Unit testing using Mocha and Chai.js

mocha - lightweight node.js test framework
chai.js - TDD (Test driven development) assertion library for node

Both **Mocha** and **Chai** run in NodeJs and the browser and allow asynchronous testing.
Chai provides us with several APIs like Assert, Expect/Should, and more. Mocha provides all the functionality required for automated testing in simpler ways.

Install as dev dependency

```
npm i —save-dev mocha
npm i —save-dev chai
```

Add test script to package.json - if test folder use tests/, if spec, use specs/

```
"scripts": {
   "test": "mocha './specs/**/*.spec.js' "      // test all files with extension .spec.js in sub-directories as well
}
```

we create a spec folder to write all our mocha tests
to test file1.js, create file1.spec.ts in spec/ directory as per modular structure

we need a test suite to write the test cases
to create a test suite using mocha -

```
describe("test_suite_title", function () {
        // we write our test cases here
        // we write each test case using it() method

        it("test_case_title", function () {
                // test the method, function, class, etc.
                // and assert/ expect the result using assert/ expect from chai

                // e.g. 1 testing add() method of class ABC, suppose we created a obj of the class above in this file
                expect(obj.add(2, 5)).to.be.equal(7);
        });
});
```

in unit testing, we can check -
- if the func, class, etc is called
- how many times called
- with what args
- if returns something, what

this is called SPY in unit testing and it is done using sinon library

```
npm i —save-dev sinon
```

```
require(‘sinon’)
describe(….. {
	// write test case to spy on the method
	it(‘spy_test’, function () {
		var spy = sinon.spy(obj, ‘func_name’);
		// use assertions to test the calls
	});
});
```

mock using sinon -
Mocking is very imp in testing as while testing we shouldn’t call the actual function

e.g. ABC.js
```
class ABC {
	constructor() {}

	function func1() {
		console.log(“hello”);
	}

	function func2(arg1, arg2) {
		this.func1();
		console.log(arg1, arg2);
	}
};
```

so when we’re testing func2, we shouldn’t actually call func1 to see whether func2 is behaving as expected. here mocking func1 comes in handy
so methods should always be mocked if it has no impact on the calling method
```
describe(….. {
	// write test case to mock the method
	it(‘mock_test’, function () {
		var mock = sinon.mock(obj);
		var expectations = mock.expects(‘func1’)    // expect func1 to be called
		expectations.exactly(1)    // how many times it should have been called
		// if it has args, use expectations.withArgs(“actuall_arg”)

		// call the func2 to test it
		obj.func2(10, 20);
		mock.verify()
	});
});
```

stubs -
in the mocking case, if now the func1 returns something that is being used in func2, we still shouldn’t call it, instead we use stubs
stubs mean that we assume func1 is returning a specific value and write the test case on basis of it
e.g. ABC.js

```
class ABC {
	constructor() {}

	function func1(arg1, arg2) {
		return arg1 + arg2;
	}

	function func2(arg1, arg2) {
		var res = this.func1(arg1, arg2);
		return res
	}
};
```

ABC.spec.js

```
describe(….. {
	// write test case to stub the method
	it(‘stub_test’, function () {
		var stub = sinon.stub(obj, “func1”);
		stub.withArgs(10, 20).returns(40);    // assume it returns 40 as value

		// call the func2 to test it
		expect(obj.func2(10, 20)).to.be.equal(40);
	});
});
```

testing a promise -
e.g. ABC.js

```
class ABC {
	constructor() {}

	function func1(arg1, arg2) {
		return arg1 + arg2;
	}

	function func2(arg1, arg2) {
		var res = this.func1(arg1, arg2);
		return res
	}
```

```
    testPromise() {
        return new Promise( function (resolve, reject) {
            setTimeout( () => resolve(10), 3000);      // return 1 after 3
seconds
        }).then( function (result) {
            return return * 2;       // print 10 * 2 i.e. 20 after 3 seconds
        });
    }
};
```

```
ABC.spec.js
describe(….. {
    // write test case to test the promise
    it('promise_test', function (done) {
        this.timeout(0);        // this will wait for as long as promise is resolved /
rejected
        // call the func to test it
        obj.testPromise().then( function (result) {
            expect(result).to.be.equal(20);
            done();
        });
    });
});
```

for async test and hooks, done() should be called which ensures completion of test
another way is to use **chai-as-promised** package
npm i —save-dev chai-as-promised

and then in spec.js file
const **chaiaspromise** = require('chai-as-promised')
chai**.use(**chaiaspromise**)**

so now we don't need the done section

```
ABC.spec.js
describe(….. {
    // write test case to stub the method
    it('promise_test', function () {
        this.timeout(0);        // this will wait for as long as promise is resolved /
rejected
        // call the func to test it
        return expect(obj.testPromise()).to.be.eventually.equal(20);
    });
});
```

bypassing ajax calls using nock -

```
ABC.js
testXHRFunc() {
        return new Promise((resolve, reject) => {
        var xhr = new XMLHttpRequest();
        xhr.open("post", "https://httpbin.org/post", true);

        xhr.onreadystatechange = () => {
        if (xhr.readyState == 4) {
                        if (xhr.status == 200) {
                                resolve(JSON.parse(xhr.responseText));
                        } else {
                                reject(xhr.status);
                        }
        }
        };
        xhr.send();
    })
        .then((result) => {
        return result;
        })
        .catch((error) => {
        return error;
        });
 }
```

we can mention what output we need from the api call using nock without actually hitting the api

```
ABC.spec.js
describe("test suite to stub XHR calls", () => {
        // bypassing actual ajax calls
        it("mock and stub xhr call ", () => {
                const scope = nock("https://httpbin.org").post("/post").reply(200,
{ id: 123 });
                // with this we are saying make a dummy call to the url and we're
expecting 200 as status and an id 123

                obj.testXHRFunc().then(function (result) {
                        expect(result).to.deep.equal({ id: 123 });
                        // expect(result).to.be.equal({ id: 1234 }); // tc fails
                        done();
                }).catch( err => done( new Error("test case failed", err)));
        });
});
```

Hooks -
Mocha provides a no. of imp hooks that we should use frequently
we can use them before / after test cases / suites - it helps reduce code significantly and also helps maintaining the code properly
primarily there are 4 hooks - before, beforeEach, after, afterEach

before & after - executed once for each test suite
beforeEach & afterEach - executed for each test case

```
describe("test suite 1", () => {
	// hooks in mocha that should be used -
	// these are applicable only for this test suite
	before(() => console.log("------- Before test suite"));
	beforeEach(() => console.log("------- Before each test case"));
	after(() => console.log("------- After test suite"));
	afterEach(() => console.log("------- After each test case"));

	 // write test cases here using it method
	it("test case for hello world", () => {
		// test the method/class, etc here by calling
		// and assert or expect, etc. the result using assert, expect methods
from chai.js
		expect(obj.sayHello()).to.be.equal("Hello");
	});

	it("test case for add functin", () => {
		expect(obj.add(10, 20)).to.be.equal(30);
	});
});
```

output -
```
 test suite 1
------- Before test suite
------- Before each test case
   ✔ test case for hello world
------- After each test case
------- Before each test case
   ✔ test case for add functin
------- After each test case
------- After test suite
```

why to use hooks
e.g. when we try to copy a spying test case and run it
it will fail saying trying to wrap a function which is already wrapped
this happens as **Sinon.spy()** creates a wrapper around the function which will

be available to all the test cases in the test suite
to make a copy, we will need to restore the wrapper using **Sinon.restore()** after using the wrapper in the test case

but instead of each time writing Sinon.restore(), instead we can use hooks like
beforeEach(() => Sinon.restore());

to create a root level hook, write the before, etc functions outside the test suites, these will be executed for each test case in each file in test/ directory

Generating a unit testing coverage report -
using instanbul's nyc package
npm i —save-dev nyc

then add to package.json:

```
“scripts”: {
	"coverage": "nyc --reporter=lcov --reporter=text npm run test"
}
```

although it works with commonJs, it doesn't work with es6 modules, so use c8 package instead
npm i —save-dev c8

then add to package.json:

```
“scripts”: {
	"coverage": “c8 --r=lcov --r=text npm run test"
}
```

create a .c8rc to configure the settings for c8

```
{
	"check-coverage": true,
	"branches": 80,
	"lines": 80,
	"functions": 80,
	"statements": 80
}
```

adding —watch keeps the unit testing running and keeps looking for any changes in the test files continuously, so that we don't have to run npm run coverage each time
in package.json

```
“scripts”: {
	“test”: "mocha './test/**/*.spec.js' --watch"
}
```