jest & Integration Testing with Supertest

Scenario

- We have an **Employee API** with routes:
    1. `GET /employees` → list employees
    2. `POST /employees` → add a new employee
- We want to **unit test utility functions** and **integration test API endpoints**.

Project Structure

```
employee-api/
│── app.js
│── routes/
│     └── employees.js
│── utils/
│     └── salary.js
│── tests/
│     ├── salary.test.js
│     └── employees.test.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install express supertest jest
```

Add in `package.json`:

```
"scripts": {
  "start": "node app.js",
  "test": "jest"
}
```

Step 2 — Employee Routes

routes/employees.js

```
const express = require("express");
const router = express.Router();

let employees = [
  { id: 1, name: "Alice", salary: 50000 },
  { id: 2, name: "Bob", salary: 60000 }
];

// GET employees
router.get("/", (req, res) => res.json(employees));

// POST employee
router.post("/", (req, res) => {
  const emp = { id: employees.length + 1, ...req.body };
  employees.push(emp);
  res.json(emp);
});

module.exports = router;
```

Step 3 — Utility Function for Unit Testing

utils/salary.js

```
function calculateBonus(salary, percentage) {
  if (percentage < 0) throw new Error("Invalid bonus
percentage");
  return salary + salary * (percentage / 100);
}

module.exports = { calculateBonus };
```

Step 4 — App Setup

app.js

```
const express = require("express");
const employeeRoutes = require("./routes/employees");

const app = express();
app.use(express.json());

app.use("/employees", employeeRoutes);

module.exports = app; // export for supertest
```

Step 5 — Unit Test with Jest

tests/salary.test.js

```
const { calculateBonus } = require("../utils/salary");

describe("calculateBonus", () => {
  test("should calculate 10% bonus correctly", () => {
    expect(calculateBonus(1000, 10)).toBe(1100);
  });

  test("should throw error for negative percentage", () => {
    expect(() => calculateBonus(1000, -5)).toThrow("Invalid
bonus percentage");
  });
});
```

Step 6 — Integration Test with Supertest

tests/employees.test.js

```
const request = require("supertest");
const app = require("../app");

describe("Employee API", () => {
  test("GET /employees should return all employees", async ()
=> {
    const res = await request(app).get("/employees");
    expect(res.statusCode).toBe(200);
    expect(res.body.length).toBeGreaterThan(0);
  });

  test("POST /employees should add a new employee", async ()
=> {
    const res = await request(app)
      .post("/employees")
      .send({ name: "Charlie", salary: 70000 });
    expect(res.statusCode).toBe(200);
    expect(res.body.name).toBe("Charlie");
  });
});
```

Explanation

1. `salary.test.js` → **unit tests** utility logic (calculateBonus).

2. `employees.test.js` → **integration tests** endpoints using **supertest**.
3. `app.js` exports app for **test harness**.

Sample Output

```
$ npm test

PASS  tests/salary.test.js
 PASS  tests/employees.test.js

Test Suites: 2 passed
Tests:       4 passed
```

Example 2 — Mocking Dependencies & Test Coverage with Mocha/Chai/Sinon

Scenario

- We have a **User service** that calls a **database** to fetch users.
- We want to **unit test** the service **without hitting a real DB** using **mocking/stubbing**.
- Also, measure **test coverage**.

Project Structure

```
user-service/
│── services/
│     └── userService.js
│── tests/
│     └── userService.test.js
│── package.json
```

Step 1 — Install Dependencies

```
npm install mocha chai sinon nyc
```

Add scripts in `package.json`:

```
"scripts": {
  "test": "mocha",
  "coverage": "nyc mocha"
}
```

Step 2 — User Service

services/userService.js

```
// Simulated DB module
const db = {
  getUsers: () => [{ id: 1, name: "Alice" }, { id: 2, name:
"Bob" }]
};

async function fetchUsers() {
  const users = await db.getUsers();
  return users.map(u => ({ ...u, active: true }));
}

module.exports = { fetchUsers, db };
```

Step 3 — Unit Test with Sinon Stub

tests/userService.test.js

```
const { expect } = require("chai");
const sinon = require("sinon");
const { fetchUsers, db } = require("../services/
userService");

describe("User Service", () => {
  it("should return users with active flag", async () => {
    const stub = sinon.stub(db, "getUsers").resolves([{ id:
1, name: "TestUser" }]);
    const users = await fetchUsers();
    expect(users[0].active).to.be.true;
    expect(users[0].name).to.equal("TestUser");
    stub.restore();
  });
});
```

Explanation

1. `db.getUsers` is **stubbed** to avoid real database calls.
2. `fetchUsers` logic is tested independently.
3. Use `nyc` for **coverage report**:

```
npm run coverage
```

Sample Coverage Output

```
--------------------|---------|----------|---------|---------
|--------------------
File                | % Stmts | % Branch | % Funcs | % Lines
| Uncovered Line #s
--------------------|---------|----------|---------|---------
|--------------------
All files           |   100%  |   100%    |  100%   |  100%
|
 services/userService.js | 100%  |   100%    |  100%    |  100%
|
--------------------|---------|----------|---------|---------
|--------------------
```

✅ **Key Concepts Covered**

- **Example 1:**
  - Unit testing utility functions (Jest)
  - Integration testing API endpoints (Supertest)
- **Example 2:**
  - Mocking/stubbing dependencies (Sinon)
  - Test coverage reporting (nyc)
  - Independent unit testing without DB