https://jestjs.io/docs/en/getting-started.html

Jest telepítése:

```
npm install --save-dev jest
```

Készítsünk egy példát, ahol összeadunk két számot: (sum.js)

```
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

És írjunk rá egy tesztet: (sum.test.js)

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

A package.json-nek kell tartalmaznia a következő részt (ha nem tartalmazza adjuk hozzá):

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Teszt futtatása:

```
npm run test
```

Using Matchers

Pontos egyezőség:

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

Objektumok egyezősége:

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

Vagy éppen annak tesztelése, hogy milyen értéket nem várunk:

```
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);
    }
  }
});
```

- `toBeNull` : `null`
- `toBeUndefined` : `undefined`
- `toBeDefined` : a `toBeUndefined ellentéte`
- `toBeTruthy` : minden kifejezés, ami igazra értékelődik ki (minden kifejezés, aminél az if igazat ad)
- `toBeFalsy` : minden kifejezés, ami hamisra értékelődik ki (minden kifejezés, aminél az if hamist ad)

```
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();
});
```

Számok:

```
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);
});
```

```
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.
});
```

Sztring:

```
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

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

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

Exceptions

```
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 the exact error message or a regexp
  expect(compileAndroidCode).toThrow('you are using the wrong JDK');
  expect(compileAndroidCode).toThrow(/JDK/);
});
```

Testing Asynchronous Code

Callbacks

```
// Don't do this!
test('the data is peanut butter', () => {
  function callback(data) {
    expect(data).toBe('peanut butter');
  }
  fetchData(callback);
});
```

A probléma az, hogy a teszt befejeződik a fetchData() befejeződése előtt. Egy másik megoldás, ahol egy done objektuma van a teszt függvénynek. Ekkor a jest vár, míg a done callback meg nem hívódik.

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

  fetchData(callback);
});
```

Ha a done() sosem hívódik meg, akkor elbukik a teszt, timeout error-ral. Az expect() bukik el, akkor hibát dob, és a done() nem hívódik meg.

Promises

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

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

`.resolves` / `.rejects`

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

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

Async/Await

```
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');
  }
});
```

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

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

Setup and Teardown

Repeating Setup For Many Tests

Ha valamit minden teszt előtt/után le kell futtatni akkor `beforeEach` / `afterEach használata szükséges:`

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

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

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

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

One-Time Setup

Ha valamit egyszer kell lefuttatni a tesztek előtt/után:

```
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();
});
```

Scoping

Ha egy decribe blokkba tesszük a kódunkat, akkor az ott lévő before after részek a blokkon belül (a blokk tesztjeire) lesznek érvényesek:

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

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

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

describe('matching cities to foods', () => {
  // 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);
  });
});
```

```
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
```

Mock Functions

Using a mock function

Képzeljük el, hogy egy olyan függvény megvalósítását teszteljük, amelyet forEach készít, a mellékelt tömb minden elemére meghívja a callback()-et.

```
function forEach(items, callback) {
  for (let index = 0; index < items.length; index++) {
    callback(items[index]);
  }
}
```

```
const mockCallback = jest.fn(x => 42 + x);
forEach([0, 1], mockCallback);

// The mock function is called twice
expect(mockCallback.mock.calls.length).toBe(2);

// The first argument of the first call to the function was 0
expect(mockCallback.mock.calls[0][0]).toBe(0);

// The first argument of the second call to the function was 1
expect(mockCallback.mock.calls[1][0]).toBe(1);

// The return value of the first call to the function was 42
expect(mockCallback.mock.results[0].value).toBe(42);
```

`.mock` property

Minden modellhez tartozik ez a speciális .mock property, ami tárolja hogyan lett meghívva a függvény, és mivel tért vissza.

```
const myMock = jest.fn();

const a = new myMock();
const b = {};
const bound = myMock.bind(b);
bound();
```

```
console.log(myMock.mock.instances);
```

```
// The function was called exactly once
expect(someMockFunction.mock.calls.length).toBe(1);

// The first arg of the first call to the function was 'first arg'
expect(someMockFunction.mock.calls[0][0]).toBe('first arg');

// The second arg of the first call to the function was 'second arg'
expect(someMockFunction.mock.calls[0][1]).toBe('second arg');

// The return value of the first call to the function was 'return value'
expect(someMockFunction.mock.results[0].value).toBe('return value');

// This function was instantiated exactly twice
expect(someMockFunction.mock.instances.length).toBe(2);

// The object returned by the first instantiation of this function
// had a `name` property whose value was set to 'test'
expect(someMockFunction.mock.instances[0].name).toEqual('test');
```

Mock Return Values

```
const myMock = jest.fn();
console.log(myMock());
// > undefined

myMock
  .mockReturnValueOnce(10)
  .mockReturnValueOnce('x')
  .mockReturnValue(true);

console.log(myMock(), myMock(), myMock(), myMock());
// > 10, 'x', true, true
```

```
const filterTestFn = jest.fn();

// Make the mock return `true` for the first call,
// and `false` for the second call
filterTestFn.mockReturnValueOnce(true).mockReturnValueOnce(false);

const result = [11, 12].filter(num => filterTestFn(num));

console.log(result);
// > [11]
console.log(filterTestFn.mock.calls);
// > [ [11], [12] ]
```

Mock Names

```
const myMockFn = jest
  .fn()
  .mockReturnValue('default')
  .mockImplementation(scalar => 42 + scalar)
  .mockName('add42');
```

```
// The mock function was called at least once
expect(mockFunc).toHaveBeenCalled();

// The mock function was called at least once with the specified args
expect(mockFunc).toHaveBeenCalledWith(arg1, arg2);

// The last call to the mock function was called with the specified args
expect(mockFunc).toHaveBeenLastCalledWith(arg1, arg2);

// All calls and the name of the mock is written as a snapshot
expect(mockFunc).toMatchSnapshot();
```

```
/ The mock function was called at least once
expect(mockFunc.mock.calls.length).toBeGreaterThan(0);

// The mock function was called at least once with the specified args
expect(mockFunc.mock.calls).toContainEqual([arg1, arg2]);

// The last call to the mock function was called with the specified args
expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1]).toEqual([
  arg1,
  arg2,
]);

// The first arg of the last call to the mock function was `42`
// (note that there is no sugar helper for this specific of an assertion)
expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1][0]).toBe(42);

// A snapshot will check that a mock was invoked the same number of times,
// in the same order, with the same arguments. It will also assert on the name.
expect(mockFunc.mock.calls).toEqual([[arg1, arg2]]);
expect(mockFunc.getMockName()).toBe('a mock name');
```

An Async Example

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

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

```
// request.js
const http = require('http');

export default function request(url) {
  return new Promise(resolve => {
    // This is an example of an http request, for example to fetch
    // user data from an API.
    // This module is being mocked in __mocks__/request.js
    http.get({path: url}, response => {
      let data = '';
      response.on('data', _data => (data += _data));
      response.on('end', () => resolve(data));
    });
  });
}
```

```
// __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.',
          }),
    );
  });
}
```

```
// __tests__/user-test.js
jest.mock('../request');

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

// The assertion for a promise must be returned.
it('works with promises', () => {
  expect.assertions(1);
  return user.getUserName(4).then(data => expect(data).toEqual('Mark'));
});
```

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

```
// async/await can be used.
it('works with async/await', async () => {
  expect.assertions(1);
  const data = await user.getUserName(4);
  expect(data).toEqual('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.toEqual('Paul');
});
```

```
// Testing for async errors using Promise.catch.
it('tests error with promises', () => {
  expect.assertions(1);
  return user.getUserName(2).catch(e =>
    expect(e).toEqual({
      error: 'User with 2 not found.',
    }),
  );
});

// Or using async/await.
it('tests error with async/await', async () => {
  expect.assertions(1);
  try {
    await user.getUserName(1);
  } catch (e) {
    expect(e).toEqual({
      error: 'User with 1 not found.',
    });
  }
});
```

```
// Testing for async errors using `.rejects`.
it('tests error with rejects', () => {
  expect.assertions(1);
  return expect(user.getUserName(3)).rejects.toEqual({
    error: 'User with 3 not found.',
  });
});

// Or using async/await with `.rejects`.
it('tests error with async/await and rejects', async () => {
  expect.assertions(1);
  await expect(user.getUserName(3)).rejects.toEqual({
    error: 'User with 3 not found.',
  });
});
```

Timer Mocks

```
// timerGame.js
'use strict';

function timerGame(callback) {
  console.log('Ready....go!');
  setTimeout(() => {
    console.log("Time's up -- stop!");
    callback && callback();
  }, 1000);
}

module.exports = timerGame;
```

```
// __tests__/timerGame-test.js
'use strict';

jest.useFakeTimers();

test('waits 1 second before ending the game', () => {
  const timerGame = require('../timerGame');
  timerGame();

  expect(setTimeout).toHaveBeenCalledTimes(1);
  expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 1000);
});
```

Run All Timers

```
test('calls the callback after 1 second', () => {
  const timerGame = require('../timerGame');
  const callback = jest.fn();

  timerGame(callback);

  // At this point in time, the callback should not have been called yet
  expect(callback).not.toBeCalled();

  // Fast-forward until all timers have been executed
  jest.runAllTimers();

  // Now our callback should have been called!
  expect(callback).toBeCalled();
  expect(callback).toHaveBeenCalledTimes(1);
});
```

Run Pending Timers

```
// infiniteTimerGame.js
'use strict';

function infiniteTimerGame(callback) {
  console.log('Ready....go!');

  setTimeout(() => {
    console.log("Time's up! 10 seconds before the next game starts...");
    callback && callback();

    // Schedule the next game in 10 seconds
    setTimeout(() => {
      infiniteTimerGame(callback);
    }, 10000);
  }, 1000);
}

module.exports = infiniteTimerGame;
```

```
// __tests__/infiniteTimerGame-test.js
'use strict';

jest.useFakeTimers();

describe('infiniteTimerGame', () => {
  test('schedules a 10-second timer after 1 second', () => {
    const infiniteTimerGame = require('../infiniteTimerGame');
    const callback = jest.fn();

    infiniteTimerGame(callback);

    // At this point in time, there should have been a single call to
    // setTimeout to schedule the end of the game in 1 second.
    expect(setTimeout).toHaveBeenCalledTimes(1);
    expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 1000);

    // Fast forward and exhaust only currently pending timers
    // (but not any new timers that get created during that process)
    jest.runOnlyPendingTimers();

    // At this point, our 1-second timer should have fired it's callback
    expect(callback).toBeCalled();

    // And it should have created a new timer to start the game over in
    // 10 seconds
    expect(setTimeout).toHaveBeenCalledTimes(2);
    expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 10000);
  });
});
```

ES6 Class Mocks

An ES6 Class Example

```
// sound-player.js
export default class SoundPlayer {
  constructor() {
    this.foo = 'bar';
  }

  playSoundFile(fileName) {
    console.log('Playing sound file ' + fileName);
  }
}
```

```
// sound-player-consumer.js
import SoundPlayer from './sound-player';

export default class SoundPlayerConsumer {
  constructor() {
    this.soundPlayer = new SoundPlayer();
  }

  playSomethingCool() {
    const coolSoundFileName = 'song.mp3';
    this.soundPlayer.playSoundFile(coolSoundFileName);
  }
}
```

Automatic mock

A `jest.mock('./sound-player') segítségével:`

```
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';
jest.mock('./sound-player'); // SoundPlayer is now a mock constructor

beforeEach(() => {
  // Clear all instances and calls to constructor and all methods:
  SoundPlayer.mockClear();
});

it('We can check if the consumer called the class constructor', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  expect(SoundPlayer).toHaveBeenCalledTimes(1);
});

it('We can check if the consumer called a method on the class instance', () => {
  // Show that mockClear() is working:
  expect(SoundPlayer).not.toHaveBeenCalled();

  const soundPlayerConsumer = new SoundPlayerConsumer();
  // Constructor should have been called again:
  expect(SoundPlayer).toHaveBeenCalledTimes(1);

  const coolSoundFileName = 'song.mp3';
  soundPlayerConsumer.playSomethingCool();

  // mock.instances is available with automatic mocks:
  const mockSoundPlayerInstance = SoundPlayer.mock.instances[0];
  const mockPlaySoundFile = mockSoundPlayerInstance.playSoundFile;
  expect(mockPlaySoundFile.mock.calls[0][0]).toEqual(coolSoundFileName);
  // Equivalent to above check:
  expect(mockPlaySoundFile).toHaveBeenCalledWith(coolSoundFileName);
  expect(mockPlaySoundFile).toHaveBeenCalledTimes(1);
});
```

Manual mock

```
// __mocks__/sound-player.js

// Import this named export into your test file:
export const mockPlaySoundFile = jest.fn();
const mock = jest.fn().mockImplementation(() => {
  return {playSoundFile: mockPlaySoundFile};
});

export default mock;
```

```
// sound-player-consumer.test.js
import SoundPlayer, {mockPlaySoundFile} from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';
jest.mock('./sound-player'); // SoundPlayer is now a mock constructor

beforeEach(() => {
  // Clear all instances and calls to constructor and all methods:
  SoundPlayer.mockClear();
  mockPlaySoundFile.mockClear();
});

it('We can check if the consumer called the class constructor', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  expect(SoundPlayer).toHaveBeenCalledTimes(1);
});

it('We can check if the consumer called a method on the class instance', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  const coolSoundFileName = 'song.mp3';
  soundPlayerConsumer.playSomethingCool();
  expect(mockPlaySoundFile).toHaveBeenCalledWith(coolSoundFileName);
});
```

Calling `jest.mock()` with the module factory parameter

```
jest.mock(path, moduleFactory)
```

```
import SoundPlayer from './sound-player';
const mockPlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: mockPlaySoundFile};
  });
});
```

Replacing the mock using `mockImplementation()` or `mockImplementationOnce()`

```
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';

jest.mock('./sound-player');

describe('When SoundPlayer throws an error', () => {
  beforeAll(() => {
    SoundPlayer.mockImplementation(() => {
      return {
        playSoundFile: () => {
          throw new Error('Test error');
        },
      };
    });
  });

  it('Should throw an error when calling playSomethingCool', () => {
    const soundPlayerConsumer = new SoundPlayerConsumer();
    expect(() => soundPlayerConsumer.playSomethingCool()).toThrow();
  });
});
```

In depth: Understanding mock constructor functions

Manual mock that is another ES6 class

```
// __mocks__/sound-player.js
export default class SoundPlayer {
  constructor() {
    console.log('Mock SoundPlayer: constructor was called');
  }

  playSoundFile() {
    console.log('Mock SoundPlayer: playSoundFile was called');
  }
}
```

Mock using module factory parameter

```
jest.mock('./sound-player', () => {
  return function() {
    return {playSoundFile: () => {}};
  };
});
```

```
jest.mock('./sound-player', () => {
  return () => {
    // Does not work; arrow functions can't be called with new
    return {playSoundFile: () => {}};
  };
});
```

Keeping track of usage (spying on the mock)

```
import SoundPlayer from './sound-player';
jest.mock('./sound-player', () => {
  // Works and lets you check for constructor calls:
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: () => {}};
  });
});
```

Spying on methods of our class

```
import SoundPlayer from './sound-player';
const mockPlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: mockPlaySoundFile};
    // Now we can track calls to playSoundFile
  });
});
```

```
// __mocks__/sound-player.js

// Import this named export into your test file
export const mockPlaySoundFile = jest.fn();
const mock = jest.fn().mockImplementation(() => {
  return {playSoundFile: mockPlaySoundFile};
});

export default mock;
```

Cleaning up between tests

```
beforeEach(() => {
  SoundPlayer.mockClear();
  mockPlaySoundFile.mockClear();
});
```

Complete example

```
// sound-player-consumer.test.js
import SoundPlayerConsumer from './sound-player-consumer';
import SoundPlayer from './sound-player';

const mockPlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: mockPlaySoundFile};
  });
});

beforeEach(() => {
  SoundPlayer.mockClear();
  mockPlaySoundFile.mockClear();
});

it('The consumer should be able to call new() on SoundPlayer', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  // Ensure constructor created the object:
  expect(soundPlayerConsumer).toBeTruthy();
});

it('We can check if the consumer called the class constructor', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  expect(SoundPlayer).toHaveBeenCalledTimes(1);
});

it('We can check if the consumer called a method on the class instance', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  const coolSoundFileName = 'song.mp3';
  soundPlayerConsumer.playSomethingCool();
  expect(mockPlaySoundFile.mock.calls[0][0]).toEqual(coolSoundFileName);
});
```

Using with MongoDB

```
yarn add @shelf/jest-mongodb -dev
```

2. Specify preset in your Jest configuration:

```
{
  "preset": "@shelf/jest-mongodb"
}
```

```
const {MongoClient} = require('mongodb');
```

```javascript
describe('insert', () => {
  let connection;
  let db;

  beforeAll(async () => {
    connection = await MongoClient.connect(global.__MONGO_URI__, {
      useNewUrlParser: true,
    });
    db = await connection.db(global.__MONGO_DB_NAME__);
  });

  afterAll(async () => {
    await connection.close();
    await db.close();
  });

  it('should insert a doc into collection', async () => {
    const users = db.collection('users');

    const mockUser = {_id: 'some-user-id', name: 'John'};
    await users.insertOne(mockUser);

    const insertedUser = await users.findOne({_id: 'some-user-id'});
    expect(insertedUser).toEqual(mockUser);
  });
});
```