# 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 practices and common usages for Sinon's different features.

## 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 (http://sinonjs.org/docs/#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 (http://sinonjs.org/docs/#stubs) 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 (http://sinonjs.org/docs/#sinon-config).

## 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.