Setup and Teardown

EDIT

Often while writing tests you have some setup work that needs to happen before tests run, and you have some finishing work that needs to happen after tests run. Jest provides helper functions to handle this.

Repeating Setup For Many Tests

If you have some work you need to do repeatedly for many tests, you can use `beforeEach` and `afterEach`.

For example, let's say that several tests interact with a database of cities. You have a method `initializeCityDatabase()` that must be called before each of these tests, and a method `clearCityDatabase()` that must be called after each of these tests. You can do this with:

Copy

```
beforeEach(() => {
  initializeCityDatabase();
});

afterEach(() => {
  clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

`beforeEach` and `afterEach` can handle asynchronous code in the same ways that tests can handle asynchronous code - they can either take a `done` parameter or return a promise. For example, if `initializeCityDatabase()` returned a promise that resolved when the database was initialized, we would want to return that promise:

One-Time Setup

In some cases, you only need to do setup once, at the beginning of a file. This can be especially bothersome when the setup is asynchronous, so you can't do it inline. Jest provides `beforeAll` and `afterAll` to handle this situation.

For example, if both `initializeCityDatabase` and `clearCityDatabase` returned promises, and the city database could be reused between tests, we could change our test code to:

```
beforeAll(() => {
  return initializeCityDatabase();
});

afterAll(() => {
  return clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

Copy

Scoping

By default, the `before` and `after` blocks apply to every test in a file. You can also group tests together using a `describe` block. When they are inside a `describe` block, the `before` and `after` blocks only apply to the tests within that `describe` block.

For example, let's say we had not just a city database, but also a food database. We could do different setup for different tests:

```
// Applies to all tests in this file
beforeEach(() => {
  return initializeCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});
```

Copy

```
  // Applies only to tests in this describe block
  beforeEach(() => {
    return initializeFoodDatabase();
  });

  test('Vienna <3 sausage', () => {
    expect(isValidCityFoodPair('Vienna', 'Wiener Schnitzel')).toBe(true);
  });

  test('San Juan <3 plantains', () => {
    expect(isValidCityFoodPair('San Juan', 'Mofongo')).toBe(true);
  });
});
```

Note that the top-level `beforeEach` is executed before the `beforeEach` inside the `describe` block. It may help to illustrate the order of execution of all hooks.

Copy

```
beforeAll(() => console.log('1 - beforeAll'));
afterAll(() => console.log('1 - afterAll'));
beforeEach(() => console.log('1 - beforeEach'));
afterEach(() => console.log('1 - afterEach'));
test('', () => console.log('1 - test'));
describe('Scoped / Nested block', () => {
  beforeAll(() => console.log('2 - beforeAll'));
  afterAll(() => console.log('2 - afterAll'));
  beforeEach(() => console.log('2 - beforeEach'));
  afterEach(() => console.log('2 - afterEach'));
  test('', () => console.log('2 - test'));
});

// 1 - beforeAll
// 1 - beforeEach
// 1 - test
// 1 - afterEach
// 2 - beforeAll
// 1 - beforeEach
// 2 - beforeEach
// 2 - test
// 2 - afterEach
// 1 - afterEach
// 2 - afterAll
// 1 - afterAll
```

Order of execution of describe and test blocks

finish and be tidied up before moving on.

Consider the following illustrative test file and output:

```
describe('outer', () => {
  console.log('describe outer-a');

  describe('describe inner 1', () => {
    console.log('describe inner 1');
    test('test 1', () => {
      console.log('test for describe inner 1');
      expect(true).toEqual(true);
    });
  });

  console.log('describe outer-b');

  test('test 1', () => {
    console.log('test for describe outer');
    expect(true).toEqual(true);
  });

  describe('describe inner 2', () => {
    console.log('describe inner 2');
    test('test for describe inner 2', () => {
      console.log('test for describe inner 2');
      expect(false).toEqual(false);
    });
  });

  console.log('describe outer-c');
});

// describe outer-a
// describe inner 1
// describe outer-b
// describe inner 2
// describe outer-c
// test for describe inner 1
// test for describe outer
// test for describe inner 2
```

Copy

General Advice

```
test.only('this will be the only test that runs', () => {
  expect(true).toBe(false);
});

test('this test will not run', () => {
  expect('A').toBe('A');
});
```

Copy

If you have a test that often fails when it's run as part of a larger suite, but doesn't fail when you run it alone, it's a good bet that something from a different test is interfering with this one. You can often fix this by clearing some shared state with `beforeEach`. If you're not sure whether some shared state is being modified, you can also try a `beforeEach` that logs data.

← TESTING ASYNCHRONOUS CODE

MOCK FUNCTIONS →