E2e testing:
http://engineering.wingify.com/posts/e2e-testing-with-webdriverjs-jasmine/

Polyfills & shims

polyfill is a browser fallback, made in javascript, that allows functionality you expect to work in modern browsers to work in older browsers. Ie to support canvas (an html5 feature) in older browsers.

It's sort of an HTML5 technique, since it is used in conjunction with HTML5, but it's not part of HTML5, and you can have polyfills without having HTML5 (for example, to support CSS3 techniques you want).

Here's a good post:

http://remysharp.com/2010/10/08/what-is-a-polyfill/
Here's a comprehensive list of Polyfills and Shims:

https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills

http://stackoverflow.com/questions/6599815/what-is-the-difference-between-a-shim-and-a-polyfill

Definitions

Alex Sexton also classifies polyfilling as a form of Regressive Enhancement. I think that sums it up nicely.

Paul also defines it as:

> A shim that mimics a future API providing fallback functionality to older browsers.

What is Mocking?
•Fake function that records how it is used (arguments, return value, etc)

Spies
•Fake function with predefined behavior

Stubs
•Fake function (spy & stub) where you define expectations in advance & assert

Mocks

Fake function (spy & stub) where you define expectations in advance & assert

Best Practices for Spies, Stubs and Mocks in Sinon.js

Sinon.js is a vital tool when writing JavaScript unit tests. This article shows you best

http://codeutopia.net/blog/

http://kentor.me/posts/testing-react-and-flux-applications-with-karma-and-webpack/

http://code.tutsplus.com/tutorials/headless-functional-testing-with-selenium-and-phantomjs--net-30545

Introduction

Testing code with Ajax, networking, timeouts, databases, or other dependencies can be difficult. For example, if you use Ajax or networking, you need to have a server, which responds to your requests. With databases, you need to have a testing database set up with data for your tests.

All of this means that writing and running tests is harder, because you need to do extra work to prepare and set up an environment where your tests can succeed.

Thankfully, we can use Sinon.js to avoid all the hassles involved. We can make use of its features to simplify the above cases into just a few lines of code.

However, getting started with Sinon might be tricky. You get a lot of functionality in the form of what it calls spies, stubs and mocks, but it can be difficult to choose when to use what. They also have some gotchas, so you need to know what you're doing to avoid problems.

In this article, we'll show you the differences between spies, stubs and mocks, when and how to use them, and give you a set of best practices to help you avoid common pitfalls.

Example Function

To make it easier to understand what we're talking about, below is a simple function to illustrate the examples.

```
function setupNewUser(info, callback) {

  var user = {

    name: info.name,

    nameLowercase: info.name.toLowerCase()

  };



  try {

    Database.save(user, callback);

  }

  catch(err) {

    callback(err);

  }

}
```

The function takes two parameters — an object with some data we want to save and a callback function. We put the data from the `info` object into the`user` variable, and save it to a database. For the purpose of this tutorial, what`save` does is irrelevant — it could send an Ajax request, or, if this was Node.js code, maybe it would talk directly to the database, but the specifics don't matter. Just imagine it does some kind of a data-saving operation.

Spies, Stubs and Mocks

Together, spies, stubs and mocks are known as *test doubles*. Similar to how stunt doubles do the dangerous work in movies, we use test doubles to replace troublemakers and make tests easier to write.

When Do You Need Test Doubles?

To best understand when to use test-doubles, we need to understand the two different types of functions we can have. We can split functions into two categories:

- Functions *without* side effects
- And functions *with* side effects

Functions without side effects are simple: the result of such a function is only dependent on its parameters — the function always returns the same value given the same parameters.

A function with side effects can be defined as a function that depends on something external, such as the state of some object, the current time, a call to a database, or some other mechanism that holds some kind of state. The result of such a function can be affected by a variety of things in addition to its parameters.

If you look back at the example function, we call two functions in it —`toLowerCase`, and `Database.save`. The former has no side effects - the result of`toLowerCase` only depends on the value of the string. However, the latter has a side effect - as previously mentioned, it does some kind of a save operation, so the result of `Database.save` is also affected by that action.

If we want to test `setupNewUser`, we may need to use a test-double on`Database.save` because it has a side effect. In other words, we can say that **we need test-doubles when the function has side effects**.

In addition to functions with side effects, we may occasionally need test doubles with functions that are causing problems in our tests. A common case is when a function performs a calculation or some other operation which is very slow and which makes our tests slow. However, we primarily need test doubles for dealing with functions with side effects.

When to Use Spies

As the name might suggest, spies are used to get information about function calls. For example, a spy can tell us how many times a function was called, what arguments each call had, what values were returned, what errors were thrown, etc.

As such, a spy is a good choice whenever the goal of a test is to verify something happened. Combined with Sinon's assertions, we can check many different results by using a simple spy.

The most common scenarios with spies involve...

- Checking how many times a function was called
- Checking what arguments were passed to a function

We can check how many times a function was called using`sinon.assert.callCount`, `sinon.assert.calledOnce`, `sinon.assert.notCalled`, and similar. For example, here's how to verify the save function was being called:

```
it('should call save once', function() {

  var save = sinon.spy(Database, 'save');



  setupNewUser({ name: 'test' }, function() { });



  save.restore();

  sinon.assert.calledOnce(save);

});
```

We can check what arguments were passed to a function using`sinon.assert.calledWith`, or by accessing the call directly using `spy.lastCall` or`spy.getCall()`. For example, if we wanted to verify the aforementioned save function receives the correct parameters, we would use the following spec:

```
it('should pass object with correct values to save', function() {

  var save = sinon.spy(Database, 'save');

  var info = { name: 'test' };
```

```
  var expectedUser = {

    name: info.name,

    nameLowercase: info.name.toLowerCase()

  };



  setupNewUser(info, function() { });



  save.restore();

  sinon.assert.calledWith(save, expectedUser);

});
```

These are not the only things you can check with spies though — Sinon provides many other assertions you can use to check a variety of different things. The same assertions can also be used with stubs.

If you spy on a function, the function's behavior is not affected. If you want to change how a function behaves, you need a stub.

When to Use Stubs

Stubs are like spies, except in that they replace the target function. They can also contain custom behavior, such as returning values or throwing exceptions. They can even automatically call any callback functions provided as parameters.

Stubs have a few common uses:

- You can use them to replace problematic pieces of code
- You can use them to trigger code paths that wouldn't otherwise trigger - such as error handling
- You can use them to help test asynchronous code more easily

Stubs can be used to replace problematic code, i.e. the code that makes writing tests difficult. This is often caused by something external - a network connection, a database, or some other non-JavaScript system. The problem with these is that they often require manual setup. For example, we would need to fill a database with test data before running our tests, which makes running and writing them more complicated.

If we *stub out* a problematic piece of code instead, we can avoid these issues entirely. Our earlier example uses `Database.save` which could prove to be a problem if we don't set up the database before running our tests. Therefore, it might be a good idea to use a stub on it, instead of a spy.

```
it('should pass object with correct values to save', function() {

  var save = sinon.stub(Database, 'save');

  var info = { name: 'test' };

  var expectedUser = {

    name: info.name,

    nameLowercase: info.name.toLowerCase()

  };



  setupNewUser(info, function() { });



  save.restore();

  sinon.assert.calledWith(save, expectedUser);

});
```

By replacing the database-related function with a stub, we no longer need an actual database for our test. A similar approach can be used in nearly any situation involving code that is otherwise hard to test.

Stubs can also be used to trigger different code paths. If the code we're testing calls another function, we sometimes need to test how it would behave under unusual conditions — most commonly if there's an error. We can make use of a stub to trigger an error from the code:

```
it('should pass the error into the callback if save fails',
function() {

  var expectedError = new Error('oops');

  var save = sinon.stub(Database, 'save');

  save.throws(expectedError);

  var callback = sinon.spy();



  setupNewUser({ name: 'foo' }, callback);



  save.restore();

  sinon.assert.calledWith(callback, expectedError);

});
```

Thirdly, stubs can be used to simplify testing asynchronous code. If we stub out an asynchronous function, we can force it to call a callback right away, making the test synchronous and removing the need of asynchronous test handling.

```
it('should pass the database result into the callback', function() {

  var expectedResult = { success: true };

  var save = sinon.stub(Database, 'save');

  save.yields(null, expectedResult);
```

```
  var callback = sinon.spy();



  setupNewUser({ name: 'foo' }, callback);



  save.restore();

  sinon.assert.calledWith(callback, null, expectedResult);

});
```

Stubs are highly configurable, and can do a lot more than this, but most follow these basic ideas.

When to Use Mocks

You should take care when using mocks - it's easy to overlook spies and stubs when mocks can do everything they can, but mocks also easily make your tests overly specific, which leads to brittle tests that break easily. A brittle test is a test that easily breaks unintentionally when changing your code.

Mocks should be used primarily when you would use a stub, but need to verify multiple more specific behaviors on it.

For example, here's how we could verify a more specific database saving scenario using a mock:

```
it('should pass object with correct values to save only once',
function() {

  var info = { name: 'test' };

  var expectedUser = {

    name: info.name,

    nameLowercase: info.name.toLowerCase()
```

```
  };

  var database = sinon.mock(Database);

  database.expects('save').once().withArgs(expectedUser);



  setupNewUser(info, function() { });



  database.verify();

  database.restore();

});
```

Note that, with a mock, we define our expectations up front. Normally, the expectations would come last in the form of an assert function call. With a mock, we define it directly on the mocked function, and then only call `verify`in the end.

In this test, we're using `once` and `withArgs` to define a mock which checks both the number of calls *and* the arguments given. If we use a stub, checking multiple conditions require multiple assertions, which can be a code smell.

Because of this convenience in declaring multiple conditions for the mock, it's easy to go overboard. We can easily make the conditions for the mock more specific than is needed, which can make the test harder to understand and easy to break. This is also one of the reasons to avoid multiple assertions, so keep this in mind when using mocks.

Best Practices and Tips

Follow these best practices to avoid common problems with spies, stubs and mocks.

Use sinon.test Whenever Possible

When you use spies, stubs or mocks, wrap your test function in `sinon.test`. This allows you to use Sinon's automatic clean-up functionality. Without it, if your test fails before your test-doubles are cleaned up, it can cause a*cascading failure* - more test failures resulting from

the initial failure. Cascading failures can easily mask the real source of the problem, so we want to avoid them where possible.

Using `sinon.test` eliminates this case of cascading failures. Here's one of the tests we wrote earlier:

```
it('should call save once', function() {

  var save = sinon.spy(Database, 'save');



  setupNewUser({ name: 'test' }, function() { });



  save.restore();

  sinon.assert.calledOnce(save);

});
```

If `setupNewUser` threw an exception in this test, that would mean the spy would never get cleaned up, which would wreak havoc in any following tests.

We can avoid this by using `sinon.test` as follows:

```
it('should call save once', sinon.test(function() {

  var save = this.spy(Database, 'save');



  setupNewUser({ name: 'test' }, function() { });



  sinon.assert.calledOnce(save);

}));
```

Note the three differences: in the first line, we wrap the test function with`sinon.test`. In the second line, we use `this.spy` instead of `sinon.spy`. And lastly, we removed the `save.restore` call, as it's now being cleaned up automatically.

You can make use of this mechanism with all three test doubles:

- `sinon.spy` becomes `this.spy`
- `sinon.stub` becomes `this.stub`
- `sinon.mock` becomes `this.mock`

Async Tests with sinon.test

You may need to disable fake timers for async tests when using `sinon.test`. This is a potential source of confusion when using Mocha's asynchronous tests together with `sinon.test`.

To make a test asynchronous with Mocha, you can add an extra parameter into the test function:

```
it('should do something async', function(done) {
```

This can break when combined with `sinon.test`:

```
it('should do something async', sinon.test(function(done) {
```

Combining these can cause the test to fail for no apparent reason, displaying a message about the test timing out. This is caused by Sinon's fake timers which are enabled by default for tests wrapped with `sinon.test`, so you'll need to disable them.

This can be fixed by changing `sinon.config` somewhere in your test code or in a configuration file loaded with your tests:

```
sinon.config = {

  useFakeTimers: false

};
```

`sinon.config` controls the default behavior of some functions like `sinon.test`. It also has some other available options.

Create Shared Stubs in beforeEach

If you need to replace a certain function with a stub in all of your tests, consider stubbing it out in a `beforeEach` hook. For example, all of our tests were using a test-double for `Database.save`, so we could do the following:

```
describe('Something', function() {

  var save;

  beforeEach(function() {

    save = sinon.stub(Database, 'save');

  });



  afterEach(function() {

    save.restore();

  });



  it('should do something', function() {

    //you can use the stub in tests by accessing the variable

    save.yields('something');

  });

});
```

Make sure to also add an `afterEach` and clean up the stub. Without it, the stub may be left in place and it may cause problems in other tests.

Checking the Order of Function Calls or Values Being Set

If you need to check that certain functions are called in order, you can use spies or stubs together with `sinon.assert.callOrder`:

```
var a = sinon.spy();

var b = sinon.spy();



a();

b();



sinon.assert.callOrder(a, b);
```

If you need to check that a certain value is set before a function is called, you can use the third parameter of `stub` to insert an assertion into the stub:

```
var object = { };

var expectedValue = 'something';

var func = sinon.stub(example, 'func', function() {

  assert.equal(object.value, expectedValue);

});



doSomethingWithObject(object);



sinon.assert.calledOnce(func);
```

The assertion within the stub ensures the value is set correctly before the stubbed function is called. Remember to also include a`sinon.assert.calledOnce` check to ensure the stub gets called. Without it, your test will not fail when the stub is not called.

Conclusion

Sinon is a powerful tool, and, by following the practices laid out in this tutorial, you can avoid the most common problems developers run into when using it. The most important thing to remember is to make use of `sinon.test` — otherwise, cascading failures can be a big source of frustration.

Why Use Fakes?

In a unit test, you might want to avoid having to test the unit's dependencies. This is especially true in white-box testing. In this case, test fakes are going to be very helpful. Sinon provides several fakes, notably spies, stubs, and mocks. Let's compare and contrast the three:

Sinon Spies

Spies sound like what they do – they watch your functions and report back on how they are called. They generally avoid the violence and mayhem of a Hollywood spy, but depending on your application, this could vary.

They don't change the functionality of your application. They simply report what they see. The sinon API for spies is fairly large, but it essentially centers around the `called` attribute (of which there are many variations).
I first setup that I want to spy on something. Then I call my subject under test (src code). Then I verify with the spy what was actually called and stop spying. That might look like this in a test:

```
describe("#fight", function () {
  it("calls prayForStrength for fight success", function () {
    sinon.spy(subject.strengthDep, "prayForStrength");
    subject.fight();
    subject.strengthDep.called.should.be.true;
```

```
    subject.strengthDep.restore();
  });
});
```

Note: this example is in mocha using a should.js assertion style
The dependency's `prayForStrength` method is referred to by name in a string to setup the spy. When `fight` is called here, `strengthDep.prayForStrength` will be called as normal – but there will be someone watching. Finally, we call `restore` on the function we spied on so that all spies are called off. If you want to do more than watch as dependencies work as described, you might want to use a stub.

Sinon Stubs

Stubs are more hands-on than spies (though they sound more useless, don't they). With a stub, you will actually change how functions are called in your test. You don't want to change the subject under test, thus changing the accuracy of your test. But you may want to test several ways that dependencies of your unit could be expected to act.

For instance, if you had a function that returned a boolean that your code used to do different things, you might want to use a stub in two different tests to verify conditions when returning different values (ie, guarantee one run of `true` and one of `false` return).
To continue the `fight` example from above, let's assume that if `prayForStrength` returns true, we are guaranteed to win the fight for the orphans (ie, `fight()` should return `true`). That might look like this:

```
describe("#fight", function () {
  it("always wins when prayForStrength is true", function () {
    sinon.stub(subject.strengthDep, "prayForStrength", function () { return true; });
    subject.fight().should.be.true;
    subject.strengthDep.restore();
  });
});
```

Notice that we use a different `sinon.stub` API. For the 3rd parameter, we're supplying our own version of `prayForStrength`. For our test, all we care about is the return value, so that's all we supply. We're not testing this dependency. We're instead testing how our subject `fight`s in a certain circumstance. There are many ways you can use sinon stubs to control how functions are called. Also note that you can still use the `called` verifications with stubs. But if you do verify a stub was called, you may want to use a mock.

Sinon Mocks

Mocks take the attributes of spies and stubs, smashes them together and changes the style a bit. A mock will both observe the calling of functions and verify that they were called in some specific way. And all this setup happens *previous* to calling your subject under test. After the call, mocks are simply asked if all went to plan.
So the previous test could be rewritten to use a mock:

```
describe("#fight", function () {
  it("always wins when prayForStrength is true", function () {
    var mock = sinon.mock(subject.strengthDep)
    mock.expects("prayForStrength").returns(true);
    subject.fight().should.be.true;
    mock.verify();
    mock.restore();
  });
});
```

The `expects` and `returns` line is where the combo magic happens. `expects` is verifying a call (like `spies` can), and `returns` is specifying functionality (like `stubs` can). The `verify` call is the line that will fail (essentially the mock assertion) if things in the subject didn't go exactly according to plan.

Spies vs. Stubs vs. Mocks

So when should I use spies or stubs or mocks? As with most art, there are many ways to accomplish what you want. Much of your choice will depend on your own style and what you become proficient in.

Some basic rules might be:

- ***Use Spies*** - if you simply want to watch and verify somethings happens in your test case.
- ***Use Stubs*** - if you simply want to specify how something will work to help your test case.
- ***Use Mocks*** - if you want to both of the above on a single dependency in your test case.

When do you find yourself most often using spies vs. stubs vs. mocks?

RENDERING TO THE DOM

Sometimes though we need to be able to have a DOM to fully render components and simulate user interactions on those components. Given that we're running our tests in NodeJS we need to do some work to set everything up correctly. Thankfully the module we installed earlier, jsdom, does a great job of sorting all this out for us. Create the file `tests/setup.js`:

```
import jsdom from 'jsdom';

function setupDom() {
  if (typeof document !== 'undefined') {
    return;
  }

  global.document = jsdom.jsdom('<html><body></body></html>');
  global.window = document.defaultView;
  global.navigator = window.navigator;
};

setupDom();
```

This file will check to see if `document` is defined, and set it up if it isn't. Note that you'll need to import this file before you import React, because React does some checks when it loads to see if a DOM is present. Update the top of `test/todo-test.js` to import this file:

```
import './setup';

import React from 'react';
// rest of imports and tests below
```

Next, we can write a test that ensures when we click on a Todo to toggle it between done and incomplete the right callback is called. If you recall from looking at the implementation of the `Todo` component we give it a property that it should call when a user toggles a todo. We'll test that when we click on a todo the component does call that function. This time though we'll be rendering into the DOM and simulating user interaction.

```
t.test('toggling a TODO calls the given prop', (t) => {
  t.plan(1);

  const doneCallback = (id) => t.equal(id, 1);
  const todo = { id: 1, name: 'Buy Milk', done: false };

  const result = TestUtils.renderIntoDocument(
    <Todo todo={todo} doneChange={doneCallback} />
  );
```

```
  // assertion will go here
});
```

Using `renderIntoDocument` from `TestUtils` we can take the component and render it into the DOM. Note that each `renderIntoDocument` renders into a detached DOM node, which means no one test can interfere with another. I pass the component `doneCallback`, which is a function that takes in the ID of the todo and makes an assertion that it will be equal to `1`, which is the ID of the given todo in the test. If this callback never gets called then Tape will detect that we haven't made all the assertions we planned and the test will fail.

Finally, we can use `findRenderedDOMComponentWithTag` to search for the paragraph element in our rendered todo component:

```
const todoText = TestUtils.findRenderedDOMComponentWithTag(result, 'p');
```

And then use `TestUtils.Simulate` to simulate a user interaction:

```
TestUtils.Simulate.click(todoText);
```

This click should cause our callback to be executed and therefore our assertion to run. Here's the full test:

```
t.test('toggling a TODO calls the given prop', (t) => {
  t.plan(1);

  const doneCallback = (id) => t.equal(id, 1);
  const todo = { id: 1, name: 'Buy Milk', done: false };

  const result = TestUtils.renderIntoDocument(
    <Todo todo={todo} doneChange={doneCallback} />
  );

  const todoText = TestUtils.findRenderedDOMComponentWithTag(result, 'p');
  TestUtils.Simulate.click(todoText);
});
```

We can write another test that looks very similar. This one tests that when we click the Delete link the correct callback is called:

```
t.test('deleting a TODO calls the given prop', (t) => {
  t.plan(1);
  const deleteCallback = (id) => t.equal(id, 1);
  const todo = { id: 1, name: 'Buy Milk', done: false };

  const result = TestUtils.renderIntoDocument(
    <Todo todo={todo} deleteTodo={deleteCallback} />
  );
```

```
  const todoLink = TestUtils.findRenderedDOMComponentWithTag(result, 'a');
  TestUtils.Simulate.click(todoLink);
});
```

And with that all our `Todo` component tests are complete, and passing!

```
 TAP version 13
# Todo component
# rendering a not-done tweet
# It renders the text of the todo
ok 1 should be equal
# The todo does not have the done class
ok 2 should be equal
# rendering a done tweet
# The todo does have the done class
ok 3 (unnamed assert)
# toggling a TODO calls the given prop
ok 4 should be equal
# deleting a TODO calls the given prop
ok 5 should be equal

1..5
# tests 5
# pass  5

# ok
```

This might seem like a lot of effort, and a lot of new functionality to learn, but the good news is that most React component tests will follow roughly this structure. Once you gain familiarity with the React test utils they will have you covered. Now we've covered a lot of the basics we'll move at a little more speed through the rest of the tests.

END TO END TESTING:

http://googletesting.blogspot.co.uk/2015/04/just-say-no-to-more-end-to-end-tests.html

Just Say No to More End-to-End Tests
Wednesday, April 22, 2015

by Mike Wacker

At some point in your life, you can probably recall a movie that you and your friends all wanted to see, and that you and your friends all regretted watching afterwards. Or maybe you remember that time your team thought they'd found the next "killer feature" for their product, only to see that feature bomb after it was released.

Good ideas often fail in practice, and in the world of testing, one pervasive good idea that often fails in practice is a testing strategy built around end-to-end tests.

Testers can invest their time in writing many types of automated tests, including unit tests, integration tests, and end-to-end tests, but this strategy invests mostly in end-to-end tests that verify the product or service as a whole. Typically, these tests simulate real user scenarios.

End-to-End Tests in Theory

While relying primarily on end-to-end tests is a bad idea, one could certainly convince a reasonable person that the idea makes sense in theory.

To start, number one on Google's list of ten things we know to be true is: "Focus on the user and all else will follow." Thus, end-to-end tests that focus on real user scenarios sound like a great idea. Additionally, this strategy broadly appeals to many constituencies:

- Developers like it because it offloads most, if not all, of the testing to others.
- Managers and decision-makers like it because tests that simulate real user scenarios can help them easily determine how a failing test would impact the user.

- Testers like it because they often worry about missing a bug or writing a test that does not verify real-world behavior; writing tests from the user's perspective often avoids both problems and gives the tester a greater sense of accomplishment.

End-to-End Tests in Practice

So if this testing strategy sounds so good in theory, then where does it go wrong in practice? To demonstrate, I present the following composite sketch based on a collection of real experiences familiar to both myself and other testers. In this sketch, a team is building a service for editing documents online (e.g., Google Docs).

Let's assume the team already has some fantastic test infrastructure in place. Every night:

1. The latest version of the service is built.
2. This version is then deployed to the team's testing environment.
3. All end-to-end tests then run against this testing environment.
4. An email report summarizing the test results is sent to the team.

The deadline is approaching fast as our team codes new features for their next release. To maintain a high bar for product quality, they also require that at least 90% of their end-to-end tests pass before features are considered complete. Currently, that deadline is one day away:

Days Left	Pass %	Notes
1	5%	Everything is broken! Signing in to the service is broken. Almost all tests sign in a user, so almost all tests failed.
0	4%	A partner team we rely on deployed a bad build to their testing environment yesterday.

-1	54%	A dev broke the save scenario yesterday (or the day before?). Half the tests save a document at some point in time. Devs spent most of the day determining if it's a frontend bug or a backend bug.
-2	54%	It's a frontend bug, devs spent half of today figuring out where.
-3	54%	A bad fix was checked in yesterday. The mistake was pretty easy to spot, though, and a correct fix was checked in today.
-4	1%	Hardware failures occurred in the lab for our testing environment.
-5	84%	Many small bugs hiding behind the big bugs (e.g., sign-in broken, save broken). Still working on the small bugs.
-6	87%	We should be above 90%, but are not for some reason.
-7	89.54%	(Rounds up to 90%, close enough.) No fixes were checked in yesterday, so the tests must have been flaky yesterday.

Analysis

Despite numerous problems, the tests ultimately did catch real bugs.

What Went Well

- Customer-impacting bugs were identified and fixed before they reached the customer.

What Went Wrong

- The team completed their coding milestone a week late (and worked a lot of overtime).
- Finding the root cause for a failing end-to-end test is painful and can take a long time.
- Partner failures and lab failures ruined the test results on multiple days.
- Many smaller bugs were hidden behind bigger bugs.
- End-to-end tests were flaky at times.

- Developers had to wait until the following day to know if a fix worked or not.

So now that we know what went wrong with the end-to-end strategy, we need to change our approach to testing to avoid many of these problems. But what is the right approach?

The True Value of Tests

Typically, a tester's job ends once they have a failing test. A bug is filed, and then it's the developer's job to fix the bug. To identify where the end-to-end strategy breaks down, however, we need to think outside this box and approach the problem from first principles. If we "focus on the user (and all else will follow)," we have to ask ourselves how a failing test benefits the user. Here is the answer:

A failing test does not directly benefit the user.

While this statement seems shocking at first, it is true. If a product works, it works, whether a test says it works or not. If a product is broken, it is broken, whether a test says it is broken or not. So, if failing tests do not benefit the user, then what does benefit the user?

A bug fix directly benefits the user.

The user will only be happy when that unintended behavior - the bug - goes away. Obviously, to fix a bug, you must know the bug exists. To know the bug exists, ideally you have a test that catches the bug (because the user will find the bug if the test does not). But in that entire process, from failing test to bug

fix, value is only added at the very last step.

Stage	Failing Test	Bug Opened	Bug Fixed
Value Added	No	No	Yes

Thus, to evaluate any testing strategy, you cannot just evaluate how it finds bugs. You also must evaluate how it enables developers to fix (and even prevent) bugs.

Building the Right Feedback Loop

Tests create a feedback loop that informs the developer whether the product is working or not. The ideal feedback loop has several properties:

- It's fast. No developer wants to wait hours or days to find out if their change works. Sometimes the change does not work - nobody is perfect - and the feedback loop needs to run multiple times. A faster feedback loop leads to faster fixes. If the loop is fast enough, developers may even run tests before checking in a change.
- It's reliable. No developer wants to spend hours debugging a test, only to find out it was a flaky test. Flaky tests reduce the developer's trust in the test, and as a result flaky tests are often ignored, even when they find real product issues.
- It isolates failures. To fix a bug, developers need to find the specific lines of code causing the bug. When a product contains millions of lines of codes, and the bug could be anywhere, it's like trying to find a needle in a haystack.

Think Smaller, Not Larger

So how do we create that ideal feedback loop? By thinking smaller, not larger.

Unit Tests

Unit tests take a small piece of the product and test that piece in isolation. They tend to create that ideal feedback loop:

- Unit tests are fast. We only need to build a small unit to test it, and the tests also tend to be rather small. In fact, one tenth of a second is considered slow for unit tests.
- Unit tests are reliable. Simple systems and small units in general tend to suffer much less from flakiness. Furthermore, best practices for unit testing - in particular practices related to hermetic tests - will remove flakiness entirely.
- Unit tests isolate failures. Even if a product contains millions of lines of code, if a unit test fails, you only need to search that small unit under test to find the bug.

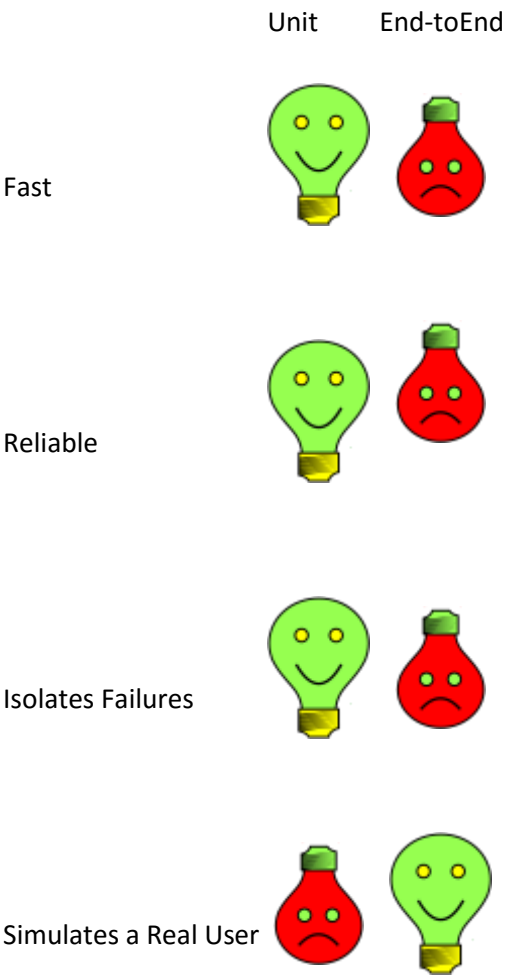
Writing effective unit tests requires skills in areas such as dependency management, mocking, and hermetic testing. I won't cover these skills here, but as a start, the typical example offered to new Googlers (or Nooglers) is how Google builds and tests a stopwatch.

Unit Tests vs. End-to-End Tests

With end-to-end tests, you have to wait: first for the entire product to be built, then for it to be deployed, and finally for all end-to-end tests to run. When the tests do run, flaky tests tend to be a fact of life. And even if a test finds a bug, that bug could be anywhere in the product.

Although end-to-end tests do a better job of simulating real user scenarios, this advantage quickly becomes outweighed by all the disadvantages of the

end-to-end feedback loop:



Integration Tests

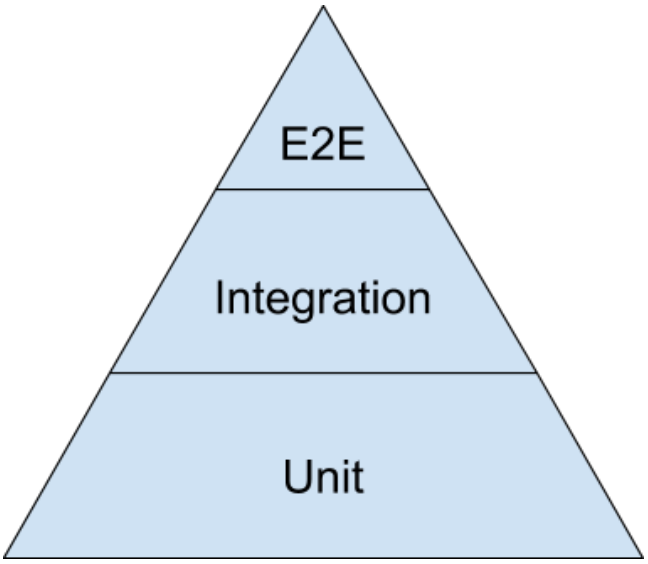
Unit tests do have one major disadvantage: even if the units work well in isolation, you do not know if they work well together. But even then, you do not necessarily need end-to-end tests. For that, you can use an integration test. An integration test takes a small group of units, often two units, and tests their behavior as a whole, verifying that they coherently work together.

If two units do not integrate properly, why write an end-to-end test when you

can write a much smaller, more focused integration test that will detect the same bug? While you do need to think larger, you only need to think a little larger to verify that units work together.

Testing Pyramid

Even with both unit tests and integration tests, you probably still will want a small number of end-to-end tests to verify the system as a whole. To find the right balance between all three test types, the best visual aid to use is the testing pyramid. Here is a simplified version of the testing pyramid from the opening keynote of the 2014 Google Test Automation Conference:



The bulk of your tests are unit tests at the bottom of the pyramid. As you move up the pyramid, your tests gets larger, but at the same time the number of tests (the width of your pyramid) gets smaller.

As a good first guess, Google often suggests a 70/20/10 split: 70% unit tests, 20% integration tests, and 10% end-to-end tests. The exact mix will be

different for each team, but in general, it should retain that pyramid shape. Try to avoid these anti-patterns:

- Inverted pyramid/ice cream cone. The team relies primarily on end-to-end tests, using few integration tests and even fewer unit tests.
- Hourglass. The team starts with a lot of unit tests, then uses end-to-end tests where integration tests should be used. The hourglass has many unit tests at the bottom and many end-to-end tests at the top, but few integration tests in the middle.

Just like a regular pyramid tends to be the most stable structure in real life, the testing pyramid also tends to be the most stable testing strategy.