# Application Testing

Testing is the process of evaluating software to find defects, ensure it works properly, and meet required standards, ensuring a reliable and smooth user experience.

Imagine you're developing an e-commerce app with a User Authentication module. A tester verifies that it works. Then, you add a Product Management module, and the tester only tests that new module. When the app is released, users discover that the User Authentication module no longer works, even though it was fine before. This happened because the tester didn’t recheck the User Authentication module after the Product Management module was added.

This highlights a problem with manual testing: every time a new feature is added, the tester must check everything to make sure nothing else breaks. This is time-consuming and resource-heavy, especially in large applications.

This is where automated testing and test cases written by developers help. Automated tests can quickly check all modules after each change, saving time, reducing errors, and catching bugs early.

As a developer, knowing that automated tests are in place gives you confidence to make changes or add new features without the constant worry of breaking existing functionality. It allows you to move faster while ensuring the stability of the application.

# Automation Testing

Automation testing uses software tools to automatically run tests on an application, instead of doing it manually. It saves time, reduces errors, and makes testing faster, especially for repetitive tasks. Test cases (pre-written test scenarios) are used to check if the app works as expected. In a CI/CD (Continuous Integration/Continuous Deployment) pipeline, automated tests run every time new code is added. If any test case fails, it stops the deployment, ensuring that broken code doesn’t get released.

### example

```
// greet.js
 export function greet(name) {
   return `Hello, ${name}!`;
      }

// greet.test.js
 import { test, describe } from 'node:test';
 import assert from 'node:assert';
 import { greet } from './greet.js';
 describe('Greeting function', () => {
 test('greet returns the correct greeting', () => {
 const expected = 'Hello, Alice!';
 const actual = greet('Alice');
 assert.strictEqual(actual, expected);
     });
         });
```

This test setup uses Node.js's built-in test runner (`node:test`) to verify that the `greet` function works as expected. The `greet` function, defined in `greet.js`, takes a name and returns a greeting like `"Hello, Alice!"`. In the test file `greet.test.js`, we import `test` and `describe` from `node:test` to structure our test, and use `assert` from `node:assert` to check if the actual output matches the expected result. The test follows the AAA pattern:

1. Arrange – Prepare everything you need for the test, like input values and expected output.
   Line: const expected = 'Hello, Alice!';

2. Act – Run the function or action you're testing using the arranged inputs.
   Line: const actual = greet('Alice');

3. Assert – Check if the actual output matches the expected one. This confirms the function behaves correctly.

Line: assert.strictEqual(actual, expected);

# Types of Testing

## 1. Unit Testing

Unit testing is the process of testing individual functions or small pieces of code in isolation to make sure they work correctly. It runs very fast as compared to other types.

Example:

Testing a greet(name) function to ensure it returns "Hello, Alice!" when passed 'Alice'.

Goal:
Catch bugs early by verifying that each unit of code behaves as expected.

## 2. Integration Testing

Integration testing checks how different parts (modules) of your application work together. It ensures that components interact correctly. It is slower than unit tests

Example:

Testing that a userService can successfully create a user and store it in a database using databaseService.

Goal:
Verify that the interaction between units works correctly (e.g., API + DB, or frontend + backend service).

## 3. End-to-End (E2E) Testing

E2E testing simulates real user actions to test the entire system from start to finish — from frontend to backend to database. Takes the most time since it tests everything.

Example:

A test that opens the website, logs in a user, adds a product to the cart, and completes a purchase.

Goal:

Ensure the whole application works as expected in a real-world scenario.

## Tests must run in Isolation

To ensure tests are reliable and repeatable, they should run in isolation. This means each test should be independent and not rely on other tests or on external systems like databases, APIs, or files. Isolated tests are easier to debug and help avoid issues where one test's failure affects another.

## Mocking

To achieve isolation, we use mocking. Mocking means replacing real parts of the system—like a database, API, or file operation—with fake versions that behave in a controlled way. These mocks return predefined responses and don't actually perform the real operations.