

Software Testing Interview Questions and Answers

Intern to senior level

Software testing interview questions and answers for QA engineers and developers: unit vs integration vs end-to-end tests, the test pyramid, mocking, flaky tests and Selenium vs Cypress vs Playwright.

5 questions Read online: <https://www.fredevtemplates.com/interview-questions/testing>

1. What is the difference between unit, integration and end-to-end tests?

Software Testing Easy Intern, Junior

Unit tests check one small piece of code in isolation, integration tests check that several pieces work together (for example code plus a real database), and end-to-end tests drive the whole application the way a user would.

Unit tests are fast and precise: when one fails, you know exactly which function broke. They replace slow or external dependencies with test doubles, so they cannot prove the pieces fit together.

Integration tests exercise real collaborations, such as a repository class against a test database or an API handler with its middleware. They catch wiring, configuration and query mistakes that unit tests miss, at the cost of more setup and slower runs.

End-to-end tests run the deployed application through its real interface, usually a browser with Playwright, Cypress or Selenium. They give the most confidence that a user journey works, but they are the slowest, most expensive to maintain and most prone to flakiness.

In the interview

A unit test checks one function or class in isolation and runs in milliseconds. An integration test checks that real parts work together, like my API handler and a test database. An end-to-end test drives the whole app through the browser like a user. I rely mostly on unit and integration tests and keep a small set of end-to-end tests for the critical journeys.

Key points

- Unit: small, isolated, fast, precise failures
- Integration: real collaborators such as a database or HTTP layer
- End-to-end: full user journeys through the UI, slowest and most brittle

```
// unit: pure function, no dependencies
test('applies a 10% discount', () => {
  expect(applyDiscount(100, 0.1)).toBe(90);
});

// end-to-end: a real browser against the running app
test('user can check out', async ({ page }) => {
  await page.goto('/cart');
  await page.getByRole('button', { name: 'Checkout' }).click();
  await expect(page.getByText('Order confirmed')).toBeVisible();
});
```

2. What is the test pyramid?

Software Testing Easy Junior, Mid-level

The test pyramid is a guideline to have many fast unit tests at the base, fewer integration tests in the middle and only a small number of slow end-to-end tests at the top, so the suite stays fast and reliable.

The shape reflects cost and speed. Lower-level tests are cheap to write and run and give precise failures; higher-level tests give more realistic confidence but are slower and break for more reasons.

The anti-pattern is the "ice-cream cone": most coverage comes from manual testing and UI tests, with few unit tests. Such suites take hours, fail randomly, and teams stop trusting them.

The pyramid is a heuristic, not a law. Some teams prefer a "testing trophy" with more integration tests, especially for frontend code where testing components with their real children gives better confidence per test. The underlying principle is the same: push each check to the lowest level that can catch the bug.

Key points

- Many unit tests, fewer integration tests, few end-to-end tests
- Avoid the ice-cream cone of mostly UI and manual tests
- Test each behaviour at the lowest level that can catch the bug

3. What is mocking in testing and when should you use it?

Software Testing Medium Junior, Mid-level

Mocking replaces a real dependency with a controllable fake so a test can run in isolation and check how the code interacts with it; use it for slow, non-deterministic or external dependencies, not for everything.

Test doubles come in a few flavours. A stub returns canned answers, a mock also records calls so you can assert on them, a spy wraps a real function to observe it, and a fake is a working lightweight implementation such as an in-memory repository.

Mock at the boundaries of your system: payment providers, email services, the clock, random numbers, network calls. That keeps tests fast and deterministic and lets you simulate failures that are hard to trigger for real, such as a timeout.

Over-mocking is the common mistake. When a test mocks every collaborator and asserts on internal calls, it breaks on every refactor while still passing when the real integration is broken. Prefer asserting on outputs and observable behaviour, and cover real integrations with a smaller set of integration tests.

Key points

- Stub, mock, spy and fake are different kinds of test doubles
- Mock external, slow or non-deterministic dependencies
- Over-mocking couples tests to implementation details

```
test('sends a receipt after payment', async () => {
  const email = { send: jest.fn().mockResolvedValue(undefined) };
  const payments = { charge: jest.fn().mockResolvedValue({ id: 'pay_1' }) };

  await checkout({ email, payments }, { userId: 7, total: 50 });

  expect(payments.charge).toHaveBeenCalledWith(7, 50);
  expect(email.send).toHaveBeenCalledWith(7, expect.stringContaining('pay_1'));
});
```

4. How do you deal with flaky tests?

Software Testing Medium Mid-level, Senior

Treat a flaky test as a bug: quarantine it so it stops blocking the pipeline, reproduce it by running it repeatedly, fix the root cause (usually timing, shared state or external dependencies), then bring it back.

A flaky test passes and fails on the same code. Its real cost is trust: once developers learn to click "re-run", genuine failures get ignored too.

Common causes are fixed sleeps instead of waiting for a condition, tests that depend on execution order or shared data, real network calls, time zones and the current date, randomness without a fixed seed, and animations in UI tests.

The process: track flaky tests (many CI systems report them), quarantine them with an owner and a deadline, reproduce by running the test in a loop or under load, and fix the cause. Use auto-waiting assertions in Playwright or Cypress, isolate test data per test, control the clock, and mock external services. Retries can hide symptoms, so use them sparingly and still track the underlying failures.

Key points

- Flaky tests destroy trust in the suite
- Usual causes: timing, shared state, external services, dates, randomness
- Quarantine with an owner, reproduce in a loop, fix the root cause
- Avoid fixed sleeps; wait for conditions

```
// flaky: assumes the request finishes within 1 second
await page.click('text=Save');
await page.waitForTimeout(1000);
expect(await page.textContent('.toast')).toBe('Saved');

// stable: waits for the condition (auto-retrying assertion)
await page.getByRole('button', { name: 'Save' }).click();
await expect(page.getByRole('status')).toHaveText('Saved');
```

5. Selenium vs Cypress vs Playwright: which should you choose for test automation?

Software Testing Medium Mid-level, Senior

Selenium is the long-established, language-agnostic standard built on WebDriver; Cypress runs inside the browser with an excellent developer experience for JavaScript apps; Playwright offers fast, auto-waiting cross-browser automation with multiple tabs, contexts and languages. For a new web project, Playwright is usually the strongest default.

Selenium drives browsers through the WebDriver protocol and has bindings for Java, Python, C#, JavaScript and more. It has a huge ecosystem and grid support, which makes it common in large enterprises and teams that test in Java. It needs explicit waits and more setup to avoid flakiness.

Cypress runs in the same run loop as the application, which gives time-travel debugging and automatic waiting. It is JavaScript/TypeScript only and historically had limits with multiple tabs, cross-origin flows and browser coverage, although these have improved.

Playwright, from Microsoft, controls Chromium, Firefox and WebKit through their native protocols. It has auto-waiting locators, parallel runs, isolated browser contexts, network interception and trace viewing, with official support for JavaScript, Python, Java and .NET.

The right choice depends on the team: existing Selenium suites and Java skills, a JavaScript-only team that values Cypress's debugging, or a new project that wants Playwright's speed and browser coverage. Whatever tool, stable selectors (roles, test ids) and isolated test data matter more than the framework.

the Key points

- Selenium: WebDriver standard, many languages, large ecosystem
- Cypress: in-browser runner, great debugging, JavaScript only
- Playwright: fast, auto-waiting, Chromium/Firefox/WebKit, several languages
- Selectors and test data design matter more than the tool

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.freedevtemplates.com/interview-questions/testing>