Case study: real world dependency stubbing

Sinon is a simple tool that only tries to do a few things and do them well: creating and injecting test doubles (spies, fakes, stubs) into objects. Unfortunately, in todays world of build pipelines, complex tooling, transpilers and different module systems, doing the simple thing quickly becomes difficult. This article is a detailed step-by-step guide on how one can approach the typical issues that arise and various approaches for debugging and solving them. The real-world case chosen is using Sinon along with SWC (https://swc.rs/), running tests written in TypeScript in the Mocha test runner (https://mochajs.org/) and wanting to replace dependencies in this (SUT (http://xunitpatterns.com/SUT.html)). The essence is that there are always *many* approaches for achieving what you want. Some require tooling, some can get away with almost no tooling, some are general in nature (not specific to SWC for instance) and some are a blend. This means you can usually make some of these approaches work for other combinations of tooling as well, once you understand what is going on. Draw inspiration from the approach and figure out what works for you!

(https://srv.carbonads.net/ads/click/x/GTND42J7CVYDT2JECWS
segment=placement:sinonjsorg;)

Don't let auth hold you back, launch your product today with P
(https://srv.carbonads.net/ads/click/x/GTND42J7CVYDT2JECV
segment=placement:sinonjsorg;)

ADS VIA CARBON (HTTP://CARBONADS.NET/?UTM_SOURCE=SINONJSO

On Typescript

The Sinon project (and its maintainers) does not explicitly list TypeScript as a supported target environment. That does not mean Sinon will not run, just that there are so many complications that we cannot come up with guides on figuring out the details for you on every system :) Typescript is a super-set of EcmaScript (JavaScript) and can be transpiled in a wide variety of ways into EcmaScript, both by targetting different runtimes (ES5, ES2015, ES2023, etc) and module systems (CommonJS, ESM, AMD, ...). Some transpiler are closer to the what the standard TypeScript compiler produces, some are laxer in various ways and additionally they have all kinds of options to tweak the result. This is indeed complex, so before you dig yourself done in this matter, it is essential that you try to figure out what the resulting code *actually* looks like. As you will see in this guide, adding a few sprinkles of `console.log` with the output of `Object.getOwnPropertyDescriptor(object, propname)` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor) is usually sufficient to understand what is going on!

All code and working setups described in this guide are on Github (https://github.com/fatso83/sinon-swc-bug) and links to the correct branch can be found in each section.

Scenario

Tech

- Mocha: drives the tests
- SWC: very fast Rust-based transpiler able to target different module systems (CJS, ESM, ...) and target runtimes (ES5, ES2020, ...)
- Typescript: Type-safe EcmaScript superset
- Sinon: library for creating and injecting test doubles (stubs, mocks, spies and fakes)
- Module system: CommonJS

Wanted outcome

Being able to replace exports on the dependency `./other` with a Sinon created test double in `main.ts` when running tests (see code below).

Problem

Running tests with `ts-node` works fine, but changing the setup to using SWC instead results in the tests failing with the following output from Mocha:

```
1) main
       should mock:
     TypeError: Descriptor for property toBeMocked is non-configurable and non-
writable
```

Original code

main.ts

```
import { toBeMocked } from "./other";

export function main() {
  const out = toBeMocked();
  console.log(out);
}
```

other.ts

```
export function toBeMocked() {
  return "I am the original function";ing
}
```

main.spec.ts

```
import sinon from "sinon";
import "./init";
import * as Other from "./other";
import { main } from "./main";
import { expect } from "chai";

const sandbox = sinon.createSandbox();

describe("main", () => {
  let mocked;
  it("should mock", () => {
    mocked = sandbox.stub(Other, "toBeMocked").returns("mocked");
    main();
    expect(mocked.called).to.be.true;
  });
});
```

Additionally, both the `.swcrc` file used by SWC and the `tsconfig.json` file used by `ts-node` is setup to produce modules of the CommonJS form, not ES Modules.

Brief Analysis

The error message indicates the resulting output of transpilation is different from that of `ts-node`, as this is Sinon telling us that it is unable to do anything with the property of an object if the property descriptor (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) is essentially immutable. Let's sprinkle some debugging statements to figure out what the differences between the two tools are. First we will add these some debugging output to the beginning of the test, for instance just after `it("should mock", () => {`, to see what the state is *before* we attempt to do any modifications:

```
console.log('Other', Other)
console.log('Other property descriptors', Object.getOwnPropertyDescri
```

Now let's try what happens when running this again, once with the existing SWC setup and a second time after changing the config file for Mocha, `.mocharc.json`, to use `'ts-node'` instead of `'@swc/register` in its `'require'` array. This `--require` option of Node is for modules that will be run by Node before *your* code, making it possible to do stuff like hook into `require` and transpile code on-the-fly.

Output of a SWC configured run of `npm test`

```
Other { toBeMocked: [Getter] }
Other property descriptors {
  __esModule: {
    value: true,
    writable: false,
    enumerable: false,
    configurable: false
  },
  toBeMocked: {
    get: [Function: get],
    set: undefined,
    enumerable: true,
    configurable: false
  }
}
    1) should mock


  0 passing (4ms)
  1 failing

  1) main
       should mock:
     TypeError: Descriptor for property toBeMocked is non-configurable and non-
writable
```

Output of a `ts-node` configured run of `npm test`

```
Other { toBeMocked: [Function: toBeMocked] }
Other property descriptors {
  __esModule: {
    value: true,
    writable: false,
    enumerable: false,
    configurable: false
  },
  toBeMocked: {
    value: [Function: toBeMocked],
    writable: true,
    enumerable: true,
    configurable: true
  }
}
mocked
    ✔ should mock
```

The important difference to note about the object `Other` is that the property `toBeMocked` is a simple writable *value* in the case of `ts-node` and a non-configurable *getter* in the case of SWC. It being a getter is not a problem for Sinon, as we have a multitude of options for replacing those, but if `configurable` is set to `false` Sinon cannot really do anything about it.

If we take a look at

Conclusion of analysis

SWC transforms the imports on the form `import * as Other from './other'` into objects where the individual exports are exposed through immutable accessors (*getters*).

We can attack this issue in mainly 3 ways:

1. somehow reconfigure SWC to produce different output when running tests that we can work with, either making writable values or configurable getters
2. use pure dependency injection, opening up `./other.ts` to be changed from the inside
3. attack how modules are loaded, injecting an additional `require` “hook” (https://levelup.gitconnected.com/how-to-add-hooks-to-node-js-require-function-dee7acd12698)

Solutions

Mutating the output from the transpiler

> Working code (https://github.com/fatso83/sinon-swc-bug/tree/swc-with-mutable-exports)

If we can just flip the `configurable` flag to `true` during transpilation, Sinon could be instructed to replace the getter. Turns out, there is a SWC *plugin* that does just that: swc_mut_cjs_exports (https://www.npmjs.com/package/swc_mut_cjs_exports). By installing that and adding the following under the `jsc` key in `.swcrc`, you know get a configurable property descriptor.

```
"experimental": {
  "plugins": [[ "swc_mut_cjs_exports", {} ]]
},
```

A getter *is* different from a value, so you need to change your testcode slightly to replace the getter:

```
const stub = sandbox.fake.returns("mocked")
sandbox.replaceGetter(Other, "toBeMocked", () => stub)
```

Use pure dependency injection

> Working code (https://github.com/fatso83/sinon-swc-bug/tree/pure-di)

This technique works regardless of language, module systems, bundlers and tool chains, but requires slight modifications of the SUT to allow modifying it. You also do not get help from Sinon in automatically resetting state.

other.ts

```
function _toBeMocked() {
  return "I am the original function";
}

export let toBeMocked = _toBeMocked

export function _setToBeMocked(mockImplementation){
    toBeMocked = mockImplementation
}
```

main.spec.ts

```
describe("main", () => {
  let mocked;
  let original = Other.toBeMocked;

  after(() => Other._setToBeMocked(original))

  it("should mock", () => {
    mocked = sandbox.stub().returns("mocked");
    Other._setToBeMocked(mocked)
    main();
    expect(mocked.called).to.be.true;
  });
```

Hooking into Node's module loading

> Working code (https://github.com/fatso83/sinon-swc-bug/tree/cjs-mocking)

This is what the article on *targetting the link seams* (/how-to/link-seams-commonjs/) is about. The only difference here is using Quibble instead of Proxyquire. Quibble is slightly terser and also supports being used as a ESM *loader*, making it a bit more modern and useful. The end result:

```
describe("main module", () => {
  let mocked, main;

  before(() => {
    mocked = sandbox.stub().returns("mocked");
    quibble("./other", { toBeMocked: mocked });
    ({main} = require("./main"));
  });

  it("should mock", () => {
    main();
    expect(mocked.called).to.be.true;
  });
});
```

Final remarks

As can be seen, there are lots of different paths to walk in order to achieve the same basic goal. Find the one that works for your case.