Section 1: The Test Anatomy

1.1 Naming

(1) What is being tested? For example, the ProductsService.addNewProduct method

(2) Under what circumstances and scenario? For example, no price is passed to the method

(3) What is the expected result? For example, the new product is not approved

1.2.Pattern

(1) What is being tested? For example, the ProductsService.addNewProduct method

(2) Under what circumstances and scenario? For example, no price is passed to the method

(3) What is the expected result? For example, the new product is not approved

1.3.Stick to black-box testing: Test only public methods

1.4. Describe expectations in a product language: use BDD-style assertions

1.5 Choose the right test doubles: Avoid mocks in favor of stubs and spies

1.6 Don't "foo", use realistic input data (Faker)

1.7 Test many input combinations using Property-based testing

sending all the possible input combinations to your unit under test it increases the serendipity of finding a bug. js-verify or testcheck checkout fast-check

1.8 If needed, use only short & inline snapshots

1.9 Avoid global test fixtures and seeds, add data per-test

1.10 Don’t catch errors, expect them

1.11 Tag your tests

1.12 Categorize tests under at least 2 levels

1.13 Other generic good testing hygiene

Learn and practice TDD principles—they are extremely valuable for many but don’t get intimidated if they don’t fit your style, you’re not the only one. Consider writing the tests before the code in a red-green-refactor style, ensure each test checks exactly one thing, when you find a bug—before fixing write a test that will detect this bug in the future, let each test fail at least once before turning green, start a module by writing a quick and simplistic code that satisfies the test - then refactor gradually and take it to a production grade level, avoid any dependency on the environment (paths, OS, etc)

Section 2: Backend Testing

2.1 Enrich your testing portfolio: Look beyond unit tests and the pyramid

2.2 Component testing might be your best affair

2.3 Ensure new releases don’t break the API using contract tests

2.4 Test your middlewares in isolation

2.5 Measure and refactor using static analysis tools

2.6 Check your readiness for Hardware-related chaos

2.7 Avoid global test fixtures and seeds, add data per-test

Section 3: Frontend Testing

3.1 Separate UI from functionality

3.2 : Query HTML elements based on attributes that are likely to survive graphic changes unlike CSS selectors and like form labels.

3.3 Whenever possible, test with a realistic and fully rendered component

3.4 Don't sleep, use frameworks built-in support for async events. Also try to speed things up

In many cases, the unit under test completion time is just unknown (e.g. animation suspends element appearance) - in that case, avoid sleeping (e.g. setTimeOut) and prefer more deterministic methods that most platforms provide. Some libraries allows awaiting on operations

3.5 Watch how the content is served over the network

Apply some active monitor that ensures the page load under real network is optimized - this includes any UX concern like slow page load or un-minified bundle. The inspection tools market is no short: basic tools like pingdom, AWS CloudWatch, gcp StackDriver can be easily configured to watch whether the server is alive and response under a reasonable SLA.

3.6 Stub flaky and slow resources like backend APIs

When coding your mainstream tests (not E2E tests), avoid involving any resource that is beyond your responsibility and control like backend API and use stubs instead (i.e. test double). Practically, instead of real network calls to APIs, use some test double library (like Sinon, Test doubles, etc) for stubbing the API response

3.7 Have very few end-to-end tests that spans the whole system to avoid fails from timeouts

3.8 Speed-up E2E tests by reusing login credentials

login only once before the tests execution start (i.e. before-all hook), save the token in some local storage and reuse it across requests

3.9 Have one E2E smoke test that just travels across the site map

For production monitoring and development-time sanity check, run a single E2E test that visits all/most of the site pages and ensures no one breaks. This type of test brings a great return on investment as it's very easy to write and maintain, but it can detect any kind of failure including functional, network and deployment issues

3.10 Expose the tests as a live collaborative document

Besides increasing app reliability, tests bring another attractive opportunity to the table - serve as live app documentation. Since tests inherently speak at a less-technical and product/UX language, using the right tools they can serve as a communication artifact that greatly aligns all the peers - developers and their customers

3.11 Detect visual issues with automated tools

Setup automated tools to capture UI screenshots when changes are presented and detect visual issues like content overlapping or breaking

Section 4: Measuring Test Effectiveness

4.1 Get enough coverage for being confident, ~80% seems to be the lucky number

4.2 Inspect coverage reports to detect untested areas and other oddities

Some issues sneak just under the radar and are really hard to find using traditional tools. These are not really bugs but more of surprising application behavior that might have a severe impact.

4.3 Measure logical coverage using mutation testing

The Traditional Coverage metric often lies: It may show you 100% code coverage, but none of your functions, even not one, return the right response. How come? it simply measures over which lines of code the test visited, but it doesn't check if the tests actually tested anything—asserted for the right response. Like someone who's traveling for business and showing his passport stamps—this doesn't prove any work done, only that he visited few airports and hotels

Mutation-based testing is here to help by measuring the amount of code that was actually TESTED not just VISITED. Stryker is a JavaScript library for mutation testing and the implementation is really neat.

4.4 Preventing test code issues with Test linters

A set of ESLint plugins were built specifically for inspecting the tests code patterns and discover issues. For example, eslint-plugin-mocha will warn when a test is written at the global level (not a son of a describe() statement) or when tests are skipped which might lead to a false belief that all tests are passing. Similarly, eslint-plugin-jest can, for example, warn when a test has no assertions at all (not checking anything)

Section 5: CI and Other Quality Measures

5.1 Enrich your linters and abort builds that have linting issues

Linters are a free lunch, with 5 min setup you get for free an auto-pilot guarding your code and catching significant issue as you type.

5.2 Shorten the feedback loop with local developer-CI

Using a CI with shiny quality inspections like testing, linting, vulnerabilities check, etc? Help developers run this pipeline also locally to solicit instant feedback and shorten the feedback loop. Why? an efficient testing process constitutes many and iterative loops: (1) try-outs -> (2) feedback -> (3) refactor. The faster the feedback is, the more improvement iterations a developer can perform per-module and perfect the results. On the flip, when the feedback is late to come fewer improvement iterations could be packed into a single day, the team might already move forward to another topic/task/module and might not be up for refining that module.

5.3 Perform e2e testing over a true production-mirror

End to end (e2e) testing are the main challenge of every CI pipeline—creating an identical ephemeral production mirror on the fly with all the related cloud services can be tedious and expensive

5.4 Parallelize test execution

When done right, testing is your 24/7 friend providing almost instant feedback. In practice, executing 500 CPU-bounded unit test on a single thread can take too long. Luckily, modern test runners and CI platforms (like Jest, AVA and Mocha extensions) can parallelize the test into multiple processes and achieve significant improvement in feedback time. Some CI vendors do also parallelize tests across

containers (!) which shortens the feedback loop even further. Whether locally over multiple processes, or over some cloud CLI using multiple machines—parallelizing demand keeping the tests autonomous as each might run on different processes

5.5 Stay away from legal issues using license and plagiarism check

A bunch of npm packages like license check and plagiarism check (commercial with free plan) can be easily baked into your CI pipeline and inspect for sorrows like dependencies with restrictive licenses or code that was copy-pasted from Stack Overflow and apparently violates some copyrights

5.6 Constantly inspect for vulnerable dependencies

This can get easily tamed using community tools such as npm audit, or commercial tools like snyk (offer also a free community version). Both can be invoked from your CI on every build

5.7 Automate dependency updates

(1) CI can fail builds that have obsolete dependencies—using tools like 'npm outdated' or 'npm-check-updates (ncu)' . Doing so will enforce developers to update dependencies.

(2) Use commercial tools that scan the code and automatically send pull requests with updated dependencies. One interesting question remaining is what should be the dependency update policy—updating on every patch generates too many overhead, updating right when a major is released might point to an unstable version (many packages found vulnerable on the very first days after being released, see the eslint-scope incident).

5.8 Other, non-Node related, CI tips

1. Use a declarative syntax. This is the only option for most vendors but older versions of Jenkins allows using code or UI
2. Opt for a vendor that has native Docker support
3. Fail early, run your fastest tests first. Create a 'Smoke testing' step/milestone that groups multiple fast inspections (e.g. linting, unit tests) and provide snappy feedback to the code committer
4. Make it easy to skim-through all build artifacts including test reports, coverage reports, mutation reports, logs, etc
5. Create multiple pipelines/jobs for each event, reuse steps between them. For example, configure a job for feature branch commits and a different one for master PR. Let each reuse logic using shared steps (most vendors provide some mechanism for code reuse)
6. Never embed secrets in a job declaration, grab them from a secret store or from the job's configuration

7. Explicitly bump version in a release build or at least ensure the developer did so
8. Build only once and perform all the inspections over the single build artifact (e.g. Docker image)
9. Test in an ephemeral environment that doesn't drift state between builds. Caching node_modules might be the only exception

5.9 Build matrix: Run the same CI steps using multiple Node versions

Quality checking is about serendipity, the more ground you cover the luckier you get in detecting issues early. When developing reusable packages or running a multi-customer production with various configuration and Node versions, the CI must run the pipeline of tests over all the permutations of configurations. For example, assuming we use MySQL for some customers and Postgres for others—some CI vendors support a feature called 'Matrix' which allow running the suit of testing against all permutations of MySQL, Postgres and multiple Node version like 8, 9 and 10. This is done using configuration only without any additional effort (assuming you have testing or any other quality checks). Other CIs who doesn't support Matrix might have extensions or tweaks to allow that.