

Testing interview questions

23 questions · 9 easy · 12 medium · 2 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

Why do we write automated tests?

Easy



#basics

Tests give you **confidence to change code**. Without them, every refactor or dependency upgrade is a gamble, and bugs reach users.

Main benefits:

- catch regressions early, when they are cheap to fix
- act as **living documentation** of how the code is meant to behave
- push you towards smaller, decoupled, testable designs
- make CI/CD and frequent deploys safe

Tests cost time to write and maintain, so the goal is not 100% of everything, but the tests that catch the bugs that matter.

2

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

Easy



#basics #strategy

- **Unit:** tests one small piece (a function, a hook, a component) in isolation, with its dependencies mocked. Very fast, precise failures, but cannot prove the pieces work together.
- **Integration:** tests several units working together, such as a component with its child components and a mocked network, or an API route hitting a real test database. Slower, but catches wiring bugs.
- **End-to-end (E2E):** drives the **real app through a real browser** like a user (Playwright, Cypress): sign up, add to cart, pay. Highest confidence, but slowest and most prone to flakiness.

A good suite mixes all three, weighted towards the cheap ones.

🔗 Likely follow-up: Which of these would you write for a checkout feature?

3

What is the testing pyramid, and what is the testing trophy?

Easy



#strategy

The **testing pyramid** says: many unit tests at the bottom, fewer integration tests, very few E2E tests at the top, because tests get slower, costlier and flakier as you go up.

The **testing trophy** (Kent C. Dodds) is a frontend-focused update:

- **static analysis** at the base (TypeScript, ESLint) catches typos and type errors for free
- a thin layer of unit tests
- the **biggest layer is integration tests**, because they give the most confidence per minute of effort
- a few E2E tests for critical flows

Both agree on the idea: balance confidence against speed and cost.

4

What is the Arrange-Act-Assert pattern?

Easy



#basics #patterns

A simple structure that keeps every test readable:

1. **Arrange**: set up the data, mocks and the thing under test
2. **Act**: do the one action you are testing
3. **Assert**: check the outcome

Rules that go with it:

- **one behaviour per test**, with a name that describes it ("returns 0 for an empty cart")
- no logic (loops, ifs) in tests
- each test independent of the others' order and state

BDD tools call the same thing Given-When-Then.

Example

```
function cartTotal(items) {
  return items.reduce((sum, i) => sum + i.price * i.qty, 0)
}
// it('sums price times quantity')
// Arrange
const items = [{ price: 10, qty: 2 }, { price: 5, qty: 1 }]
// Act
const total = cartTotal(items)
// Assert (expect(total).toBe(25))
console.log(total === 25 ? 'pass' : 'fail: ' + total)
```

5

Jest vs Vitest: what are the basics of writing a test?

Easy



#jest #vitest

Both are test runners with the same API: `describe` groups tests, `it` / `test` defines one, `expect` with **matchers** asserts (`toBe`, `toEqual`, `toHaveBeenCalledWith`, `toThrow`). Hooks like `beforeEach` and `afterEach` handle setup and teardown.

- `toBe` uses `Object.is` (same reference), `toEqual` compares structure
- `jest.fn()` / `vi.fn()` create mock functions

Vitest is the modern default for Vite and most new projects: it reuses the Vite config, supports ESM and TypeScript natively, and is very fast in watch mode. **Jest** is older and still widespread (React Native, many existing codebases). Migrating is mostly a find-and-replace of `jest` with `vi`.

Example

```
import { describe, expect, it } from 'vitest'
import { slugify } from './slugify'

describe('slugify', () => {
  it('lowercases and joins words with dashes', () => {
    expect(slugify('Hello World')).toBe('hello-world')
  })
  it('removes special characters', () => {
    expect(slugify('Rock & Roll!')).toBe('rock-roll')
  })
})
```

🔗 Likely follow-up: What is the difference between `toBe` and `toEqual`?

6

What is the difference between mocks, stubs, spies and fakes?

Medium 🔥🔥 #mocking

They are all **test doubles**, stand-ins for real dependencies:

- **Stub**: returns canned answers (`getUser` always returns Asha). You do not check how it was called.
- **Spy**: wraps or records calls to a function so you can assert on them (called once, with these arguments). `vi.spyOn` can keep the real implementation.
- **Mock**: a double with **expectations about how it is used**; in Jest/Vitest, `vi.fn()` is both a stub and a spy.
- **Fake**: a working lightweight implementation (an in-memory database, a fake clock).

Mock at the **boundaries** (network, time, randomness, payments), not your own internal modules, or tests break on every refactor while missing real bugs.

Example

```
// a hand-rolled vi.fn(): a stub that is also a spy
function fn(impl = () => undefined) {
  const spy = (...args) => { spy.calls.push(args); return impl(...args) }
  spy.calls = []
  return spy
}

const sendEmail = fn(() => ({ ok: true })) // stubbed result

function register(user, mailer) { return mailer(user.email, 'Welcome!') }

console.log(register({ email: 'a@b.com' }, sendEmail)) // { ok: true }
console.log(sendEmail.calls.length, sendEmail.calls[0]) // 1 [a@b.com, Welcome!]
```

🔗 Likely follow-up: What are the dangers of over-mocking?

7

What is the philosophy of React Testing Library?

Easy



#react testing library #react

"The more your tests resemble the way your software is used, the more confidence they can give you."

So RTL tests the component **like a user would**: render it, find elements by what users see (role, label, text), interact, and assert on the visible result. It deliberately does not give easy access to state, props or component instances.

Benefits:

- tests **survive refactors** (switching to `useReducer` or a store does not break them)
- they fail when the user experience actually breaks
- querying by role nudges you towards accessible markup

Avoid testing implementation details: internal state, function names, CSS classes, or how many times a component rendered.

8

Which React Testing Library queries should you use, and what is the difference between `getBy`, `queryBy` and `findBy`?

Medium



#react testing library #react

Priority order: `getByRole` (with `{ name }`) first, then `getByLabelText`, `getByPlaceholderText`, `getByText`, and `getByTestId` only as a **last resort**.

The prefix decides behaviour:

- `getBy...`: returns the element or **throws** if missing; use for things that should be there
- `queryBy...`: returns `null` if missing; use to assert something is **not** there
- `findBy...`: returns a **promise** that retries until the element appears (default 1s); use for async UI

Each has an `All` version for multiple matches. Use `screen` for queries and `@testing-library/user-event` for interactions, since it simulates full user behaviour (focus, key presses) rather than single events like `fireEvent`.

Example

```
it('shows an error for an invalid email', async () => {
  const user = userEvent.setup()
  render(<SignupForm />)

  await user.type(screen.getByLabelText(/email/i), 'not-an-email')
  await user.click(screen.getByRole('button', { name: /sign up/i }))

  expect(await screen.findByRole('alert')).toHaveTextContent(/valid
email/i)
  expect(screen.queryByText(/welcome/i)).not.toBeInTheDocument()
})
```

9

What would you test in a React component?

Medium



#react #strategy

Test **behaviour from the user's point of view**, focusing on what could realistically break:

- it renders the right content for given props (and empty, loading and error states)
- user interactions produce the right result: typing, clicking, submitting
- callbacks are called with the right data (`onSubmit` receives the form values)
- conditional UI: shown or hidden based on state or permissions
- accessibility basics: labels, roles, focus after actions
- data fetching with the network mocked (success and failure)

Skip: third-party library internals, styling details, and trivial components with no logic. A good question to ask: "what would a bug here look like to the user?"

10

How do you test asynchronous code?

Medium



#async

Make the test wait for the async work, otherwise it finishes before the assertion runs and passes falsely.

- `async` test functions with `await`: `await expect(fetchUser(1)).resolves.toEqual(user)` or `.rejects.toThrow()`
- in UI tests, `await screen.findBy...` or `await waitFor(() => expect(...))` to wait for the DOM to update

- **fake timers** (`vi.useFakeTimers()`, `vi.advanceTimersByTime(500)`) for debounce, polling and timeouts, so tests do not actually wait
- mock the network so responses are deterministic

A classic bug is a forgotten `await` or `return` on a promise, which makes a test **pass no matter what**.

Example

```
async function fetchUser(id, api) {
  const res = await api(id)
  if (!res.ok) throw new Error('Not found')
  return res.data
}

const api = async (id) => (id === 1 ? { ok: true, data: { name: 'Asha' } } : { ok: false })

// expect(fetchUser(1, api)).resolves / .rejects in Jest or Vitest
console.log(await fetchUser(1, api))
try { await fetchUser(2, api) } catch (e) { console.log('rejected:', e.message) }
```

? Likely follow-up: How do fake timers work?

11

How do you mock network requests? What is MSW?

Medium



#mocking #msw

Options:

- mock the fetch function or HTTP client module (`vi.mock('./api')`): quick, but couples tests to how you fetch
- **MSW (Mock Service Worker)**: intercepts requests at the **network level** and returns responses from handlers you define

MSW is preferred because your code runs unchanged (real `fetch`, real React Query or axios), so tests stay valid if you swap the client. The same handlers work in Node tests, the browser during development and Storybook.

Set up default "happy path" handlers, then override per test with `server.use(...)` for errors, empty lists or slow responses.

Example

```
import { http, HttpResponse } from 'msw'
import { setupServer } from 'msw/node'

const server = setupServer(
  http.get('/api/users', () => HttpResponse.json([{ id: 1, name: 'Asha' }])),
)
beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())

it('shows an error when the API fails', async () => {
  server.use(http.get('/api/users', () => new HttpResponse(null, {
    status: 500 })))
  // render and assert on the error message...
})
```

12

What are snapshot tests? What are their pros and cons?

Easy



#snapshots

A snapshot test saves the rendered output (or any serialisable value) to a file on the first run, then fails if a later run produces something different.

Pros:

- almost free to write
- catches unexpected changes in output such as error messages, config or generated code

Cons:

- large snapshots are not really reviewed; people just press **u** to update, so they **turn into noise**
- they fail on harmless changes (a class name) and do not say what is **correct**
- they test implementation, not behaviour

Use them for small, stable outputs (inline snapshots with **toMatchInlineSnapshot**), not whole component trees. Visual regression tools (Playwright screenshots, Chromatic) are better for UI appearance.

13

What does code coverage measure, and what are its limits?

Medium



#coverage #strategy

Coverage reports how much code ran during tests:

- **statements** and **lines** executed
- **branches**: both sides of each `if`, ternary and `&&`
- **functions** called

It is useful for finding **untested areas**, and a minimum threshold in CI stops coverage from silently dropping.

Limits: coverage shows code was **executed**, not that it was **checked**. A test with no assertions can reach 100%. It also says nothing about missing code, edge cases or wrong requirements, so **high coverage does not mean good tests**.

Chasing 100% leads to brittle tests of trivial code. Many teams aim for roughly 70 to 80%, focused on critical logic. **Mutation testing** (Stryker) measures test quality more honestly.

Likely follow-up: What coverage target would you set, and why?

14

What is TDD? What are its pros and cons?

Easy



#tdd #process

Test-driven development follows a short loop: **red, green, refactor**.

1. write a failing test for the next small behaviour
2. write the simplest code that makes it pass
3. clean up the code with the test as a safety net

Pros: forces you to think about the API and edge cases first, gives high coverage by design, encourages small decoupled units, and gives quick feedback.

Cons: slower at the start, awkward when requirements are unclear or for exploratory UI work, and tests can over-specify implementation if you are not careful.

A balanced answer: TDD works very well for business logic, parsers and bug fixes (write the failing test that reproduces the bug first).

15

Playwright vs Cypress for end-to-end testing?

Medium



#e2e #playwright

Both drive a real browser, auto-wait for elements and have great debugging tools.

Playwright (Microsoft):

- Chromium, Firefox **and WebKit** (Safari)
- runs tests in **parallel** for free, multiple tabs and origins, many languages
- trace viewer, codegen, also handles API testing

Cypress:

- runs inside the browser, with an excellent interactive runner and time-travel debugging
- historically limited to one tab and origin; parallelism mostly through its paid cloud

Today Playwright is the more common choice for new projects. Whichever you use: test critical user journeys only, use role-based locators, and seed data through APIs rather than the UI.

Example

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

test('user can log in', async ({ page }) => {
  await page.goto('/login')
  await page.getByLabel('Email').fill('asha@example.com')
  await page.getByLabel('Password').fill('secret123')
  await page.getByRole('button', { name: 'Log in' }).click()
  await expect(page.getByRole('heading', { name: 'Dashboard'
})).toBeVisible()
})
```

16

What is a flaky test, and how do you fix flaky tests?

Medium



#e2e #ci

A **flaky test** passes and fails on the same code. It destroys trust: people start ignoring red builds.

Common causes and fixes:

- **timing:** fixed `sleep`s or not awaiting UI updates; use auto-waiting assertions, `findBy`, `waitFor`
- **shared state:** tests depending on order or leftover data; reset mocks, databases and storage per test
- **real network or third parties:** mock them or use stable test environments
- **time and randomness:** fake timers, fixed dates, seeded random values
- **animations and parallel runs:** disable animations, isolate data per worker

Process: quarantine, then fix quickly; retries in CI hide the problem and should only be a stopgap.

? Likely follow-up: How would you track down a test that fails only in CI?

17

How do you test a REST API (for example with Supertest)?

Medium



#api #node #integration

Supertest sends HTTP requests to your Express (or similar) app `in memory`, without starting a real server on a port, and lets you assert on status, headers and body.

What to cover per endpoint:

- the happy path (status code and response shape)
- validation errors (`400`), auth (`401` / `403`), not found (`404`)
- side effects: was the record actually saved?

Use a **real test database** (Docker or Testcontainers, or SQLite in memory), reset between tests, so you test real queries. Mock only external services like email or payments. Export the `app` separately from the code that calls `listen` so tests can import it.

Example

```
import request from 'supertest'
import { app } from '../src/app'

describe('POST /api/users', () => {
  it('creates a user', async () => {
    const res = await request(app).post('/api/users').send({ email:
'asha@example.com' })
    expect(res.status).toBe(201)
    expect(res.body).toMatchObject({ id: expect.any(Number), email:
'asha@example.com' })
  })
  it('rejects an invalid email', async () => {
    await request(app).post('/api/users').send({ email: 'nope'
}).expect(400)
  })
})
```

18

How do you test a custom React hook?

Medium



#react #hooks

Two approaches:

- `renderHook` from React Testing Library renders the hook inside a tiny test component and gives you `result.current`. Wrap state updates in `act()` (or use `waitFor` for async ones).
- Test it **through a component** that uses it, which is often more realistic.

If the hook needs context (a router, a QueryClient, a theme), pass a `wrapper` component that provides it.

For pure logic, a good refactor is to `extract it into a plain function` (a reducer or calculator) and unit-test that directly; the hook becomes a thin wrapper.

Example

```
import { act, renderHook } from '@testing-library/react'

it('increments the counter', () => {
  const { result } = renderHook(() => useCounter(5))
  act(() => result.current.increment())
  expect(result.current.count).toBe(6)
})
```

19

How do you mock a module, a date or a timer in Jest/Vitest?

Medium



#mocking #vitest

- **Module:** `vi.mock('./analytics')` replaces every export with mock functions; pass a factory to control them. Calls are hoisted to the top of the file.
- **One method:** `vi.spyOn(obj, 'method').mockReturnValue(...)`, restored with `mockRestore` or `restoreAllMocks` in config.
- **Time:** `vi.useFakeTimers()` then `vi.setSystemTime(new Date('2025-01-01'))` and `vi.advanceTimersByTime(1000)`.
- **Environment:** `vi.stubEnv('API_URL', '...')`, `vi.stubGlobal('fetch', ...)`.

Always reset mocks between tests (`clearMocks` / `restoreMocks` in config) so **state does not leak** from one test into the next.

Example

```
vi.mock('./analytics', () => ({ track: vi.fn() }))
import { track } from './analytics'

it('tracks the purchase', async () => {
  vi.useFakeTimers()
  vi.setSystemTime(new Date('2025-01-01T10:00:00Z'))
  await checkout(cart)
  expect(track).toHaveBeenCalledWith('purchase', { at: '2025-01-01T10:00:00.000Z' })
  vi.useRealTimers()
})
```

20

How do you keep tests independent? What are `beforeEach` and `afterEach` for?

Easy



#basics #setup

Each test should be able to run **alone, in any order, and in parallel** with the same result.

- `beforeEach`: build fresh data and mocks for every test (not shared mutable objects at module level)
- `afterEach`: clean up: restore mocks, clear storage, reset MSW handlers, roll back database transactions
- `beforeAll` / `afterAll`: expensive one-time setup such as starting a test server or database container

Prefer small **factory functions** (`makeUser({ role: 'admin' })`) over large shared fixtures, so each test shows the data that **matters to it**. Testing Library unmounts rendered components automatically after each test.

21

How do tests fit into CI/CD?

Medium



#ci #process

A typical pipeline on every pull request:

1. install, then **lint** and **type-check** (fastest feedback)
2. **unit and integration** tests, with coverage thresholds
3. build the app
4. **E2E** tests against a preview deployment or a local build (often only critical paths, or sharded across machines)

Merging is blocked until checks pass (branch protection). Keep the pipeline fast with caching, parallel jobs and test sharding; **a slow pipeline gets bypassed**.

Store artifacts for debugging failures: Playwright traces, screenshots and videos. Run the full or nightly suite for slower checks such as visual regression and cross-browser runs.

22

How do you test Next.js Server Components and Server Actions?

Hard



#nextjs #strategy

Tooling is still catching up: React Testing Library and Vitest cannot render **async** Server Components directly in a normal `render` call.

Practical strategy:

- move logic out of components into plain functions (data access, validation, formatting) and **unit-test** those
- test **Server Actions** as functions: call them with a `FormData`, mock auth and the database (or use a test database), assert on results and `revalidatePath` calls
- test **Client Components** with RTL as usual
- cover the full server-plus-client flow with **Playwright E2E tests**, which see the real rendered result

Some teams also `await` the component function and render the returned JSX, but that is a workaround, not an official API.

23

You join a project with no tests. Where do you start?

Hard



#strategy #process

Do not try to reach high coverage everywhere. Aim for **the most confidence for the least effort**:

1. add **static checks** first: TypeScript strict mode and ESLint in CI
2. write a few **E2E smoke tests** for the money paths (sign up, login, checkout), which protect the most with the fewest tests
3. add **characterisation tests** around code before you change it, capturing current behaviour even if it is odd
4. for every bug fixed, add a test that reproduces it first
5. unit-test complex business logic as you touch it

Make tests run in CI on every pull request so they keep their value, and keep them fast.