JavaScript Testing: Unit vs Functional vs Integration Tests

Unit tests, integration tests, and functional tests are all types of automated tests which form essential cornerstones of continuous delivery, a development methodology that allows you to safely ship changes to production in days or hours rather than months or years.

Automated tests enhance software stability by catching more errors before software reaches the end user. They provide a safety net that allows developers to make changes without fear that they will unknowingly break something in the process.

The Cost of Neglecting Tests

Contrary to popular intuition, maintaining a quality test suite can dramatically enhance developer productivity by catching errors immediately. Without them, end users encounter more bugs, which can lead to increased reliance on customer service, quality assurance teams, and bug reports.

Test Driven Development takes a little more time up front, but bugs that reach customers cost more in many ways:

They interrupt the user experience, which can cost you in sales, usage metrics, they can even drive customers away permanently.
Every bug report must be validated by QA or developers.
Bug fixes are interruptions which cause a costly context switch. Each interruption can waste up to 20 minutes per bug, not counting the actual fix.
Bug diagnosis happens outside the normal context of feature development, sometimes by different developers who are unfamiliar with the code and the surrounding implications of it.
Opportunity cost: The development team must wait for bug fixes before they can continue working on the planned development roadmap.

The cost of a bug that makes it into production is many times larger than the cost of a bug caught by an automated test suite. In other words, TDD has an overwhelmingly positive ROI.

Different Types of Tests

The first thing you need to understand about different types of tests is that they all have a job to do. They play important roles in continuous delivery.

A while back, I was consulting on an ambitious project where the team was having a hard time building a reliable test suite. Because it was hard to use and understand, it rarely got used or maintained.

One of the problems I observed with the existing test suite is that it confused unit tests, functional tests, and integration tests. It made absolutely no distinction between any of them.

The result was a test suite that was not particularly well suited for anything.

Roles Tests Play in Continuous Delivery

Each type of test has a unique role to play. You don't choose between unit tests, functional tests, and integration tests. Use all of them, and make sure you can run each type of test suite in isolation from the others.

Most apps will require both unit tests and functional tests, and many complex apps will also require integration tests.

Unit tests ensure that individual components of the app work as expected. Assertions test the component API.
Integration tests ensure that component collaborations work as expected. Assertions may test component API, UI, or side-effects (such as database I/O, logging, etc…)
Functional tests ensure that the app works as expected from the user's perspective. Assertions primarily test the user interface.

You should isolate unit tests, integration tests, and functional tests from each other so that you can easily run them separately during different phases of development. During continuous integration, tests are frequently used in three ways:

During development, for developer feedback. Unit tests are particularly helpful here.
In the staging environment, to detect problems and stop the deploy process if something goes wrong. Typically the full suite of all test types are run at this stage.
In the production environment, a subset of production-safe functional tests known as smoke tests are run to ensure that none of the critical functionality was broken during the deploy process.

Which Test Types Should You Use? All of Them.

In order to understand how different tests fit in your software development process, you need to understand that each kind of test has a job to do, and those tests roughly fall into three broad categories:

User experience tests (end user experience)
Developer API tests (developer experience)
Infrastructure tests (load tests, network integration tests, etc…)

User experience tests examine the system from the perspective of the user, using the actual user interface, typically using the target platforms or devices.

Developer API tests examine the system from the perspective of a developer. When I say API, I don't mean HTTP APIs. I mean the surface area API of a unit: the interface used by developers to interact with the module, function, class, etc…

Unit Tests: Realtime Developer Feedback

Unit tests ensure that individual components work in isolation from each other. Units are typically modules, functions, etc…

For example, your app may need to route URLs to route handlers. A unit test may be written against the URL parser to ensure that the relevant components of the URL are parsed correctly. Another unit test might ensure that the router calls the correct handler for a given URL.

However, if you want to test that when a specific URL is posted to, a corresponding record gets added to the database, that would be an integration test, not a unit test.

Unit tests are frequently used as a developer feedback mechanism during development. For example, I run lint and unit tests on every file change and monitor the results in a development console which gives me real-time feedback as I'm working.

For this to work well, unit tests must run very quickly, which means that asynchronous operations such as network and file I/O should be avoided in unit tests.

Since integration tests and functional tests very frequently rely on network connections and file I/O, they tend to significantly slow down the test run when there are lots of tests, which can stretch the run time from milliseconds into minutes. In the case of very large apps, a complete functional test run can take more than an hour.

Unit tests should be:

Dead simple.
Lightning fast.
A good bug report.

What do I mean by "a good bug report?"

I mean that whatever test runner and assertion library you use, a failing unit test should tell you at a glance:

1. Which component is under test?
2. What is the expected behavior?

3. What was the actual result?
4. What is the expected result?
5. How is the behavior reproduced?

The first four questions should be visible in the failure report. The last question should be clear from the test's implementation. Some assertion types are not capable of answering all those questions in a failure report, but most `equal`, `same`, or `deepEqual` assertions should. In fact, if those were the only assertions in any assertion library, most test suites would probably be better off. Simplify.

Here are some simple unit test examples from real projects using Tape:

```
import test from 'tape';
import hello from 'store/reducers/hello';

test('...initial', assert => {
  const message = `should set { mode: 'display', subject: 'world' }`;
  const expected = {
    mode: 'display',
    subject: 'World'
  };

  const actual = hello();

  assert.deepEqual(actual, expected, message);
  assert.end();
});
```

```
import test from 'tape',
import credential from '../credential';

test('hash', function (t) {

  const pw = credential();


  pw.hash('foo', function (err, hash) {
    t.error(err, 'should not throw an error');
```

```
    t.ok(JSON.parse(hash).hash,
      'should be a json string representing the hash.');

    t.end();
  });
});
```

Integration Tests

Integration tests ensure that various units work together correctly. For example, a Node route handler might take a logger as a dependency. An integration test might hit that route and test that the connection was properly logged.

In this case, we have two units under test:

1. The route handler
2. The logger

If we were unit testing the logger, our tests wouldn't invoke the route handler, or know anything about it.

If we were unit testing the route handler, our tests would stub the logger, and ignore the interactions with it, testing only whether or not the route responded appropriately to the faked request.

Let's look at this in more depth. The route handler is a factory function which uses dependency injection to inject the logger into the route handler. Let's look at the signature (See the rtype docs for help reading signatures):

```
createRoute({ logger: LoggerInstance }) => RouteHandler
```

Let's see how we can test this:

```
import test from 'tape';

import createLog from 'shared/logger';
import routeRoute from 'routes/my-route';
```

```
test('logger/route integration', assert => {
  const msg = 'Logger logs router calls to memory';

  const logMsg = 'hello';
  const url = `http://127.0.0.1/msg/${ logMsg }`;

  const logger = createLog({ output: 'memory' });
  const routeHandler = createRoute({ logger });

  routeHandler({ url });

  const actual = logger.memoryLog[0];
  const expected = logMsg;

  assert.equal(actual, expected, msg);
  assert.end();
});
```

We'll walk through the important bits in more detail. First, we create the logger and tell it to log in memory:

```
const logger = createLog({ output: 'memory' });
```

Create the router and pass in the logger dependency. This is how the router accesses the logger API. Note that in your unit tests, you can stub the logger and test the route in isolation:

```
const routeHandler = createRoute({ logger });
```

Call the route handler with a fake request object to test the logging:

```
routeHandler({ url });
```

The logger should respond by adding the message to the in-memory log. All we need to do now is check to see if the message is there:

```
  const actual = logger.memoryLog[0];
```

Similarly, for APIs that write to a database, you can connect to the

database and check to see if the data is updated correctly, etc...

Many integration tests test interactions with services, such as 3rd party APIs, and may need to hit the network in order to work. For this reason, integration tests should always be kept separate from unit tests, in order to keep the unit tests running as quickly as they can.

Functional Tests

Functional tests are automated tests which ensure that your application does what it's supposed to do from the point of view of the user. Functional tests feed input to the user interface, and make assertions about the output that ensure that the software responds the way it should.

Functional tests are sometimes called end-to-end tests because they test the entire application, and it's hardware and networking infrastructure, from the front end UI to the back end database systems. In that sense, functional tests are also a form of integration testing, ensuring that machines and component collaborations are working as expected.

Functional tests typically have thorough tests for "happy paths" — ensuring the critical app capabilities, such as user logins, signups, purchase work flows, and all the critical user workflows all behave as expected.

Functional tests should be able to run in the cloud on services such as Sauce Labs, which typically use the WebDriver API via projects like Selenium.

That takes a bit of juggling. Luckily, there are some great open source projects that make it fairly easy.

My favorite is Nightwatch.js. Here's what a simple Nightwatch functional test suite looks like this example from the Nightwatch docs:

```
module.exports = {
  'Demo test Google' : function (browser) {
    browser
      .url('http://www.google.com')
```

```
      .waitForElementVisible('body', 1000)
      .setValue('input[type=text]', 'nightwatch')
      .waitForElementVisible('button[name=btnG]', 1000)
      .click('button[name=btnG]')
      .pause(1000)
      .assert.containsText('#main', 'Night Watch')
      .end();
  }
};
```

As you can see, functional tests hit real URLs, both in staging environments, and in production. They work by simulating actions the end user might take in order to accomplish their goals in your app. They can click buttons, input text, wait for things to happen on the page, and make assertions by looking at the actual UI output.

Smoke Tests

After you deploy a new release to production, it's important to find out right away whether or not it's working as expected in the production environment. You don't want your users to find the bugs before you do — it could chase them away!

It's important to maintain a suite of automated functional tests that act like smoke tests for your newly deployed releases. Test all the critical functionality in your app: The stuff that most users will encounter in a typical session.

Smoke tests are not the only use for functional tests, but in my opinion, they're the most valuable.

What Is Continuous Delivery?

Prior to the continuous delivery revolution, software was released using a waterfall process. Software would go through the following steps, one at a time. Each step had to be completed before moving on to the next:

1. Requirement gathering

2. Design
3. Implementation
4. Verification
5. Deployment
6. Maintenance

It's called waterfall because if you chart it with time running from right to left, it looks like a waterfall cascading from one task to the next. In other words, in theory, you can't really do these things concurrently.

In theory. In reality, a lot of project scope is discovered as the project is being developed, and scope creep often leads to disastrous project delays and rework. Inevitably, the business team will also want "simple changes" made after delivery without going through the whole expensive, time-consuming waterfall process again, which frequently results in an endless cycle of change management meetings and production hot fixes.

A clean waterfall process is probably a myth. I've had a long career and consulted with hundreds of companies, and I've never seen the theoretical waterfall work the way it's supposed to in real life. Typical waterfall release cycles can take months or years.

The Continuous Delivery Solution

Continuous delivery is a development methodology that acknowledges that scope is uncovered as the project progresses, and encourages incremental improvements to software in short cycles that ensure that software can be released at any time without causing problems.

With continuous delivery, changes can ship safely in a matter of hours.

In contrast to the waterfall method, I've seen the continuous delivery process running smoothly at dozens of organizations — but I've never seen it work anywhere without a quality array of test suites that includes both unit tests and functional tests, and frequently includes integration tests, as well.

Hopefully now you have everything you need to get started on your continuous delivery foundations.

Conclusion

As you can see, each type of test has an important part to play. Unit tests for fast developer feedback, integration tests to cover all the corner cases of component integrations, and functional tests to make sure everything works right for the end users.

How do you use automated tests in your code, and how does it impact your confidence and productivity? Let me know in the comments.

Meet the author

Eric Elliott

Eric Elliott is the author of Programming JavaScript Applications (O'Reilly) and Learn JavaScript with Eric Elliott. He has contributed to software experiences for Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC, and top recording artists including Usher, Frank Ocean, Metallica, and many more. He spends most of his time in the San Francisco Bay Area with the most beautiful woman in the world.