Sinon.JS

Standalone test spies, stubs and mocks for JavaScript.
No dependencies, works with any unit testing framework.

This page contains the entire Sinon.JS API documentation along with brief introductions to the concepts Sinon implements. Please ask questions on the mailing list if you're stuck. I also really appreciate suggestions to improve the documentation so Sinon.JS is easier to work with. Get in touch!

Test spies

API reference

sinon.spy() spy API spy call API

What is a test spy?

A test spy is a function that records arguments, return value, the value of `this` and exception thrown (if any) for all its calls. A test spy can be an anonymous function or it can wrap an existing function.

When to use spies?

Test spies are useful to test both callbacks and how certain functions/methods are used throughout the system under test. The following simplified example shows how to use spies to test how a function handles a callback:

```
"test should call subscribers on publish": function () {
    var callback = sinon.spy();
    PubSub.subscribe("message", callback);

    PubSub.publishSync("message");

    assertTrue(callback.called);
}
```

Spying on existing methods

`sinon.spy` can also spy on existing functions. When doing so, the original function will behave just as normal (including when used as a constructor) but you will have access to data about all calls. The following is a slightly contrived example:

```
{
    setUp: function () {
        sinon.spy(jQuery, "ajax");
    },

    tearDown: function () {
        jQuery.ajax.restore(); // Unwraps the spy
    },

    "test should inspect jQuery.getJSON's usage of jQuery.ajax": function () {
        jQuery.getJSON("/some/resource");

        assert(jQuery.ajax.calledOnce);
        assertEquals("/some/resource", jQuery.ajax.getCall(0).args[0].url);
        assertEquals("json", jQuery.ajax.getCall(0).args[0].dataType);
    }
}
```

Creating spies: `sinon.spy()`

`var spy = sinon.spy();`

Creates an anonymous function that records arguments, `this` value, exceptions and return values for all calls.

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

Spies on the provided function

`var spy = sinon.spy(object, "method");`

Creates a spy for `object.method` and replaces the original method with the spy. The spy acts exactly like the original method in all cases. The original method can be restored by calling `object.method.restore()`. The returned spy is the function object which replaced the original method. `spy === object.method`.

Spy API

Spies provide a rich interface to inspect their usage. The above examples showed the `calledOnce` boolean property as well as the `getCall` method and the returned object's `args` property. There are three ways of inspecting call data.

The preferred approach is to use the spy's `calledWith` method (and friends) because it keeps your test from being too specific about which call did what and so on. It will return `true` if the spy was ever called with the provided arguments.

```
"test should call subscribers with message as first argument" : function () {
    var message = 'an example message';
    var spy = sinon.spy();

    PubSub.subscribe(message, spy);
    PubSub.publishSync(message, "some payload");

    assert(spy.calledWith(message));
}
```

If you want to be specific, you can directly check the first argument of the first call. There are two ways of achieving this:

```
"test should call subscribers with message as first argument" : function () {
    var message = 'an example message';
    var spy = sinon.spy();

    PubSub.subscribe(message, spy);
    PubSub.publishSync(message, "some payload");

    assertEquals(message, spy.args[0][0]);
}
```

```
"test should call subscribers with message as first argument" : function () {
    var message = 'an example message';
    var spy = sinon.spy();

    PubSub.subscribe(message, spy);
    PubSub.publishSync(message, "some payload");

    assertEquals(message, spy.getCall(0).args[0]);
}
```

The first example uses the two-dimensional `args` array directly on the spy, while the second example fetches the first call object and then accesses it's `args` array. Which one to use is a matter of preference, but the recommended approach is going with `spy.calledWith(arg1, arg2, ...)` unless there's a need to make the tests highly specific.

Spy API

Spy objects are objects returned from `sinon.spy()`. When spying on existing methods with `sinon.spy(object, method)`, the following properties and methods are also available on `object.method`.

spy.withArgs(arg1[, arg2, ...]);

Creates a spy that only records calls when the received arguments match those passed to `withArgs`. This is useful to be more expressive in your assertions, where you can access the spy with the same call.

```
"should call method once with each argument": function () {
    var object = { method: function () {} };
    var spy = sinon.spy(object, "method");
    spy.withArgs(42);
    spy.withArgs(1);

    object.method(42);
    object.method(1);

    assert(spy.withArgs(42).calledOnce);
    assert(spy.withArgs(1).calledOnce);
}
```

spy.callCount

The number of recorded calls.

spy.called

`true` if the spy was called at least once

spy.calledOnce

`true` if spy was called exactly once

spy.calledTwice

`true` if the spy was called exactly twice

spy.calledThrice

`true` if the spy was called exactly thrice

spy.firstCall

The first call

spy.secondCall

The second call

spy.thirdCall

The third call

spy.lastCall

The last call

spy.calledBefore(anotherSpy);

Returns `true` if the spy was called before `anotherSpy`

spy.calledAfter(anotherSpy);

Returns `true` if the spy was called after `anotherSpy`

spy.calledOn(obj);

Returns `true` if the spy was called at least once with `obj` as `this`.

spy.alwaysCalledOn(obj);

Returns `true` if the spy was always called with `obj` as `this`.

spy.calledWith(arg1, arg2, ...);

Returns `true` if spy was called at least once with the provided arguments. Can be used for partial matching, Sinon only checks the provided arguments against actual arguments, so a call that received the provided arguments (in the same spots) and possibly others as well will return `true`.

spy.alwaysCalledWith(arg1, arg2, ...);

Returns `true` if spy was always called with the provided arguments (and possibly others).

spy.calledWithExactly(arg1, arg2, ...);

Returns `true` if spy was called at least once with the provided arguments and no others.

spy.alwaysCalledWithExactly(arg1, arg2, ...);

Returns `true` if spy was always called with the exact provided arguments.

spy.calledWithMatch(arg1, arg2, ...);

Returns `true` if spy was called with matching arguments (and possibly others). This behaves the same as `spy.calledWith(sinon.match(arg1), sinon.match(arg2), ...)`.

spy.alwaysCalledWithMatch(arg1, arg2, ...);

Returns `true` if spy was always called with matching arguments (and possibly others). This behaves the same as `spy.alwaysCalledWith(sinon.match(arg1), sinon.match(arg2), ...)`.

spy.calledWithNew();

Returns `true` if spy/stub was called the `new` operator. Beware that this is inferred based on the value of the `this` object and the spy function's `prototype`, so it may give false positives if you actively return the right kind of object.

spy.neverCalledWith(arg1, arg2, ...);

Returns `true` if the spy/stub was never called with the provided arguments.

spy.neverCalledWithMatch(arg1, arg2, ...);

Returns `true` if the spy/stub was never called with matching arguments. This behaves the same as `spy.neverCalledWith(sinon.match(arg1), sinon.match(arg2), ...)`.

spy.threw();

Returns `true` if spy threw an exception at least once.

spy.threw("TypeError");

Returns `true` if spy threw an exception of the provided type at least once.

spy.threw(obj);

Returns `true` if spy threw the provided exception object at least once.

spy.alwaysThrew();

Returns `true` if spy always threw an exception.

spy.alwaysThrew("TypeError");

Returns `true` if spy always threw an exception of the provided type.

`spy.alwaysThrew(obj);`

Returns `true` if spy always threw the provided exception object.

`spy.returned(obj);`

Returns `true` if spy returned the provided value at least once. Uses deep comparison for objects and arrays. Use `spy.returned(sinon.match.same(obj))` for strict comparison (see matchers).

`spy.alwaysReturned(obj);`

Returns `true` if spy always returned the provided value.

`var spyCall = spy.getCall(n);`

Returns the *nth* [call](#spycall). Accessing individual calls helps with more detailed behavior verification when the spy is called more than once. Example:

```
sinon.spy(jQuery, "ajax");
jQuery.ajax("/stuffs");
var spyCall = jQuery.ajax.getCall(0);

assertEquals("/stuffs", spyCall.args[0]);
```

`spy.thisValues`

Array of `this` objects, `spy.thisValues[0]` is the `this` object for the first call.

`spy.args`

Array of arguments received, `spy.args[0]` is an array of arguments recevied in the first call.

`spy.exceptions`

Array of exception objects thrown, `spy.exceptions[0]` is the exception thrown by the first call. If the call did not throw an error, the value at the call's location in `.exceptions` will be 'undefined'.

`spy.returnValues`

Array of return values, `spy.returnValues[0]` is the return value of the first call. If the call did not explicitly return a value, the value at the call's location in `.returnValues` will be 'undefined'.

`spy.reset()`

Resets the state of a spy.

`spy.printf(format string", [arg1, arg2, ...])`

Returns the passed format string with the following replacements performed:

- %n: the name of the spy ("spy" by default)
- %c: the number of times the spy was called, in words ("once", "twice", etc.)
- %C: a list of string representations of the calls to the spy, with each call prefixed by a newline and four spaces
- %t: a comma-delimited list of `this` values the spy was called on
- %<var>n</var>: the formatted value of the *n*th argument passed to `printf`
- %*: a comma-delimited list of the (non-format string) arguments passed to `printf`

Individual spy calls

var spyCall = spy.getCall(n)

Returns the *nth* [call](#spycall). Accessing individual calls helps with more detailed behavior verification when the spy is called more than once. Example:

```
sinon.spy(jQuery, "ajax");
jQuery.ajax("/stuffs");
var spyCall = jQuery.ajax.getCall(0);

assertEquals("/stuffs", spyCall.args[0]);
```

spyCall.calledOn(obj);

Returns `true` if `obj` was `this` for this call.

spyCall.calledWith(arg1, arg2, ...);

Returns `true` if call received provided arguments (and possibly others).

spyCall.calledWithExactly(arg1, arg2, ...);

Returns `true` if call received provided arguments and no others.

spyCall.calledWithMatch(arg1, arg2, ...);

Returns `true` if call received matching arguments (and possibly others). This behaves the same as `spyCall.calledWith(sinon.match(arg1), sinon.match(arg2), ...)`.

spyCall.notCalledWith(arg1, arg2, ...);

Returns `true` if call did not receive provided arguments.

spyCall.notCalledWithMatch(arg1, arg2, ...);

Returns `true` if call did not receive matching arguments. This behaves the same as `spyCall.notCalledWith(sinon.match(arg1), sinon.match(arg2), ...)`.

spyCall.threw();

Returns `true` if call threw an exception.

spyCall.threw(TypeError");`

Returns `true` if call threw exception of provided type.

spyCall.threw(obj);

Returns `true` if call threw provided exception object.

spyCall.thisValue

The call's `this` value.

spyCall.args

Array of received arguments.

spyCall.exception

Exception thrown, if any.

spyCall.returnValue

Return value.

Test stubs

API reference

What are stubs?

Test stubs are functions (spies) with pre-programmed behavior. They support the full test spy API in addition to methods which can be used to alter the stub's behavior.

As spies, stubs can be either anonymous, or wrap existing functions. When wrapping an existing function with a stub, the original function is not called.

When to use stubs?

Use a stub when you want to:

1. Control a method's behavior from a test to force the code down a specific path. Examples include forcing a method to throw an error in order to test error handling.
2. When you want to prevent a specific method from being called directly (possibly because it triggers undesired behavior, such as a XMLHttpRequest or similar).

The following example is yet another test from Morgan Roderick's PubSubJS which shows how to create an anonymous stub that throws an exception when called.

```
"test should call all subscribers, even if there are exceptions" : function(){
    var message = 'an example message';
    var error = 'an example error message';
    var stub = sinon.stub().throws();
    var spy1 = sinon.spy();
    var spy2 = sinon.spy();

    PubSub.subscribe(message, stub);
    PubSub.subscribe(message, spy1);
    PubSub.subscribe(message, spy2);

    PubSub.publishSync(message, undefined);

    assert(spy1.called);
    assert(spy2.called);
    assert(stub.calledBefore(spy1));
}
```

Note how the stub also implements the spy interface. The test verifies that all callbacks where called, and also that the exception throwing stub was called before one of the other callbacks.

Defining stub behavior on consecutive calls

Calling behavior defining methods like `returns` or `throws` multiple times overrides the behavior of the stub. As of Sinon version 1.8, you can use the `onCall` method to make a stub respond differently on consecutive calls.

Note that in Sinon version 1.5 to version 1.7, multiple calls to the `yields*` and `callsArg*` family of methods define a sequence of behaviors for consecutive calls. As of 1.8, this functionality has been removed in favor of the `onCall` API.

Stub API

`var stub = sinon.stub();`

Creates an anonymous stub function.

`var stub = sinon.stub(object, "method");`

Replaces `object.method` with a stub function. The original function can be restored by calling `object.method.restore();` (or `stub.restore();`). An exception is thrown if the property is not already a function, to help avoid typos when stubbing methods.

`var stub = sinon.stub(object, "method", func);`

Replaces `object.method` with a `func`, wrapped in a `spy`. As usual, `object.method.restore();` can be used to restore the original method.

`var stub = sinon.stub(obj);`

Stubs all the object's methods. Note that it's usually better practice to stub individual methods, particularly on objects that you don't understand or control all the methods for (e.g. library dependencies). Stubbing individual methods tests intent more precisely and is less susceptible to unexpected behavior as the object's code evolves.
If you want to create a stub object of `MyConstructor`, but don't want the constructor to be invoked, use this utility function.

```
var stub = sinon.createStubInstance(MyConstructor)
```

`stub.withArgs(arg1[, arg2, ...]);`

Stubs the method only for the provided arguments. This is useful to be more expressive in your assertions, where you can access the spy with the same call. It is also useful to create a stub that can act differently in response to different arguments.

```
"test should stub method differently based on arguments": function () {
    var callback = sinon.stub();
    callback.withArgs(42).returns(1);
    callback.withArgs(1).throws("TypeError");

    callback(); // No return value, no exception
    callback(42); // Returns 1
    callback(1); // Throws TypeError
}
```

`stub.onCall(n);`

Added in Sinon.JS 1.8

Defines the behavior of the stub on the *nth* call. Useful for testing sequential interactions.

```
"test should stub method differently on consecutive calls": function () {
    var callback = sinon.stub();
    callback.onCall(0).returns(1);
    callback.onCall(1).returns(2);
    callback.returns(3);

    callback(); // Returns 1
    callback(); // Returns 2
    callback(); // All following calls return 3
}
```

There are methods `onFirstCall`, `onSecondCall`,`onThirdCall` to make stub definitions read more naturally.

```
                    `onCall` can be combined with all of the behavior defining methods in this section.  In particular, it can be used togethe
```

```
"test should stub method differently on consecutive calls with certain argument": function () {
    var callback = sinon.stub();
    callback.withArgs(42)
        .onFirstCall().returns(1)
        .onSecondCall().returns(2);
```

```
    callback.returns(0);

    callback(1); // Returns 0
    callback(42); // Returns 1
    callback(1); // Returns 0
    callback(42); // Returns 2
    callback(1); // Returns 0
    callback(42); // Returns 0
}
```

Note how the behavior of the stub for argument 42 falls back to the default behavior once no more calls have been defined.

`stub.onFirstCall();`

Alias for `stub.onCall(0);`

`stub.onSecondCall();`

Alias for `stub.onCall(1);`

`stub.onThirdCall();`

Alias for `stub.onCall(2);`

`stub.returns(obj);`

Makes the stub return the provided value.

`stub.returnsArg(index);`

Causes the stub to return the argument at the provided index. `stub.returnsArg(0);` causes the stub to return the first argument.

`stub.returnsThis();`

Causes the stub to return its `this` value. Useful for stubbing jQuery-style fluent APIs.

`stub.throws();`

Causes the stub to throw an exception (`Error`).

`stub.throws("TypeError");`

Causes the stub to throw an exception of the provided type.

`stub.throws(obj);`

Causes the stub to throw the provided exception object.

`stub.callsArg(index);`

Causes the stub to call the argument at the provided index as a callback function. `stub.callsArg(0);` causes the stub to call the first argument as a callback.

`stub.callsArgOn(index, context);`

Like above but with an additional parameter to pass the `this` context.

`stub.callsArgWith(index, arg1, arg2, ...);`

Like `callsArg`, but with arguments to pass to the callback.

`stub.callsArgOnWith(index, context, arg1, arg2, ...);`

Like above but with an additional parameter to pass the `this` context.

stub.yields([arg1, arg2, ...])

Almost like `callsArg`. Causes the stub to call the first callback it receives with the provided arguments (if any). If a method accepts more than one callback, you need to use `callsArg` to have the stub invoke other callbacks than the first one.

stub.yieldsOn(context, [arg1, arg2, ...])

Like above but with an additional parameter to pass the `this` context.

stub.yieldsTo(property, [arg1, arg2, ...])

Causes the spy to invoke a callback passed as a property of an object to the spy. Like `yields`, `yieldsTo` grabs the first matching argument, finds the callback and calls it with the (optional) arguments.

stub.yieldsToOn(property, context, [arg1, arg2, ...])

Like above but with an additional parameter to pass the `this` context.

```
"test should fake successful ajax request": function () {
    sinon.stub(jQuery, "ajax").yieldsTo("success", [1, 2, 3]);

    jQuery.ajax({
        success: function (data) {
            assertEquals([1, 2, 3], data);
        }
    });
}
```

spy.yield([arg1, arg2, ...])

Invoke callbacks passed to the `spy` with the given arguments. If the spy was never called with a function argument, `yield` throws an error. Also aliased as `invokeCallback`.

spy.yieldTo(callback, [arg1, arg2, ...])

Invokes callbacks passed as a property of an object to the spy. Like `yield`, `yieldTo` grabs the first matching argument, finds the callback and calls it with the (optional) arguments.

```
"calling callbacks": function () {
    var callback = sinon.stub();
    callback({
        "success": function () {
            console.log("Success!");
        },
        "failure": function () {
            console.log("Oh noes!");
        }
    });

    callback.yieldTo("failure"); // Logs "Oh noes!"
}
```

spy.callArg(argNum)

Like `yield`, but with an explicit argument number specifying which callback to call. Useful if a function is called with more than one callback, and simply calling the first callback is not desired.

```
"calling the last callback": function () {
    var callback = sinon.stub();
    callback(function () {
        console.log("Success!");
    }, function () {
        console.log("Oh noes!");
    });
```

```
        callback.callArg(1); // Logs "Oh noes!"
    }
```

spy.callArgWith(argNum, [arg1, arg2, ...])

```
    Like `callArg`, but with arguments.
```

stub.callsArgAsync(index);

Same as their corresponding non-Async counterparts, but with callback being deferred (executed not immediately but after short timeout and in another “thread”)

stub.callsArgAsync(index);

stub.callsArgOnAsync(index, context);

stub.callsArgWithAsync(index, arg1, arg2, ...);

stub.callsArgOnWithAsync(index, context, arg1, arg2, ...);

stub.yieldsAsync([arg1, arg2, ...])

stub.yieldsOnAsync(context, [arg1, arg2, ...])

stub.yieldsToAsync(property, [arg1, arg2, ...])

stub.yieldsToOnAsync(property, context, [arg1, arg2, ...])

Same as their corresponding non-Async counterparts, but with callback being deferred (executed not immediately but after short timeout and in another “thread”)

Mocks

API reference

What are mocks?

Mocks (and mock expectations) are fake methods (like spies) with pre-programmed behavior (like stubs) as well as *pre-programmed expectations*. A mock will fail your test if it is not used as expected.

When to use mocks?

Mocks should only be used for the *method under test*. In every unit test, there should be one unit under test. If you want to control how your unit is being used and like stating expectations upfront (as opposed to asserting after the fact), use a mock.

When to **not** use mocks?

Mocks come with built-in expectations that may fail your test. Thus, they enforce implementation details. The rule of thumb is: if you wouldn’t add an assertion for some specific call, don’t mock it. Use a stub instead. In general you should never have more than **one** mock (possibly with several expectations) in a single test.

Expectations implement both the spies and stubs APIs.

To see how mocks look like in Sinon.JS, here’s one of the PubSubJS tests again, this time using a method as callback and using mocks to verify its behavior:

```
"test should call all subscribers when exceptions": function () {
    var myAPI = { method: function () {} };

    var spy = sinon.spy();
    var mock = sinon.mock(myAPI);
    mock.expects("method").once().throws();

    PubSub.subscribe("message", myAPI.method);
    PubSub.subscribe("message", spy);
    PubSub.publishSync("message", undefined);

    mock.verify();
    assert(spy.calledOnce);
}
```

Mocks API

`var mock = sinon.mock(obj);`

Creates a mock for the provided object. Does not change the object, but returns a mock object to set expectations on the object's methods.

`var expectation = mock.expects("method");`

Overrides `obj.method` with a mock function and returns it. See expectations below.

`mock.restore();`

Restores all mocked methods.

`mock.verify();`

Verifies all expectations on the mock. If any expectation is not satisfied, an exception is thrown. Also restores the mocked methods.

Expectations

All the expectation methods return the expectation, meaning you can chain them. Typical usage:

```
sinon.mock(jQuery).expects("ajax").atLeast(2).atMost(5);
jQuery.ajax.verify();
```

`var expectation = sinon.expectation.create([methodName]);`

Creates an expectation without a mock object, basically an anonymous mock function. Method name is optional and is used in exception messages to make them more readable.

`var expectation = sinon.mock();`

The same as the above.

`expectation.atLeast(number);`

Specify the minimum amount of calls expected.

`expectation.atMost(number);`

Specify the maximum amount of calls expected.

`expectation.never();`

Expect the method to never be called.

`expectation.once();`

Expect the method to be called exactly once.

`expectation.twice();`

Expect the method to be called exactly twice.

`expectation.thrice();`

Expect the method to be called exactly thrice.

`expectation.exactly(number);`

Expect the method to be called exactly `number` times.

`expectation.withArgs(arg1, arg2, ...);`

Expect the method to be called with the provided arguments and possibly others.

`expectation.withExactArgs(arg1, arg2, ...);`

Expect the method to be called with the provided arguments and no others.

`expectation.on(obj);`

Expect the method to be called with `obj` as `this`.

`expectation.verify();`

Verifies the expectation and throws an exception if it's not met.

Fake timers

API reference

Fake timers is a synchronous implementation of `setTimeout` and friends that Sinon.JS can overwrite the global functions with to allow you to more easily test code using them. Fake timers provide a `clock` object to pass time, which can also be used to control `Date` objects created through either `new Date();` or `Date.now();` (if supported by the browser).

When faking timers with IE you also need sinon-ie-1.12.1, which should be loaded after sinon-1.12.1.js.

The fake timers can be used completely stand-alone by downloading sinon-timers-1.12.1 When using the fake timers in IE you also need sinon-timers-ie-1.12.1. Load it after the first file.

```
{
    setUp: function () {
        this.clock = sinon.useFakeTimers();
    },

    tearDown: function () {
        this.clock.restore();
    },

    "test should animate element over 500ms" : function(){
        var el = jQuery("<div></div>");
        el.appendTo(document.body);

        el.animate({ height: "200px", width: "200px" });
        this.clock.tick(510);

        assertEquals("200px", el.css("height"));
```

```
        assertEquals("200px", el.css("width"));
    }
}
```

Fake timers API

`var clock = sinon.useFakeTimers();`

Causes Sinon to replace the global `setTimeout`, `clearTimeout`, `setInterval`, `clearInterval` and `Date` with a custom implementation which is bound to the returned `clock` object. Starts the clock at the UNIX epoch (timestamp of 0)

`var clock = sinon.useFakeTimers(now);`

As above, but rather than starting the clock with a timestamp of 0, start at the provided timestamp.

`var clock = sinon.useFakeTimers([now, ]prop1, prop2, ...);`

Sets the clock start timestamp and names functions to fake. Possible functions are setTimeout, clearTimeout, setInterval, clearInterval, and Date. Can also be called without the timestamp.

`clock.tick(ms);`

Tick the clock ahead `ms` milliseconds. Causes all timers scheduled within the affected time range to be called.

`clock.restore();`

Restore the faked methods. Call in e.g. `tearDown`.

Fake `XMLHttpRequest`

API reference

sinon.useFakeXMLHttpRequest FakeXMLHttpRequest Simulating responses Fake server

Provides a fake implementation of `XMLHttpRequest` and provides several interfaces for manipulating objects created by it. Also fakes the native `XMLHttpRequest` and `ActiveXObject` (if available, and only for `XMLHTTP` progids). Helps with testing requests made with XHR.

When faking XHR with IE you also need sinon-ie-1.12.1, which should be loaded after sinon-1.12.1.js.

The fake server and XHR can be used completely stand-alone by downloading sinon-server-1.12.1. When using the fake server in IE you also need sinon-ie-1.12.1. Load it after the first file.

```
{
    setUp: function () {
        this.xhr = sinon.useFakeXMLHttpRequest();
        var requests = this.requests = [];

        this.xhr.onCreate = function (xhr) {
            requests.push(xhr);
        };
    },

    tearDown: function () {
        this.xhr.restore();
    },

    "test should fetch comments from server" : function () {
        var callback = sinon.spy();
        myLib.getCommentsFor("/some/article", callback);
        assertEquals(1, this.requests.length);

        this.requests[0].respond(200, { "Content-Type": "application/json" },
```

```
                                '[{ "id": 12, "comment": "Hey there" }]');
            assert(callback.calledWith([{ id: 12, comment: "Hey there" }]));
        }
    }
```

sinon.useFakeXMLHttpRequest

var xhr = sinon.useFakeXMLHttpRequest();

Causes Sinon to replace the native `XMLHttpRequest` object in browsers that support it with a custom implementation which does not send actual requests. In browsers that support `ActiveXObject`, this constructor is replaced, and fake objects are returned for `XMLHTTP` progIds. Other progIds, such as `XMLDOM` are left untouched.

xhr.onCreate = function (xhr) {};

By assigning a function to the `onCreate` property of the returned object from `useFakeXMLHttpRequest()` you can subscribe to newly created `FakeXMLHttpRequest` objects. See below for the fake xhr object API. Using this observer means you can still reach objects created by e.g. `jQuery.ajax` (or other abstractions/frameworks).

xhr.restore();

Restore original function(s).

FakeXMLHttpRequest

String request.url

The URL set on the request object.

String request.method

The request method as a string.

Object request.requestHeaders

An object of all request headers, i.e.:

```
{
    "Accept": "text/html, */*",
    "Connection": "keep-alive"
}
```

String request.requestBody

The request body

int request.status

The request's status code. Undefined if the request has not been handled (see respond below).

String request.statusText

Only populated if the respond method is called (see below).

boolean request.async

Whether or not the request is asynchronous.

`String request.username`

Username, if any.

`String request.password`

Password, if any.

`Document request.responseXML`

When using `respond`, this property is populated with a parsed document if response headers indicate as much (see the spec).

`String request.getResponseHeader(header);`

The value of the given response header, if the request has been responded to (see `respond`).

`Object request.getAllResponseHeaders();`

All response headers as an object.

Filtered requests

When using Sinon.JS for mockups or partial integration/functional testing, you might want to fake some requests, while allowing others to go throught to the backend server. With filtered `FakeXMLHttpRequest`s (new in Sinon 1.3.0), you can.

`FakeXMLHttpRequest.useFilters`

Default `false`. When set to `true`, Sinon will check added filters if certain requests should be “unfaked”.

`FakeXMLHttpRequest.addFilter(fn)`

Add a filter that will decide whether or not to fake a request. The filter will be called when `xhr.open` is called, with the exact same arguments (`method`, `url`, `async`, `username`, `password`). If the filter returns true, the request will not be faked.

Simulating server responses

`request.setResponseHeaders(object);`

```
Sets response headers (e.g. `{ "Content-Type": "text/html", /* ... */ }`,
  updates the `readyState` property and fires `onreadystatechange`.
```

`request.setResponseBody(body);`

Sets the respond body, updates the `readyState` property and fires `onreadystatechange`. Additionally, populates `responseXML` with a parsed document if response headers indicate as much.

`var server = sinon.fakeServer.create();`

Creates a new server. This function also calls `sinon.useFakeXMLHttpRequest()`.

`request.respond(status, headers, body);`

Calls the above two methods and sets the `status` and `statusText` properties. Status should be a number,

the status text is looked up from `sinon.FakeXMLHttpRequest.statusCodes`.

`Boolean request.autoRespond`

When set to `true`, causes the server to automatically respond to incoming requests after a timeout. The default timeout is 10ms but you can control it through the `autoRespondAfter` property. Note that this feature is intended to help during mockup development, and is not suitable for use in tests.

`Number request.autoRespondAfter`

When `autoRespond` is `true`, respond to requests after this number of milliseconds. Default is 10.

Fake server

High-level API to manipulate `FakeXMLHttpRequest` instances. For help with handling JSON-P please refer to our [notes below](#json-p)

```
{
    setUp: function () {
        this.server = sinon.fakeServer.create();
    },

    tearDown: function () {
        this.server.restore();
    },

    "test should fetch comments from server" : function () {
        this.server.respondWith("GET", "/some/article/comments.json",
            [200, { "Content-Type": "application/json" },
             '[{ "id": 12, "comment": "Hey there" }]']);

        var callback = sinon.spy();
        myLib.getCommentsFor("/some/article", callback);
        this.server.respond();

        sinon.assert.calledWith(callback, [{ id: 12, comment: "Hey there" }]));
    }
}
```

`var server = sinon.fakeServer.create();`

Creates a new server. This function also calls `sinon.useFakeXMLHttpRequest()`.

`var server = sinon.fakeServerWithClock.create();`

Creates a server that also manages fake timers. This is useful when testing XHR objects created with e.g. jQuery 1.3.x, which uses a timer to poll the object for completion, rather than the usual `onreadystatechange`.

`server.respondWith(response);`

`response` can be on of three things:

1. A string representing the response body
2. An array with status, headers and response body, e.g. `[200, { "Content-Type": "text/html", "Content-Length": 2 }, "OK"]`
3. A function.

Default status is 200 and default headers are none. Causes the server to respond to any request not matched by another response with the provided data. The default catch-all response is `[404, {}, ""]`.

When the response is a function, it will be passed the request object. You must manually call respond on it to complete the request.

`server.respondWith(url, response);`

Responds to all requests to given URL, e.g. `/posts/1`.

`server.respondWith(method, url, response);`

Responds to all `method` requests to the given URL with the given response. `method` is an HTTP verb.

`server.respondWith(urlRegExp, response);`

URL may be a regular expression, e.g. `/\/post\//\d+` If the response is a function, it will be passed any capture groups from the regular expression along with the XMLHttpRequest object:

```
server.respondWith(/\/todo-items\/(\d+)/, function (xhr, id) {  xhr.respond(200, { ?Content-Type?: ?application/json? }, ?[{ ?id?: ? + id + ? }]
```

`server.respondWith(method, urlRegExp, response);`

Responds to all `method` requests to URLs matching the regular expression.

`server.respond();`

Causes all queued asynchronous requests to receive a response. If none of the responses added through `respondWith` match, the default response is `[404, {}, ""]`. Synchronous requests are responded to immediately, so make sure to call `respondWith` upfront. If called with arguments, `respondWith` will be called with those arguments before responding to requests.

`server.autoRespond = true;`

If set, will call to `srv.respondWith` automatically after every request.

`server.autoRespondAfter = ms;`

Causes the server to automatically respond to incoming requests after a timeout.

`Boolean `server.fakeHTTPMethods``

If set to `true`, server will find `_method` parameter in POST body and recognize that as the actual method. Supports a pattern common to Ruby on Rails applications. For custom HTTP method faking, override `server.getHTTPMethod(request)`.

`server.getHTTPMethod(request)`

Used internally to determine the HTTP method used with the provided request. By default this method simply returns `request.method`. When `server.fakeHTTPMethods` is true, the method will return the value of the `_method` parameter if the method is “POST”. This method can be overrided to provide custom behavior.

`server.restore();`

Restores the native XHR constructor.

JSON-P

[API reference]

JSON-P doesn’t use Ajax requests, which is what the fake server is concerned with. A JSON-P request actually creates a script element and inserts it into the document. There is no sensible/unobtrusive enough way to fake this automatically. Your best option is to simply stub jQuery in this case:

```
sinon.stub(jQuery, "ajax");
sinon.assert.calledOnce(jQuery.ajax);
```

Potentially we could have had the fake server detect jQuery and fake any calls to jQuery.ajax when JSON-P is used, but that felt like a compromise in the focus of the Sinon project compared to simply documenting the above work-around.

Assertions

API reference

Sinon.JS ships with a set of assertions that mirror most behavior verification methods and properties on spies and stubs. The advantage of using the assertions is that failed expectations on stubs and spies can be expressed directly as assertion failures with detailed and helpful error messages.

To make sure assertions integrate nicely with your test framework, you should customize either `sinon.assert.fail` or `sinon.assert.failException` and look into `sinon.assert.expose` and `sinon.assert.pass`.

The assertions can be used with either spies or stubs.

```
"test should call subscribers with message as first argument" : function () {
    var message = "an example message";
    var spy = sinon.spy();

    PubSub.subscribe(message, spy);
    PubSub.publishSync(message, "some payload");

    sinon.assert.calledOnce(spy);
    sinon.assert.calledWith(spy, message);
}
```

Assertions API

sinon.assert.fail(message)

Every assertion fails by calling this method. By default it throws an error of type `sinon.assert.failException`. If your testing framework looks for assertion errors by checking for a specific exception, you can simply override the kind of exception thrown. If that does not fit with your testing framework of choice, override the `fail` method to do the right thing.

sinon.assert.failException

Defaults to `"AssertError"`.

sinon.assert.pass(assertion)

Called every time an assertion passes. Default implementation does nothing.

sinon.assert.notCalled(spy)

Passes if spy was never called.

sinon.assert.called(spy)

Passes if spy was called at least once.

sinon.assert.calledOnce(spy)

Passes if spy was called once and only once.

sinon.assert.calledTwice()

Passes if spy was called exactly twice.

sinon.assert.calledThrice()

Passes if spy was called exactly three times.

sinon.assert.callCount(spy, num)

Passes if the spy was called exactly `num` times.

sinon.assert.callOrder(spy1, spy2, ...)

Passes if the provided spies where called in the specified order.

sinon.assert.calledOn(spy, obj)

Passes if the spy was ever called with `obj` as its `this` value.

sinon.assert.alwaysCalledOn(spy, obj)

Passes if the spy was always called with `obj` as its `this` value.

sinon.assert.calledWith(spy, arg1, arg2, ...)

Passes if the spy was called with the provided arguments.

sinon.assert.alwaysCalledWith(spy, arg1, arg2, ...)

Passes if the spy was always called with the provided arguments.

sinon.assert.neverCalledWith(spy, arg1, arg2, ...)

Passes if the spy was never called with the provided arguments.

sinon.assert.calledWithExactly(spy, arg1, arg2, ...)

Passes if the spy was called with the provided arguments and no others.

sinon.assert.alwaysCalledWithExactly(spy, arg1, arg2, ...)

Passes if the spy was always called with the provided arguments and no others.

sinon.assert.calledWithMatch(spy, arg1, arg2, ...)

Passes if the spy was called with matching arguments. This behaves the same as `sinon.assert.calledWith(spy, sinon.match(arg1), sinon.match(arg2), ...)`.

sinon.assert.alwaysCalledWithMatch(spy, arg1, arg2, ...)

Passes if the spy was always called with matching arguments. This behaves the same as `sinon.assert.alwaysCalledWith(spy, sinon.match(arg1), sinon.match(arg2), ...)`.

sinon.assert.neverCalledWithMatch(spy, arg1, arg2, ...)

Passes if the spy was never called with matching arguments. This behaves the same as `sinon.assert.neverCalledWith(spy, sinon.match(arg1), sinon.match(arg2), ...)`.

sinon.assert.threw(spy, exception)

Passes if the spy threw the given exception. The exception can be a string denoting its type, or an actual object. If only one argument is provided, the assertion passes if the spy ever threw any exception.

sinon.assert.alwaysThrew(spy, exception)

Like above, only required for all calls to the spy.

sinon.assert.expose(object, options)

Exposes assertions into another object, to better integrate with the test framework. For instance, JsTestDriver uses global assertions, and to make Sinon.JS assertions appear alongside them, you can do.

```
sinon.assert.expose(this);
```

This will give you `assertCalled(spy)`,`assertCallOrder(spy1, spy2, ...)` and so on.

The method accepts an optional options object with two options. **prefix** is a prefix to give assertions. By default it is “assert”, so `sinon.assert.called` becomes `target.assertCalled`. By passing a blank string, the exposed method will be `target.called`. The second option, **includeFail** is true by default, and copies over the `fail` and `failException` properties.

Matchers

API reference

Matcher API Combining matchers Custom matchers

Matchers can be passed as arguments to `spy.calledWith`, `spy.returned` and the corresponding `sinon.assert` functions as well as `spy.withArgs`. Matchers allow to be either more fuzzy or more specific about the expected value.

```
"test should assert fuzzy": function () {
    var book = {
        pages: 42,
        author: "cjno"
    };
    var spy = sinon.spy();

    spy(book);

    sinon.assert.calledWith(spy, sinon.match({ author: "cjno" }));
    sinon.assert.calledWith(spy, sinon.match.has("pages", 42));
}


"test should stub method differently based on argument types": function () {
    var callback = sinon.stub();
    callback.withArgs(sinon.match.string).returns(true);
    callback.withArgs(sinon.match.number).throws("TypeError");

    callback("abc"); // Returns true
    callback(123); // Throws TypeError
}
```

Matchers API

`sinon.match(number)`

Requires the value to be == to the given number.

`sinon.match(string)`

Requires the value to be a string and have the expectation as a substring.

`sinon.match(regexp)`

Requires the value to be a string and match the given regular expression.

`sinon.match(object)`

Requires the value to be not null or undefined and have at least the same properties as `expectation`. This supports nested matchers.

`sinon.match(function)`

See custom matchers.

sinon.match.any

Matches anything.

sinon.match.defined

Requires the value to be defined.

sinon.match.truthy

Requires the value to be truthy.

sinon.match.falsy

Requires the value to be falsy.

sinon.match.bool

Requires the value to be a boolean.

sinon.match.number

Requires the value to be a number.

sinon.match.string

Requires the value to be a string.

sinon.match.object

Requires the value to be an object.

sinon.match.func

Requires the value to be a function.

sinon.match.array

Requires the value to be an array.

sinon.match.regexp

Requires the value to be a regular expression.

sinon.match.date

Requires the value to be a date object.

sinon.match.same(ref)

Requires the value to strictly equal `ref`.

sinon.match.typeOf(type)

Requires the value to be of the given type, where `type` can be one of `"undefined"`, `"null"`, `"boolean"`, `"number"`, `"string"`, `"object"`, `"function"`, `"array"`, `"regexp"` or `"date"`.

sinon.match.instanceOf(type)

Requires the value to be an instance of the given type.

sinon.match.has(property[, expectation])

Requires the value to define the given property. The property might be inherited via the prototype chain. If the

optional expectation is given, the value of the property is deeply compared with the expectation. The expectation can be another matcher.

`sinon.match.hasOwn(property[, expectation])`

Same as `sinon.match.has` but the property must be defined by the value itself. Inherited properties are ignored.

Combining matchers

All matchers implement `and` and `or`. This allows to logically combine mutliple matchers. The result is a new matchers that requires both (and) or one of the matchers (or) to return true.

```
var stringOrNumber = sinon.match.string.or(sinon.match.number);

var bookWithPages = sinon.match.instanceOf(Book).and(sinon.match.has("pages"));
```

Custom matchers

Custom matchers are created with the `sinon.match` factory which takes a test function and an optional message. The test function takes a value as the only argument, returns `true` if the value matches the expectation and `false` otherwise. The message string is used to generate the error message in case the value does not match the expectation.

```
var trueIsh = sinon.match(function (value) {
    return !!value;
}, "trueIsh");
```

Sandboxes

API reference

sinon.sandbox sinon.test sinon.testCase

Sandboxes simplify working with fakes that need to be restored and/or verified. If you're using fake timers, fake XHR, or you are stubbing/spying on globally accessible properties you should use a sandbox to ease cleanup. By default the spy, stub and mock properties of the sandbox is bound to whatever object the function is run on, so if you don't want to manually `restore()`, you have to use `this.spy()` instead of `sinon.spy()` (and stub, mock).

```
"test using sinon.test sandbox": sinon.test(function () {
    var myAPI = { method: function () {} };
    this.mock(myAPI).expects("method").once();

    PubSub.subscribe("message", myAPI.method);
    PubSub.publishSync("message", undefined);
})
```

Sandbox API

`var sandbox = sinon.sandbox.create();`

Creates a sandbox object

`var sandbox = sinon.sandbox.create(config);`

The `sinon.sandbox.create(config)` method is mostly an integration feature, and as an end-user of Sinon.JS you will probably not need it.

Creates a pre-configured sandbox object. The configuration can instruct the sandbox to include fake timers, fake server, and how to interact with these. The default configuration looks like:

```
sinon.defaultConfig = {
    // ...
    injectInto: null,
    properties: ["spy", "stub", "mock", "clock", "server", "requests"],
    useFakeTimers: true,
    useFakeServer: true
}
```

injectInto

The sandbox's methods can be injected into another object for convenience. The `injectInto` configuration option can name an object to add properties to. Usually, this is set by `sinon.test` such that it is the `this` value in a given test function.

properties

What properties to inject. Note that simply naming "server" here is not sufficient to have a `server` property show up in the target object, you also have to set `useFakeServer` to true.

useFakeTimers

If `true`, the sandbox will have a `clock` property. Can also be an array of timer properties to fake.

useFakeServer

If `true`, `server` and `requests` properties are added to the sandbox. Can also be an object to use for fake server. The default one is `sinon.fakeServer`, but if you're using jQuery 1.3.x or some other library that does not set the XHR's `onreadystatechange` handler, you might want to do:

```
sinon.config = {
    useFakeServer: sinon.fakeServerWithClock
};
```

`sandbox.spy();`

Works exactly like `sinon.spy`, only also adds the returned spy to the internal collection of fakes for easy restoring through `sandbox.restore()`.

`sandbox.stub();`

Works almost exactly like `sinon.stub`, only also adds the returned stub to the internal collection of fakes for easy restoring through `sandbox.restore()`. The sandbox `stub` method can also be used to stub any kind of property. This is useful if you need to override an object's property for the duration of a test, and have it restored when the test completes.

`sandbox.mock();`

Works exactly like `sinon.mock`, only also adds the returned mock to the internal collection of fakes for easy restoring through `sandbox.restore()`.

`sandbox.useFakeTimers();`

Fakes timers and binds the `clock` object to the sandbox such that it too is restored when calling `sandbox.restore()`. Access through `sandbox.clock`.

`sandbox.useFakeXMLHttpRequest();`

Fakes XHR and binds the resulting object to the sandbox such that it too is restored when calling `sandbox.restore()`. Access requests through `sandbox.requests`.

`sandbox.useFakeServer();`

Fakes XHR and binds a server object to the sandbox such that it too is restored when calling `sandbox.restore()`. Access requests through `sandbox.requests` and server through `sandbox.server`.

`sandbox.restore();`

Restores all fakes created through sandbox.

Test methods

Wrapping test methods in `sinon.test` allows Sinon.JS to automatically create and manage sandboxes for you. The function's behavior can be configured through `sinon.config`.

`var wrappedFn = sinon.test(fn);`

The `wrappedFn` function works exactly like the original one in all respect - in addition a sandbox object is created and automatically restored when the function finishes a call. By default the spy, stub and mock properties of the sandbox is bound to whatever object the function is run on, so you can do `this.spy()` (and stub, mock) and it works exactly like `sandbox.spy()` (and stub, mock), except you don't need to manually `restore()`.

```
{
    injectIntoThis: true,
    injectInto: null,
    properties: ["spy", "stub", "mock", "clock", "server", "requests"],
    useFakeTimers: true,
    useFakeServer: true
}
```

Simply set `sinon.config` to override any or all of these, e.g.:

```
sinon.config = {
    useFakeTimers: false,
    useFakeServer: false
}
```

In this case, defaults are used for the non-existent properties. Additionally, sandboxes and tests will not have automatic access to the fake timers and fake server when using this configuration.

sinon.config

The configuration controls how Sinon binds properties when using `sinon.test`. The default configuration looks like:

`Boolean injectIntoThis`

Causes properties to be injected into the `this` object of the test function. Default `true`.

`Object injectInto`

Object to bind properties to. If this is `null` (default) and `injectIntoThis` is `false` (not default), the properties are passed as arguments to the test function instead.

`Array properties`

Properties to expose. Default is all: `["spy", "stub", "mock", "clock", "server", "requests"]`. However, the last three properties are only bound if the following two configuration options are `true` (which is the default).

`Boolean useFakeTimers`

Causes timers to be faked and allows `clock` property to be exposed. Default is `true`.

`Boolean useFakeServer`

Causes fake XHR and server to be created and allows `server` and `requests` properties to be exposed. Default is `true`.

Test cases

If you need the behavior of `sinon.test` for more than one test method in a test case, you can use `sinon.testCase`, which behaves exactly like wrapping each test in `sinon.test` with one exception: `setUp` and `tearDown` can share fakes.

```
var obj = sinon.testCase({});
```

Sinon.JS utilities

API reference

Sinon.JS has a few utilities used internally in lib/sinon.js. Unless the method in question is documented here, it should not be considered part of the public API, and thus is subject to change.

Utils API

`sinon.createStubInstance(constructor)`

Creates a new object with the given function as the protoype and stubs all implemented functions. The given constructor function is not invoked. See also the stub API.

`sinon.format(object)`

Formats an object for pretty printing in error messages. Sinon < 1.3.0 defaulted to coercing objects to strings. As of 1.3.0, Sinon uses buster-format by default, or Node's util module. Feel free to override this method with your own implementation if you prefer different visualization of e.g. objects. The method should return a string.

`sinon.log(string)`

Logs internal errors, helpful for debugging. By default this property is a noop function, set it to something that prints warnings in your environment for more help, e.g. (if you are using JsTestDriver):

```
sinon.log = function (message) {  jstestdriver.console.log(message); };
```

Sinon uses Semantic versioning.