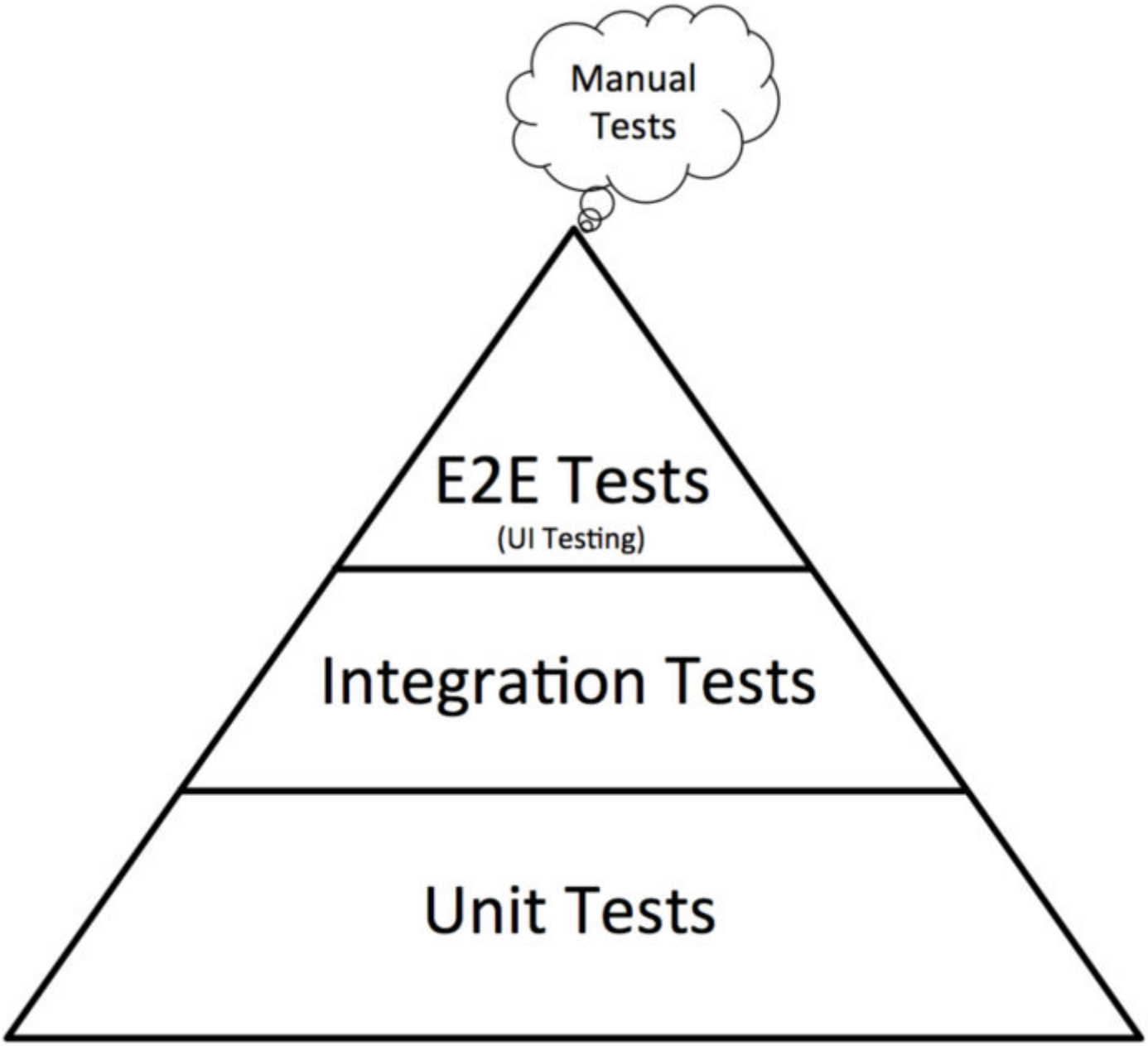
Software Testing

Testing Pyramid



Layer	Tests	Fast?	Reliable?	Cost	Purpose
Unit Tests	Test single functions/methods	Super fast	Very reliable	Cheap	Test internal logic
Integration Tests	Test interactions between modules (e.g., DB + API)	Fast-ish	Reliable	Medium	Check systems work together

Layer	Tests	Fast?	Reliable?	Cost	Purpose
End-to-End (E2E) Tests	Test whole user flows (e.g., login)	Slow	Often flaky	Expensive	Mimic real user behavior

Unit Testing

Unit testing is the process of testing the smallest testable parts of an application (called units, like functions, methods, or classes) in isolation to ensure they behave correctly.

- Focues on one thing at a time
- Doesnt test external systems (eg database,file system)
- Should be automated and repeatable

Core concepts:-

Term	What it does
Test Case	A single scenario that tests a specific behaviour
Assertion	Statement that checks if a value meets expectations
Test suite	A collection of test cases
Mocking	Replacing real dependencies with fake ones
Test Coverage	How much of your code is exercised by tests

Code Coverage

Code coverage is a metric that tells you what percentage of your code is executed when your tests run. It helps answer:

Term	What it Measures	Example
Line Coverage	Which lines were executed	`if (x > 0)` — was this line run?
Branch Coverage	Whether every branch (if/else) executed	Did both `if` **and** `else` paths run?
Function Coverage	Which functions/methods were called	Was `calculateTax()` called at all?
Statement Coverage	Similar to line coverage	
Condition Coverage	All boolean conditions tested	`if (a && b)` — were both `a` and `b` true/false?

- 70–90% is considered healthy
- Focus more on meaningful coverage rather than number-chasing
- **100% is not always realistic — some code (e.g. error logs, debug-only paths) isn't worth testing**

Modified Condition/Decision Coverage

In software development, MC/DC is a testing metric that ensures your conditional logic is thoroughly tested. It's used especially when:

- You work with complex if, while, or switch conditions
- You build software that must be highly reliable
- You want to ensure that each part of a decision is meaningful and tested

MC/DC helps you:

- Avoid logic bugs that slip past simple tests
- Catch missing test cases early
- Justify test coverage to stakeholders or auditors
- Make your code robust in the face of change

Example

```
function isTransactionAllowed(user, amount) {
  return user.isActive && !user.isFrozen && amount < user.limit;
}
```

Conditions:

- user.isActive
- !user.isFrozen
- amount < user.limit

To satisfy MC/DC:
Write test cases where each condition is flipped independently and causes the decision to change:

Active, not frozen, within limit →

Active, not frozen, over limit →

Inactive, not frozen, within limit →

Active, frozen, within limit →

Applying MC/DC in Development

When Writing Code:

- Break large expressions into smaller parts
- Avoid deep nesting of logic

When Writing Tests:

- Use truth tables
- Focus on minimal MC/DC test sets

Test Case	A (isActive)	B (!isFrozen)	C (amount < limit)	Result
TC1	false	true	true	false
TC2	true	true	true	true
TC3	true	false	true	false
TC4	true	true	false	false

When to Use MC/DC

Use MC/DC when...	Don't worry about MC/DC when...
Complex or safety-critical decision logic	Simple CRUD or prototype apps
High-stakes business or compliance code	Non-critical tooling
Refactoring complex boolean logic	Iterating quickly on internal apps
Formal QA or coverage requirement	Small scripts or UI tweaks

Integration Testing

Integration Testing verifies that multiple components/modules of a system work together as expected.

What Does It Test?

You're testing the interaction between two or more units, such as:

- API route ↔ Controller ↔ Service ↔ Database
- Microservice A ↔ Microservice B
- Backend ↔ Cache (Redis) ↔ Queue
- Module A ↔ External API

Key Goals

- Detect interface mismatches
- Ensure data flow and transformations work properly
- Catch real-world issues like DB failures, unexpected responses, timeouts

What to Test in Integration Tests?

Scenario	Example
DB interaction	Does `POST /signup` store data in the database?
Auth flows	Does login return a JWT if credentials are valid?
Service connections	Does the backend store uploaded file to S3 correctly?
Inter-service calls	Does Service A call Service B and handle the response?
State transitions	Does the cart update correctly after checkout?

Patterns of Integration Testing

1. Big Bang Integration
    - Combine all components and test everything together
    - Bad for isolating errors
2. Top-Down Integration
    - Start with high-level modules, mock lower ones (use stubs)
3. Bottom-Up Integration
    - Start with low-level modules (e.g., DB), mock top layers (drivers)
4. Sandwich (Hybrid)

- Combine both approaches; test middle layer first

Real-Life Example

Flow:

```
Request → Express Route → Controller → Service → MongoDB
```

Test Code:

```
// user.test.js
const request = require('supertest');
const app = require('../app'); // Express app
const mongoose = require('mongoose');

beforeAll(async () => {
  await mongoose.connect('mongodb://localhost:27017/testdb');
});

afterAll(async () => {
  await mongoose.connection.dropDatabase();
  await mongoose.connection.close();
});

describe('POST /users', () => {
  it('should create a user', async () => {
    const res = await request(app)
      .post('/users')
      .send({ name: 'John', email: 'john@example.com' });

    expect(res.statusCode).toBe(201);
    expect(res.body.user.email).toBe('john@example.com');
  });
});
```

Test Data Management

Integration tests touch real DBs. You must:

- Seed test data before tests
- Rollback or clean DB after tests

- Use Dockerized DBs or tools like Testcontainers
- Optionally use in-memory DBs like SQLite or Redis Memory

Dealing with External Services

Options:

1. Use mocks/fakes – simulate APIs, queues, etc.
2. Use test instances – e.g., test S3 bucket
3. Use Docker/Testcontainers – spin up actual Redis, DB, etc.

When to Write Integration Tests?

Do write when:

- Your app calls another module, DB, or service
- Data is being passed/converted across boundaries
- A bug could come from incorrect connections

Don't write when:

- You're just testing a function (sum(a, b))
- You've already tested it fully in E2E
- You don't control the external system and can't mock it

Black-box vs White-box Testing

These are two fundamental software testing methodologies based on what the tester knows about the internal system.

Black-box Testing:

Black-box Testing tests a system from the outside — without any knowledge of its internal logic or code.

- The tester doesn't care how the code works, only what it does.

Key Features: Black-box Testing

Feature	Black-box Testing
Access to source code	No
Who does it?	QA testers, end users
What is tested?	Inputs and outputs (behavior)
Speed	Slower (more exploratory/manual)
Reliability	High (covers real usage)

What Can You Test?

Area	Example
Functional correctness	Does `login()` return a token on success?
UI behavior	Does clicking "Submit" create a post?
API endpoints	Does `POST /users` create a new user?
Error handling	Does app show error on bad input?
Edge cases	What if user enters no password?

Techniques Used:

Technique	What it means
Equivalence partitioning	Test input groups (valid vs invalid ranges)
Boundary value analysis	Test edges like 0, 1, `MAX_INT`
Decision table testing	Test all combinations of inputs/decisions
State transition testing	Test states: logged-in → logged-out
Error guessing	Try likely failure scenarios (bad input)

White-box Testing

White-box Testing (aka clear box, glass box, structural testing) tests the internal logic, code, and structure of the system.

- The tester knows everything about the code, including flow, logic, and structure.

Key Features:

Feature	White-box Testing
Access to source code	Yes
Who does it?	Developers or test engineers
What is tested?	Logic branches, paths, loops
Speed	Very fast (automated usually)
Reliability	Great for internal bugs

What Can You Test?

Test Target	Example
Logic branches	Does `if-else` behave as expected?
Conditions	Is `if (a && b)` tested for all values?
Loops	Are edge cases in loops covered?
Function coverage	Is every function tested?
Internal errors	Are exceptions handled correctly?

Techniques Used:

Technique	What it covers
Statement coverage	Has every line run at least once?
Branch coverage	Has each `if/else` branch run?
Path coverage	Have all possible execution paths run?
Condition coverage	Have all boolean conditions been tested?
MC/DC	Every condition independently affects output (used in aviation, etc.)

Black-box vs White-box Testing: Comparison

Feature	Black-box	White-box
Knowledge of code	Not required	Required
Tested by	QA, testers	Developers, SDETs
Scope	External behavior	Internal structure

Feature	Black-box	White-box
Tools	Selenium, Postman	Jest, JUnit, Pytest
Speed	Slower	Faster
Bugs found	UI, API, logic flow	Code logic, paths

When to Use Which?

Use Case	Recommended Testing
Testing public APIs or UI	Black-box
Testing a login route behavior	Black-box + Integration
Testing backend auth logic	White-box
Validating loops, paths, edge logic	White-box
Full user flow (e.g., signup → email)	Black-box

Summary

Aspect	Black-box	White-box
Focus	What the app does	How the app works
View	Outside-in	Inside-out
Ideal for	User behavior, APIs	Logic, code correctness
Who writes it	QA testers	Developers
Test type examples	UI test, API test	Unit test, path coverage

End-to-End (E2E) Testing

E2E Testing simulates how a real user interacts with your application from start to finish.
It verifies that all integrated pieces of the system (frontend, backend, DB, APIs, auth, etc.) work together correctly.

- E2E = Entire user journey is tested. Think: "From login ➡ checkout ➡ order confirmation"

What Does E2E Test

Area	Example

Area	Example
Authentication	Can a user register and then log in?
UI interaction	Does clicking a button open a modal?
Navigation	Can the user go from homepage ➞ cart ➞ checkout?
Full workflows	Can user add items to cart, checkout, and get a confirmation message?
Error handling	Does an invalid email show the correct error message?
3rd-party flows	Can the user complete a Stripe or Google OAuth login?

E2E vs Unit vs Integration

Feature	Unit Test	Integration Test	E2E Test
Scope	One function/module	Multiple modules	Full system
Speed	Super fast	Medium	Slow
Setup	Minimal	DB or mocks	Full app + DB + browser
Example	`add(2, 3)` returns 5	API connects to DB	User logs in and sees dashboard
Breakage cause	Logic bug	Miscommunication between modules	UI/API/network issue

Pros:

Benefit	Why it matters
Real user simulation	Tests what really matters
Prevents regression bugs	Catch bugs caused by UI/backend updates
Cross-system coverage	Tests frontend, backend, DB, auth, etc.
Business flow validation	Ensures real workflows (e.g., checkout) work

Cons

Drawback	Description
Slow	UI, network, and DB make it slower
Flaky	Minor delays or race conditions can fail test

Drawback	Description
Costly	Setup, maintenance, CI execution time
Debugging is harder	Failures can be vague (“button not found”)

Best Practices

Tip	Description
Test critical user flows only	Don’t test every button — focus on core workflows
Use data resets	Clean DB or use mock/stubbed data before each test
Isolate from real systems	Use test environments, test DBs, and avoid production APIs
Wait intelligently	Use explicit waits (`cy.wait()` , `await`) instead of fixed delays
Run on CI/CD	Integrate with GitHub Actions, GitLab CI, Jenkins, etc.
Screenshot/video on fail	Helps debugging in CI (Cypress/Playwright do this well)

Example

```
describe('User Signup Flow', () => {
  it('should sign up and redirect to dashboard', () => {
    cy.visit('/signup');
    cy.get('input[name="email"]').type('hyxal@example.com');
    cy.get('input[name="password"]').type('securepass123');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
    cy.contains('Welcome, Hyxal');
  });
});
```