✅ Unit Testing in Software Development

📌 What is Unit Testing?

- **Definition**: Unit Testing is a software testing method where individual components or functions of a program are tested in isolation.
- **Goal**: To validate that each unit of the software performs as expected.
- **"Unit"**: Usually refers to a single function, method, or class.

🎯 Why Unit Testing?

- **Early Bug Detection**: Catches bugs during development, reducing downstream issues.
- **Simplifies Integration**: Ensures each module works correctly before integration.
- **Improves Code Quality**: Encourages modular, maintainable, and testable code.
- **Documentation**: Acts as living documentation of how the code is supposed to behave.
- **Refactoring Confidence**: Allows safe code refactoring by verifying nothing breaks.

🧰 Unit Testing Frameworks

These provide tools to write, run, and report tests.

🔹 JavaScript (Node.js)

- **Mocha**: Feature-rich test framework running on Node.js and in the browser.
- **Jest**: All-in-one testing framework by Facebook (includes assertions, mocks).

- **Jasmine**: BDD-focused testing framework, standalone.
- **Chai**: Assertion library (often used with Mocha).

🔄 What are Mocks and Stubs?

Term	Description	Purpose
Stub	Dummy object that returns predefined responses	Used to isolate code by simulating dependencies
Mock	Object that records which functions were called, with what arguments	Used to verify behavior or interaction

- **Use Case**: Both are used when testing components with dependencies like databases, APIs, or external services.

🧪 What is Jest?

Jest is a popular testing framework maintained by Facebook. It supports:

- Zero config testing
- Fast performance
- Built-in mocking
- Code coverage

📦 Setting Up Jest in a Node.js Project

1. Install Jest

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

2. Update `package.json`

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

🗒️ Example Project Structure

```
my-app/
├── sum.js
├── sum.test.js
└── package.json
```

✅ Sample Unit Test in Jest

`sum.js`

```
function sum(a, b) {
  return a + b;
}

module.exports = sum;
```

sum.test.js

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

▸ **Run the test**

```
npm test
```

🧠 Key Jest Concepts

Concept	Description
`test()` / `it()`	Defines a test case
`expect()`	Creates an assertion
Matchers like `.toBe()`	Used to assert the expected result
`.toEqual()`	For deep object/array comparison
`.toThrow()`	To test if a function throws an error

🛠️ Real-World Example: Testing a User Login Function

auth.js

```
function login(username, password) {
  if (username === 'admin' && password === '1234') {
    return 'Login success';
  } else {
    throw new Error('Invalid credentials');
  }
}
```

```
module.exports = login;
```

auth.test.js

```
const login = require('./auth');

test('successful login', () => {
  expect(login('admin', '1234')).toBe('Login success');
});

test('login fails with wrong password', () => {
  expect(() => login('admin', 'wrong')).toThrow('Invalid credentials');
});
```

🧪 Tips for Writing Good Unit Tests

- Test **one unit** per test.
- Cover **edge cases** and **error conditions**.
- Use **descriptive test names**.
- Keep test data **minimal and relevant**.
- Mock external dependencies (like DB, APIs) if needed.

🛠️ Project Setup

1. Install Required Packages

```
npm install express
npm install --save-dev jest supertest
```

📁 Folder Structure

```
my-app/
├── app.js
├── app.test.js
└── package.json
```

📝 `app.js` — Express App with GET and POST

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

const users = [];

// GET endpoint
app.get('/users', (req, res) => {
  res.status(200).json(users);
});

// POST endpoint
app.post('/users', (req, res) => {
  const user = req.body;
  users.push(user);
  res.status(201).json(user);
});

module.exports = app;
```

🧪 `app.test.js` — Jest + Supertest Test Cases

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

describe('User API Endpoints', () => {
  it('GET /users should return empty array initially', async () => {
    const res = await request(app).get('/users');
```

```
    expect(res.statusCode).toBe(200);
    expect(res.body).toEqual([]);
  });

  it('POST /users should create a new user', async () => {
    const newUser = { id: 1, name: 'John' };
    const res = await request(app).post('/users').send(newUser);
    expect(res.statusCode).toBe(201);
    expect(res.body).toEqual(newUser);
  });

  it('GET /users should return array with newly added user', async () => {
    const res = await request(app).get('/users');
    expect(res.statusCode).toBe(200);
    expect(res.body.length).toBe(1);
    expect(res.body[0].name).toBe('John');
  });
});
```

🧪 Update `package.json` to Run Jest

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

▸ Run Tests

```
npm test
```

✅ This tests:

- A `GET` endpoint returns initial empty array

- A `POST` endpoint correctly adds a user
- `GET` again returns the updated user list

❓ Why Are We Using `supertest` in Jest Tests for Express APIs?

🔍 Problem:

When you create an **Express API**, you typically run the app on a server (e.g., `app.listen(port)`) and then **send HTTP requests** to test it. But in **unit testing**, you want to:

- **Avoid starting a real server**
- Still **send HTTP requests** to test routes like `GET /users`, `POST /users`

✅ Solution: Use `supertest`

`supertest` is a library that allows you to:

- Make **fake HTTP requests** to your Express app
- Test your routes **without starting the server**

🧪 What `supertest` Does

It wraps your Express `app` object and allows you to simulate real HTTP requests:

```
request(app).get('/users')        // Simulates a GET request
request(app).post('/users').send({ name: "John" }) // Simulates POST with
body
```

You don’t need to run:

```
app.listen(3000); // ❌ Not needed in tests
```

🧾 Code Example Recap

```
const request = require('supertest'); // 👈 This is Supertest
const app = require('./app');         // Your Express app

test('GET /users', async () => {
  const res = await request(app).get('/users'); // 👈 Fake HTTP request
  expect(res.statusCode).toBe(200);
});
```

✅ Advantages of Using Supertest

Feature	Benefit
No server needed	Tests run faster and more isolated
Works with Jest	Easily integrates with `async/await` tests
Supports all HTTP methods	GET, POST, PUT, DELETE, etc.

Can test headers, body, status	Full request/response testing

🧪 Summary

`supertest` allows you to **unit test Express routes** just like a real client would, but **without spinning up a real HTTP server** — making tests **fast**, **reliable**, and **easy to automate**.