Версия: 29.7

Пример с асинхронностью

Во-первых включите поддержку Babel в Jest, как описано в руководстве Приступая к работе.

Давайте создадим модуль, который получает данные пользователя из API и возвращает имя пользователя.

user.js

```
import request from './request';

export function getUserName(userID) {
  return request(`/users/${userID}`).then(user => user.name);
}
```

В приведенной выше реализации мы ожидаем, что модуль `request.js` вернёт промис. Мы связываем вызов с `then`, чтобы получить имя пользователя.

Теперь представим реализацию `request.js`, которая идет в сеть и извлекает данные некоторого пользователя:

request.js

```
const http = require('http');

export default function request(url) {
  return new Promise(resolve => {
    // Пример HTTP запроса для извлечения
    // данных пользователя из API.
    // Для данного модуля используется мок из __mocks__/request.js
    http.get({path: url}, response => {
      let data = '';
      response.on('data', _data => data += _data);
      response.on('end', () => resolve(data));
    });
  });
}
```

Из-за того что мы не хотим обращаться к сети в нашем тесте, мы вручную создадим мок для нашего модуля `request.js` в папке `__mocks__` (название папки чувствительно к регистру `__MOCKS__` не сработает). Он может выглядеть примерно так:

__mocks__/request.js

```
const users = {
  4: {name: 'Mark'},
  5: {name: 'Paul'},
};

export default function request(url) {
  return new Promise((resolve, reject) => {
    const userID = parseInt(url.substr('/users/'.length), 10);
    process.nextTick(() =>
      users[userID]
        ? resolve(users[userID])
        : reject({
            error: `User with ${userID} not found.`,
          }),
    );
  });
}
```

Теперь давайте напишем тест для нашей асинхронной функции.

__tests__/user-test.js

```
jest.mock('../request');

import * as user from '../user';

// Должен быть возвращен тестируемый промис.
it('works with promises', () => {
  expect.assertions(1);
  return user.getUserName(4).then(data => expect(data).toBe('Mark'));
});
```

Мы вызываем `jest.mock('../request')`, чтобы указать Jest использовать наш вручную заданный мок. `it` ожидает, что возвращаемое значение будет промисом, который в итоге будет разрешен. Вы можете создавать столько цепочек промисов, сколько хотите и вызывать `expect` в любой момент, пока в конце вы возвращаете промис.

`.resolves`

Существует менее подробный способ использования `resolves` для получения значения выполненного промиса вместе с любым другим сопоставителем. Если промис отклонён, то утверждение не будет выполнено.

```
it('works with resolves', () => {
  expect.assertions(1);
  return expect(user.getUserName(5)).resolves.toBe('Paul');
});
```

`async/await`

Также возможно написание тестов с использованием синтаксиса `async`/`await`. Вот так можно переписать предыдущие примеры:

```
// async/await могут быть использованы.
it('works with async/await', async () => {
  expect.assertions(1);
  const data = await user.getUserName(4);
  expect(data).toBe('Mark');
});

// async/await can also be used with `.resolves`.
it('works with async/await and resolves', async () => {
  expect.assertions(1);
  await expect(user.getUserName(5)).resolves.toBe('Paul');
});
```

Чтобы включить async/await в вашем проекте, установите `@babel/preset-env` и включите эту функцию в файле `babel.config.js`.

Обработка ошибок

Ошибки могут быть пойманы с помощью метода `.catch`. Убедитесь, что добавлены `expect.assertions`, чтобы убедиться, что вызвано определенное количество утверждений. В противном случае, завершённый промис не провалит тест:

```
// Тестирование асинхронных ошибок с помощью Promise.catch.
it('тестирует ошибки с использованием промисов', () => {
  expect.assertions(1);
  return user.getUserName(2).catch(e =>
    expect(e).toEqual({
      error: 'Пользователь с ID 2 не найден.',
    }),
  );
});

// Или используя async/await.
it('тестирование ошибок с async/await', async () => {
```

```
  expect.assertions(1);
  try {
    await user.getUserName(1);
  } catch (e) {
    expect(e).toEqual({
      error: 'User with 1 not found.',
    });
  }
});
```

`.rejects`

`.rejects` хэлпер работает как `.resolves` хэлпер. Если промис будет выполнен, то тест автоматически прервётся. `expect.assertions(number)` не является обязательным, но рекомендуется для подтверждения точного числа проверок вызванных во время теста. В противном случае легко забыть `return`/`await` в проверках с использованием `.resolves`.

```
// Тестирование асинхронных ошибок с помощью `.rejects`.
it('тестирование ошибок с rejects', () => {
  expect.assertions(1);
  return expect(user.getUserName(3)).rejects.toEqual({
    error: 'User with 3 not found.',
  });
});

// Или используйте async/await с `.rejects`.
it('тестирование ошибок с async/await и rejects', async () => {
  expect.assertions(1);
  await expect(user.getUserName(3)).rejects.toEqual({
    error: 'User with 3 not found.',
  });
});
```

Код для этого примера доступен по адресу examples/async.

Если вы хотите протестировать таймеры, как, например, `setTimeout`, обратите внимание на документацию Timer mocks.