Version: 29.7

Using Matchers

Jest uses "matchers" to let you test values in different ways. This document will introduce some commonly used matchers. For the full list, see the `expect` API doc.

Common Matchers

The simplest way to test a value is with exact equality.

```
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});
```

In this code, `expect(2 + 2)` returns an "expectation" object. You typically won't do much with these expectation objects except call matchers on them. In this code, `.toBe(4)` is the matcher. When Jest runs, it tracks all the failing matchers so that it can print out nice error messages for you.

`toBe` uses `Object.is` to test exact equality. If you want to check the value of an object, use `toEqual`:

```
test('object assignment', () => {
  const data = {one: 1};
  data['two'] = 2;
  expect(data).toEqual({one: 1, two: 2});
});
```

`toEqual` recursively checks every field of an object or array.

> **TIP**

> `toEqual` ignores object keys with `undefined` properties, `undefined` array items, array sparseness, or object type mismatch. To take these into account use `toStrictEqual` instead.

You can also test for the opposite of a matcher using `not`:

```
test('adding positive numbers is not zero', () => {
  for (let a = 1; a < 10; a++) {
    for (let b = 1; b < 10; b++) {
      expect(a + b).not.toBe(0);
    }
  }
});
```

Truthiness

In tests, you sometimes need to distinguish between `undefined`, `null`, and `false`, but you sometimes do not want to treat these differently. Jest contains helpers that let you be explicit about what you want.

- `toBeNull` matches only `null`
- `toBeUndefined` matches only `undefined`
- `toBeDefined` is the opposite of `toBeUndefined`
- `toBeTruthy` matches anything that an `if` statement treats as true
- `toBeFalsy` matches anything that an `if` statement treats as false

For example:

```
test('null', () => {
  const n = null;
  expect(n).toBeNull();
  expect(n).toBeDefined();
  expect(n).not.toBeUndefined();
  expect(n).not.toBeTruthy();
  expect(n).toBeFalsy();
});
```

```
test('zero', () => {
  const z = 0;
  expect(z).not.toBeNull();
  expect(z).toBeDefined();
  expect(z).not.toBeUndefined();
  expect(z).not.toBeTruthy();
  expect(z).toBeFalsy();
});
```

You should use the matcher that most precisely corresponds to what you want your code to be doing.

Numbers

Most ways of comparing numbers have matcher equivalents.

```
test('two plus two', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);

  // toBe and toEqual are equivalent for numbers
  expect(value).toBe(4);
  expect(value).toEqual(4);
});
```

For floating point equality, use `toBeCloseTo` instead of `toEqual`, because you don't want a test to depend on a tiny rounding error.

```
test('adding floating point numbers', () => {
  const value = 0.1 + 0.2;
  //expect(value).toBe(0.3);           This won't work because of rounding
error
  expect(value).toBeCloseTo(0.3); // This works.
});
```

Strings

You can check strings against regular expressions with `toMatch`:

```
test('there is no I in team', () => {
  expect('team').not.toMatch(/I/);
});

test('but there is a "stop" in Christoph', () => {
  expect('Christoph').toMatch(/stop/);
});
```

Arrays and iterables

You can check if an array or iterable contains a particular item using `toContain`:

```
const shoppingList = [
  'diapers',
  'kleenex',
  'trash bags',
  'paper towels',
  'milk',
];

test('the shopping list has milk on it', () => {
  expect(shoppingList).toContain('milk');
  expect(new Set(shoppingList)).toContain('milk');
});
```

Exceptions

If you want to test whether a particular function throws an error when it's called, use `toThrow`.

```
function compileAndroidCode() {
  throw new Error('you are using the wrong JDK!');
}
```

```
test('compiling android goes as expected', () => {
  expect(() => compileAndroidCode()).toThrow();
  expect(() => compileAndroidCode()).toThrow(Error);

  // You can also use a string that must be contained in the error message
or a regexp
  expect(() => compileAndroidCode()).toThrow('you are using the wrong
JDK');
  expect(() => compileAndroidCode()).toThrow(/JDK/);

  // Or you can match an exact error message using a regexp like below
  expect(() => compileAndroidCode()).toThrow(/^you are using the wrong
JDK$/); // Test fails
  expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK!
$/); // Test pass
});
```

TIP

The function that throws an exception needs to be invoked within a wrapping function otherwise the `toThrow` assertion will fail.

And More

This is just a taste. For a complete list of matchers, check out the reference docs.

Once you've learned about the matchers that are available, a good next step is to check out how Jest lets you test asynchronous code.

Edit this page

*Last updated on **Sep 12, 2023** by **Simen Bekkhus***