

Write Good Tests With Mocha

before() Hooks

In a test file, the function `before()` will be executed first, regardless of its placement in the code block.

`before()` is often used to set up code, like variables and values, for other function calls to use in their execution.

```
before(() => {  
  path = './message.txt';  
});
```

beforeEach() Hooks

In a test file, the function `beforeEach()` will be executed before each test. `beforeEach()` is often used to set up or reset code, like variables and values, for other function calls to use in their execution.

```
beforeEach(() => {  
  testCounter++;  
});
```

after() Hooks

In a test file, the function `after()` will be executed last, regardless of its placement in the code block.

`after()` is often used to print out results from the tests that were run in the suite or to reset variables and values.

```
after(() => {  
  console.log("number of tests: " +  
    testCounter);  
});
```

afterEach() Hooks

In a test file, the function `afterEach()` will be executed after each test. `afterEach()` is often used to print out results from a particular test that was run in the suite or to reset variables and values.

```
afterEach(() => {  
  path = './message.txt';  
});
```

Test Frameworks

Test frameworks are used to organize and automate tests that provide useful feedback when errors occur.

describe() functions

In Mocha, the `describe()` function is used to group tests. It accepts a string to describe the group of tests and a callback function which contains `it()` tests.

Calls to `describe()` are commonly nested to resemble the structure of the code being tested.

```
describe('group of tests', () => {  
  //Write it functions here  
  
});
```

it() functions

In Mocha, the `it()` function is used to execute individual tests. It accepts a string to describe the test and a callback function to execute assertions. Calls to `it()` are commonly nested within `describe()` blocks.

```
describe('+', () => {
  it('returns the sum of its
  arguments', () => {
    // Write assertions here

  });
});
```

The assert Library

The `assert` library is used to make assertions. It contains numerous functions that enable the tester to write easily readable assertions and throw `AssertionError`s within a test.

```
describe('+', () => {
  it('returns the sum of its
  arguments', () => {
    // Write assertion here
    assert.ok(3 + 4 === 7)

  });
});
```

assert.ok()

The `assert.ok()` function is be used to evaluate a boolean expression within a test. If the expression evaluates to `false`, an `AssertionError` is thrown.

```
describe('+', () => {
  it('returns the sum of its
  arguments', () => {
    // Write assertion here
    assert.ok(3 + 4 === 7)

  });
});
```

Setup Phase

In testing, the Setup phase is where objects, variables, and set conditions that tests depend on are created.

```
describe('.pop', () => {
  it('returns the last element in the
  array [3phase]', () => {
    // Setup
    const knightString = 'knight';
    const jediPath = ['padawan',
knightString];
    // Exercise
    const popped = jediPath.pop();
    // Verify
    assert.ok(popped ===
knightString);
  });
});
```

Exercise Phase

In testing, the Exercise phase is where the functionality under test is executed.

```
describe('.pop', () => {
  it('returns the last element in the
  array [3phase]', () => {
    // Setup
    const knightString = 'knight';
    const jediPath = ['padawan',
knightString];
    // Exercise
    const popped = jediPath.pop();
    // Verify
    assert.ok(popped ===
knightString);
  });
});
```

Verify Phase

In testing, the Verify phase is where expectations are checked against the result of the exercise phase.

`assert` would be used here.

```
describe('.pop', () => {
  it('returns the last element in the
  array [3phase]', () => {
    // Setup
    const knightString = 'knight';
    const jediPath = ['padawan',
    knightString];
    // Exercise
    const popped = jediPath.pop();
    // Verify
    assert.ok(popped ===
    knightString);
  });
});
```

Teardown Phase

In testing, the Teardown phase is where the environment is reset before the next test runs. The teardown phase ensures that a test is isolated from other tests.

```
it('creates a new file with a string
of text', () => {
  // Setup
  path = './message.txt';
  str = '';

  // Exercise: write to file
  fs.appendFileSync(path, str);

  // Verify: compare file contents
  to string
  const contents
  = fs.readFileSync(path);
  assert.equal(contents.toString(),
  str);

  // Teardown: restore file
  fs.unlinkSync(path);
});
```

Tests in Isolation

A project's tests should run in isolation from one another. One test shouldn't affect another. Tests should be able to run in any order.

`assert.equal()`

`assert.equal()` verifies a loose equality (`==`) comparison. Using `assert.equal()` is more expressive, it's more clear that it's verifying equality than `assert.ok()`.

```
const landAnimals = ['giraffe',  
  'squirrel'];  
const waterAnimals = ['shark',  
  'stingray'];
```

```
landAnimals.push('frog');  
waterAnimals.push('frog');
```

```
assert.equal(landAnimals[2],  
  waterAnimals[2]);
```

`assert.strictEqual()`

`assert.strictEqual()` verifies a strict equality (`===`) comparison.

```
const a = 3;  
const b = '3';  
assert.equal(a, b);  
assert.strictEqual(a, b);
```

`assert.deepEqual()`

`assert.deepEqual()` compares values within two objects. It will compare the values using loose (`==`) equality.

```
const a = {relation: 'twin', age:  
  '17'};  
const b = {relation: 'twin', age:  
  '17'};  
  
assert.strictEqual(a, b);
```

Why Test?

Testing can catch and identify issues with your implementation code before you deploy it to users