Testing, Performance & Real-time

Web Technologies — Lecture 12

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

The Three Things We Add Today

- **Testing** — proves your code works *and keeps working*
- **Performance** — ensures it works *fast enough*
- **Real-time** — ensures users see updates *the moment they happen*

Every production-grade web app needs all three. The capstone is graded on them.

The Pyramid

- **Unit tests** — many, fast, test one function
- **Integration tests** — some, medium speed, test components together (e.g., API + DB)
- **End-to-end (E2E) tests** — few, slow, drive the real browser

Rule of thumb

80% unit, 15% integration, 5% E2E. The pyramid points down: fewer tests as they get more expensive.

Setup — Vitest

```
npm install -D vitest
```

```
// package.json
"scripts": { "test": "vitest" }
```

Vitest is the modern Vite-native runner; Jest is the older equivalent. Same API.

Your First Unit Test

```
// math.js
export function add(a, b) { return a + b; }
```

```
// math.test.js
import { describe, it, expect } from 'vitest';
import { add } from './math.js';

describe('add', () => {
  it('adds two positives', () => {
    expect(add(2, 3)).toBe(5);
  });

  it('handles negatives', () => {
    expect(add(-1, 1)).toBe(0);
  });
});
```

Arrange / Act / Assert

```
it('removes a todo', () => {
  // Arrange
  const list = [{ id: 1, text: 'a' }, { id: 2, text: 'b' }];

  // Act
  const result = removeTodo(list, 1);

  // Assert
  expect(result).toEqual([{ id: 2, text: 'b' }]);
});
```

One assertion idea per test. Tests should fail *loudly* and pin-point the bug.

Mocking

```
import { vi } from 'vitest';

it('calls the API on submit', async () => {
  const fakeFetch = vi.fn().mockResolvedValue({
    ok: true,
    json: () => Promise.resolve({ id: 42 })
  });
  global.fetch = fakeFetch;

  await submit({ name: 'Amina' });

  expect(fakeFetch).toHaveBeenCalledWith('/api/students', expect.any(Object));
});
```

Mock external calls (network, DB, time) to keep tests fast and deterministic.

React Testing Library

```
npm install -D @testing-library/react @testing-library/jest-dom jsdom
```

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import Counter from './Counter';

it('increments on click', async () => {
  render(<Counter />);
  expect(screen.getByText('Count: 0')).toBeInTheDocument();

  await userEvent.click(screen.getByRole('button', { name: /increment/i }));

  expect(screen.getByText('Count: 1')).toBeInTheDocument();
});
```

Test **behaviour**, not implementation. Find elements the way users do.

Spring Boot — @SpringBootTest

```
@SpringBootTest(webEnvironment = RANDOM_PORT)
class StudentApiTest {

    @Autowired TestRestTemplate http;

    @Test
    void createsAndReadsStudent() {
        var s = new Student("Amina", "a@suza.ac.tz", "BCS");
        var created = http.postForObject("/api/students", s, Student.class);
        assertNotNull(created.getId());

        var fetched = http.getForObject(
            "/api/students/" + created.getId(), Student.class);
        assertEquals("Amina", fetched.getName());
    }
}
```

Express + Supertest

```
import request from 'supertest';
import { app } from './app.js';

it('POST /api/students creates a row', async () => {
  const res = await request(app)
    .post('/api/students')
    .send({ name: 'Amina', email: 'a@suza.ac.tz' });

  expect(res.status).toBe(201);
  expect(res.body.id).toBeDefined();
});
```

Playwright in 60 Seconds

```
npm init playwright@latest
```

```
import { test, expect } from '@playwright/test';

test('user can sign up and see dashboard', async ({ page }) => {
  await page.goto('/signup');
  await page.fill('[name=email]', 'amina@suza.ac.tz');
  await page.fill('[name=password]', 'CorrectHorse42');
  await page.click('button[type=submit]');

  await expect(page).toHaveURL('/dashboard');
  await expect(page.locator('h1')).toContainText('Welcome');
});
```

Runs in real Chromium / Firefox / WebKit. Records video on failure.

Core Web Vitals (Google's metrics)

Metric	What it measures	Good
LCP (Largest Contentful Paint)	time to render the main content	< 2.5 s
INP (Interaction to Next Paint)	responsiveness to clicks/taps	< 200 ms
CLS (Cumulative Layout Shift)	visual stability (no jumps)	< 0.1

These are

SEO factors — Google ranks faster sites higher.

Lighthouse & Audits

- Run in Chrome devtools $\rightarrow$ Lighthouse tab
- Scores: Performance, Accessibility, Best Practices, SEO (0–100)
- Tells you *exactly* what to fix and by how much it'd improve
- Aim for $\geq$ **90** on each category

Frontend Performance Levers

- **Compress images**: WebP/AVIF, sized for the device
- **Lazy-load** images and offscreen iframes: `<img loading="lazy">`
- **Code-split** JS by route (Vite/React do this for you)
- **Minify + gzip/brotli** CSS, JS, HTML
- Preload critical fonts; `font-display: swap`
- Use a **CDN** for static assets
- Avoid layout-shifting: set `width`/`height` on images

Backend Performance Levers

- **Database indexes** on filter/join columns (Lecture 9)
- Avoid the **N+1** query problem (eager loading)
- **HTTP cache headers**: `Cache-Control: max-age=86400`
- **Cache hot reads in Redis**; bust on write
- **Pagination** for large lists — never return 10 000 rows
- **Async / queue** long-running work (email, image resize)

```
GET /api/students -> Cache-Control: public, max-age=60
```

Caching Layers — Mental Model

Browser cache → CDN → Reverse-proxy cache (nginx) → Application cache (Redis) → Database

- Each layer is faster than the next
- Hit rate matters more than raw speed
- Cache invalidation = the second-hardest problem in CS

HTTP vs WebSockets

HTTP	**WebSocket**
request/response	full-duplex stream
client initiates	either side can send anytime
new connection per request	one long-lived connection
Best for: APIs, pages	Best for: chat, dashboards, games

URL scheme: `ws://` (or `wss://` over TLS).

WebSocket — Browser Side

```
const ws = new WebSocket('wss://api.example.com/feed');

ws.addEventListener('open',    () => console.log('connected'));
ws.addEventListener('message', (e) => {
  const msg = JSON.parse(e.data);
  appendChat(msg);
});
ws.addEventListener('close',   () => reconnectWithBackoff());

document.querySelector('#send').addEventListener('click', () => {
  ws.send(JSON.stringify({ text: input.value }));
});
```

WebSocket — Node Server

```
import { WebSocketServer } from 'ws';
const wss = new WebSocketServer({ port: 8080 });
const clients = new Set();

wss.on('connection', (ws) => {
  clients.add(ws);
  ws.on('message', (data) => {
    // broadcast to all
    for (const c of clients) c.send(data);
  });
  ws.on('close', () => clients.delete(ws));
});
```

Server-Sent Events (SSE)

- One-way: server → browser only
- Plain HTTP, very simple to implement
- Auto-reconnect built into the browser
- Great for: live notifications, score boards, log tails

```
GET /events
Content-Type: text/event-stream

data: { "newOrder": 42 }

data: { "newOrder": 43 }
```

When to Use What

Need	Pick
Most pages and APIs	HTTP
One-way live feed	Server-Sent Events
Chat / multiplayer / live editing	WebSockets
Cross-platform mobile push	Firebase Cloud Messaging

Test What You Care About

- Test the **behaviour**, not the implementation
- Tests are **documentation**: a future you reads them to understand the code
- **Coverage is a tool, not a goal** — 100% coverage of trivial code = waste
- Always test the **boundary cases**: empty input, max length, network failure
- Run tests on every PR via GitHub Actions (Lecture 13)

Performance Mindset

- **Measure first.** Don't optimise on a hunch.
- Pick a goal: "LCP < 2.5 s on a mid-range Android over 4G"
- Optimise the **biggest cost** first — usually images, then JS bundle, then DB queries
- Re-measure after every change
- Premature optimisation is the root of all evil — but deferred optimisation is also a sin

Summary

- Testing pyramid: many unit, some integration, few E2E
- Vitest / RTL / Supertest / Playwright cover all layers
- Core Web Vitals (LCP, INP, CLS) are how Google judges your site
- Cache aggressively and invalidate carefully
- WebSockets / SSE for real-time; HTTP for everything else

Next: Deployment & DevOps — Docker, CI/CD, observability.

Questions?