Web Science Application Development

RPI

October 8, 2025

Next Parts of Class

Core Engineering Pillars

- **Authentication & Authorization**
- **Testing** (Unit & Integration)
- **Deployment** (Architecture & CI/CD Pipelines)
- **Agentic Systems** (LangChain & Tooling)

In Parallel, We Will:

- Deepen our understanding of **React fundamentals and patterns**
- Compare and evaluate **modern web frameworks** (e.g., Flask)
- Connect architectural decisions to real-world tradeoffs

Readings - Testing

- [Web Application Testing for QA Teams: A Step-by-Step Guide](https://www.virtuosoqa.com/post/testing-web-applications)
- [Jest Documentation](https://jestjs.io/docs/getting-started)
- [Supertest Documentation](https://github.com/ladjs/supertest)
- [Playwright Testing Documentation](https://playwright.dev/docs/intro)
- [Testing Pyramid (Martin Fowler)](https://martinfowler.com/articles/practical-test-pyramid.html)
- [Google Testing Blog](https://testing.googleblog.com/)

How do we know our code actually works?

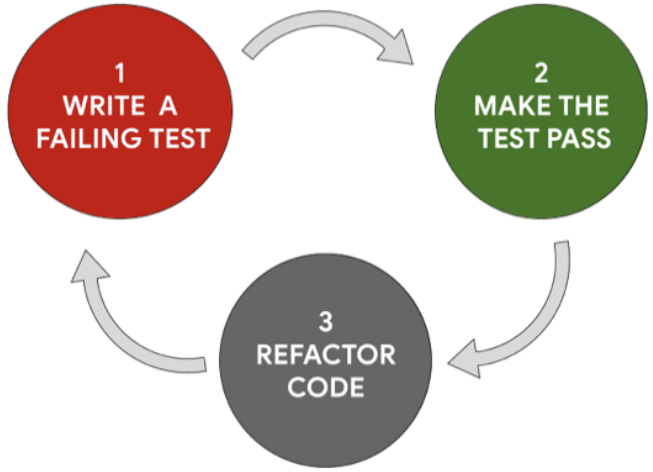
Why Testing Matters

- Ensures reliability
- Prevents regressions
- Improves code quality
- Enables faster, reliable development
- Without testing, can’t make changes without breaking things

Test-Driven Development

Test-Driven Development (TDD) is the practice of working in a structured cycle where writing tests comes before writing production code. The process involves three steps, sometimes called the *red-green-refactor cycle*:

1. **Write a failing test**
2. **Make the test pass by writing just enough production code**
3. **Refactor the production code to meet your quality standards**



Particularly Important in the age of AI

Testing Functionality and Deployments

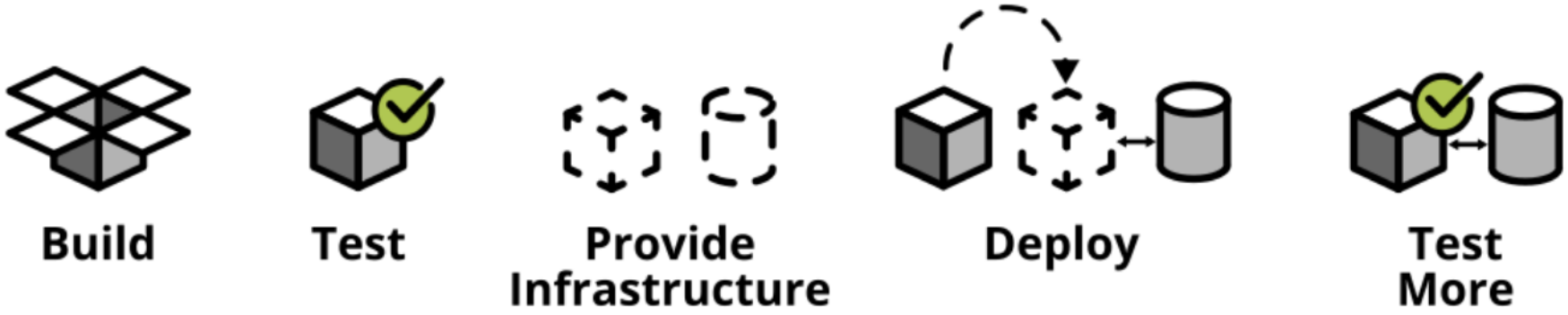


Figure 1: Use build pipelines to automatically and reliably get your software into production

Source: https://martinfowler.com/articles/practical-test-pyramid.html

Types of Tests

- **Unit Tests**
  - Test a single function in isolation
  - Example: does bcrypt.hash() produce a valid hash?
  - Fast — run hundreds in seconds
- **Integration Tests**
  - Test components working together
  - Example: does POST /api/register actually store a user?
  - Slower — may need a running server or database
- **End-to-End (E2E) Tests**
  - Test the full user flow from client to server
  - Example: register → login → access protected route
  - Slowest — but catches real-world bugs

The Testing Pyramid

If you want to get serious about automated tests for your software there is one key concept you should know about: the **test pyramid**. Mike Cohn came up with this concept in his book *Succeeding with Agile*. It's a great visual metaphor telling you tc think about different layers of testing. It also tells you how much testing to do on e layer.

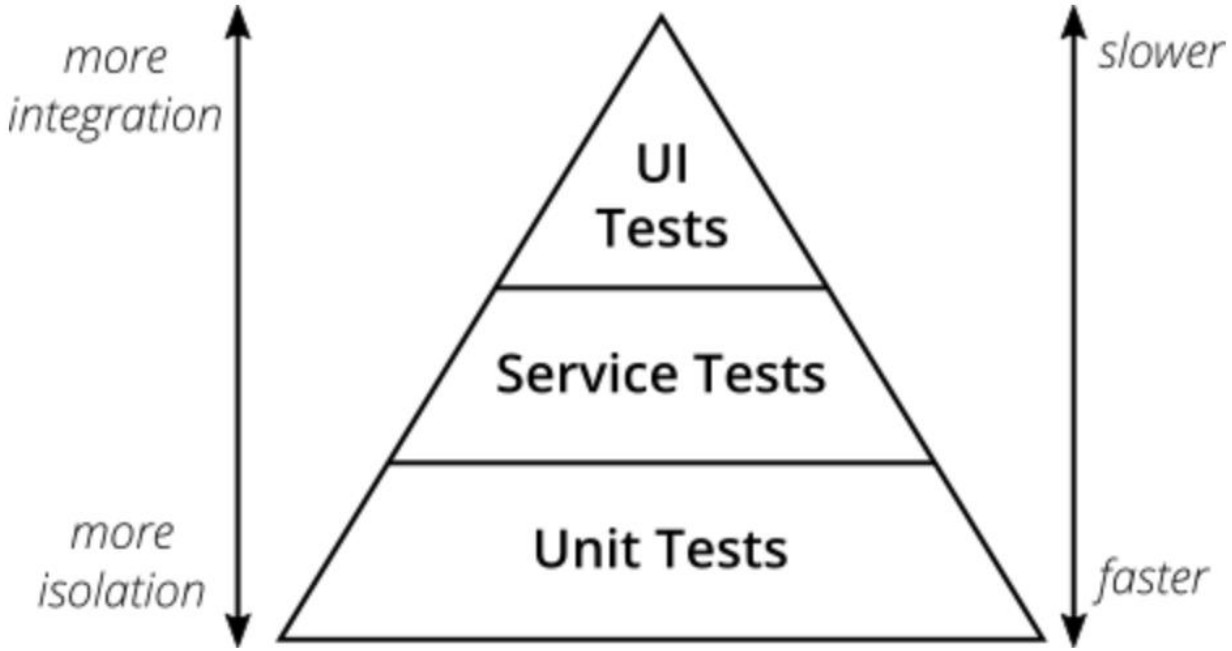


Figure 2: The Test Pyramid

Testing in the Demo Repo

- **27 tests across 3 layers, all in the same repo**
  - 12 unit tests (Jest) - test pure utility functions
  - 10 integration tests (Supertest) - test API endpoints
  - 5 E2E tests (Playwright) - test full browser flows
- **What the tests cover:**
  - Input validation (valid, invalid, edge cases, null)
  - Registration, login, and auth token flow
  - Protected routes (no token, valid token, invalid token)
  - Full UI: register, login, access secret, logout
- **Key idea: tests document expected behavior**
  - If the tests pass, the API contract is met

Automated Testing & CI/CD

- **CI = Continuous Integration**
  - Every push triggers automated tests
  - Catch bugs before they reach production
  - GitHub Actions runs our autograder on every push
- **CD = Continuous Deployment**
  - If tests pass, deploy automatically
  - If tests fail, block the deployment
- **Why this matters:**
  - Manual testing doesn't scale
  - Automated tests give confidence to ship fast
  - Tests are a safety net — not a replacement for good design

Testing Frameworks Overview

- Demo of Unit, Integration, and End-to-End testing
- Node.js / Express JWT App

Testing Layers

- • Unit: test functions
- • Integration: test APIs
- • E2E: test full user flow

Unit Testing (Jest)

- **Fast (ms) - Tests isolated functions with no dependencies**

- **Example from utils.test.js:**

```
test('rejects short password', () => {
  const result = validateCredentials('alice', '123');
  expect(result.valid).toBe(false);
});
```

  - No server, no database, no network - just function in, result out
  - Covers: valid inputs, invalid inputs, edge cases, null handling

Integration Testing (Supertest)

- **Medium speed - Tests API routes and middleware together**

- **Example from api.test.js (using Supertest):**

```
test('registers a new user', async () => {
  const res = await request(app)
    .post('/api/register')
    .send({ username: 'alice', password: 'pass123' });
  expect(res.status).toBe(201);
});
```

  - Supertest makes HTTP requests without starting a real server
  - Verifies status codes, response bodies, auth flow

End-to-End Testing (Playwright)

- **Slower (seconds) - Tests full app in a real Chromium browser**

- **Example from auth.spec.js (using Playwright):**

```
test('can register and login', async ({ page }) => {
  await page.locator('input[type="text"]')
    .fill('alice');
  await page.locator('button').click();
  await expect(page.locator('.success'))
    .toBeVisible();
});
```

  - Fills forms, clicks buttons, asserts on visible page content

Project Structure

```
server.js              # Express app with JWT auth
utils.js               # Pure functions (unit test targets)
public/index.html      # React frontend
playwright.config.js   # E2E test config
tests/
  unit/utils.test.js         # Jest unit tests
  integration/api.test.js    # Supertest API tests
  e2e/auth.spec.js           # Playwright browser tests
```

Running Tests

- **`npm test`** Unit + Integration (Jest)
- **`npm run test:unit`** Unit tests only
- **`npm run test:integration`** API tests only
- **`npm run test:e2e`** Browser tests (Playwright)
- **`npm run test:all`** Everything

Try It - GitHub Codespaces

- **Open the demo repo in Codespaces:**
  - Go to the repo on GitHub
  - Click the green Code button, then Codespaces tab
  - Click “Create codespace on main”

- **Everything is pre-configured:**
  - Node.js 20 + npm dependencies auto-installed
  - Playwright browsers auto-installed
  - VS Code extensions for Jest and Playwright

- **Run all tests immediately:**

What Test Output Looks Like

- ## Jest (Unit + Integration):

```
PASS tests/unit/utils.test.js
  validateCredentials
    ✓ accepts valid username and password (2 ms)
    ✓ rejects missing username (1 ms)
    ✓ rejects short password (< 6 characters)
Tests: 22 passed, 22 total | Time: 0.98s
```

- ## Playwright (E2E):

```
✓ page displays the title and login status (868ms)
✓ can register, login, and access the secret (1.3s)
✓ logout clears the session (1.1s)
5 passed (5.8s)
```

Exercise: Write Your Own Tests (ungraded)

- Add one new test to each layer. See the README for hints and solutions.

- **1. Unit Test**
  - Test validateCredentials with a whitespace-only username
  - Does the current code handle this? If not, fix the bug!

- **2. Integration Test**
  - Test that /api/refresh returns a new access token after login

- **3. E2E Test**

Key Takeaways

- **Use all 3 testing layers**
  - Unit tests catch logic bugs fast; integration tests verify APIs; E2E tests catch real user-facing bugs
- **Balance speed vs realism**
  - More unit tests (fast), fewer E2E tests (slow but realistic)
- **Automate testing in your workflow**
  - CI/CD runs tests on every push - catch bugs before production
- **Tests document expected behavior**
  - If the tests pass, the API contract is met