Версия: 29.7

Тестирование асинхронного кода

Зачастую JavaScript код выполняется асинхронно. При работе с асинхронным кодом, Jest нужно знать когда тестируемый код завершен, до того, как он сможет перейти к следующему тесту. В Jest этого можно добиться несколькими способами.

Промисы

Возвращайте промис в своем тесте, и Jest будет ждать `resolve` — успешного завершения промиса. Если промис отклонён(reject), то утверждение не будет выполнено.

Например, представьте что `fetchData` вместо использования коллбэка возвращает промис, который должен в случае успешного выполнения вернуть строку `'peanut butter'`. Мы можем протестировать это поведение так:

```
test('the data is peanut butter', () => {
  return fetchData().then(data => {
    expect(data).toBe('peanut butter');
  });
});
```

Асинхронный / Ожидающий

Кроме того, Вы можете использовать `async` и `await` в Ваших тестах. Чтобы написать асинхронный тест, просто используйте `async` перед определением функции передаваемой в `test`. Например, тот же `fetchData` сценарий может быть протестирован следующим образом:

```
test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});

test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
```

```
    await fetchData();
  } catch (e) {
    expect(e).toMatch('error');
  }
});
```

Вы можете комбинировать `async` и `await` вместе с `.resolves` или `.rejects`.

```
test('данные являются арахисовым маслом', async () => {
  await expect(fetchData()).resolves.toBe('арахисовое масло');
});

test('fetch вернёт ошибку', async () => {
  await expect(fetchData()).rejects.toMatch('error');
});
```

В этих случаях, `async` и `await` удобный синтаксический сахар для той же самой логики, что использовалась с примерами на промисах.

> **ОСТОРОЖНО**
>
> Убедитесь, что вы возвращаете промис или ожидаете его завершения. Если вы пропустите выражение `return`/`await`, ваш тест будет завершён до того, как промис, полученный от `fetchData` разрешит(resolve) или отклонит(reject) его.

Если Вы ожидаете, что промис будет отклонён, используйте метод `.catch`. Убедитесь, что добавлены `expect.assertions`, чтобы убедиться, что вызвано определенное количество утверждений. В противном случае, завершённый промис не провалит тест.

```
test('the fetch fails with an error', () => {
  expect.assertions(1);
  return fetchData().catch(e => expect(e).toMatch('error'));
});
```

Обратные вызовы

Если вы не используете промисы, вы можете использовать обратные вызовы. Например, предположим, что `fetchData` вместо возврата промиса ожидает коллбек, т.е. извлекает некоторые данные и вызывает `callback(null, data)` по завершении. И вы хотите проверить, что возвращаемые данные это строка `'арахисовое масло'`.

По умолчанию Jest тесты завершаются, как только они достигают конца их исполнения. Это значит, что этот тест *не будет* работать как предполагается:

```
// Не делайте так!
test('the data is peanut butter', () => {
  function callback(error, data) {
    if (error) {
      throw error;
    }
    expect(data).toBe('peanut butter');
  }

  fetchData(callback);
});
```

Проблема в том, что тест завершится, как только завершится выполнение `fetchData`, прежде чем будет вызван `callback`.

Существует альтернативная форма `test`, которая исправляет это. Вместо того чтобы помещать тест в функцию с пустым аргументом, передавайте в нее аргумент с именем `done`. Перед завершением теста Jest будет ждать вызова `done`, и только потом тест завершится.

```
test('the data is peanut butter', done => {
  function callback(error, data) {
    if (error) {
      done(error);
      return;
    }
    try {
      expect(data).toBe('peanut butter');
      done();
    } catch (error) {
      done(error);
    }
  }

  fetchData(callback);
});
```

Если `done()` никогда не вызовется, тест упадет (по тайм-ауту), а это как раз то, чего мы хотим.

Если `expect` завершится неудачно, то он выбросит ошибку и `done()` не будет вызван. Если мы хотим логгировать ошибку, мы должны обернуть `expect` в блок `try` и передать ошибку в блоке `catch` в `done`. В противном случае мы закончим с ошибкой непрозрачного тайм-аута, которая не показывает какое значение было получено `expect(data)`.

> **ОСТОРОЖНО**
>
> Jest выдаст ошибку, если той же тестовой функции будет передан `done()` обратный вызов и она вернет обещание. Это сделано в качестве меры предосторожности, чтобы избежать утечек

памяти в ваших тестах.

`.resolves` / `.rejects`

Вы также можете использовать `.resolves` в выражении expext и Jest будет ожидать что промис будет выполнен. Если промис будет выполнен, то тест автоматически прервётся.

```
test('the data is peanut butter', () => {
  return expect(fetchData()).resolves.toBe('peanut butter');
});
```

Обязательно убедитесь, что вы возвращаете успешное исполнение промиса — если забыть про этот `return`, то тест завершится еще до того как успешно завершится промис, вернувшийся из `fetchData`, и у then() появится возможность выполнить обратный вызов.

Если Вы ожидаете, что промис будет отклонён, используйте `.rejects`. Он работает аналогично `.resolves`. Если промис будет выполнен, то тест автоматически прервётся.

```
test('the fetch fails with an error', () => {
  return expect(fetchData()).rejects.toMatch('error');
});
```

Ни одна их этих форм не обладает серьезными преимуществами перед остальными, и вы можете смешивать и сопоставлять их во всем своем коде или даже в рамках одного файла. Все зависит только от того, в каком стиле вам легче писать тесты.