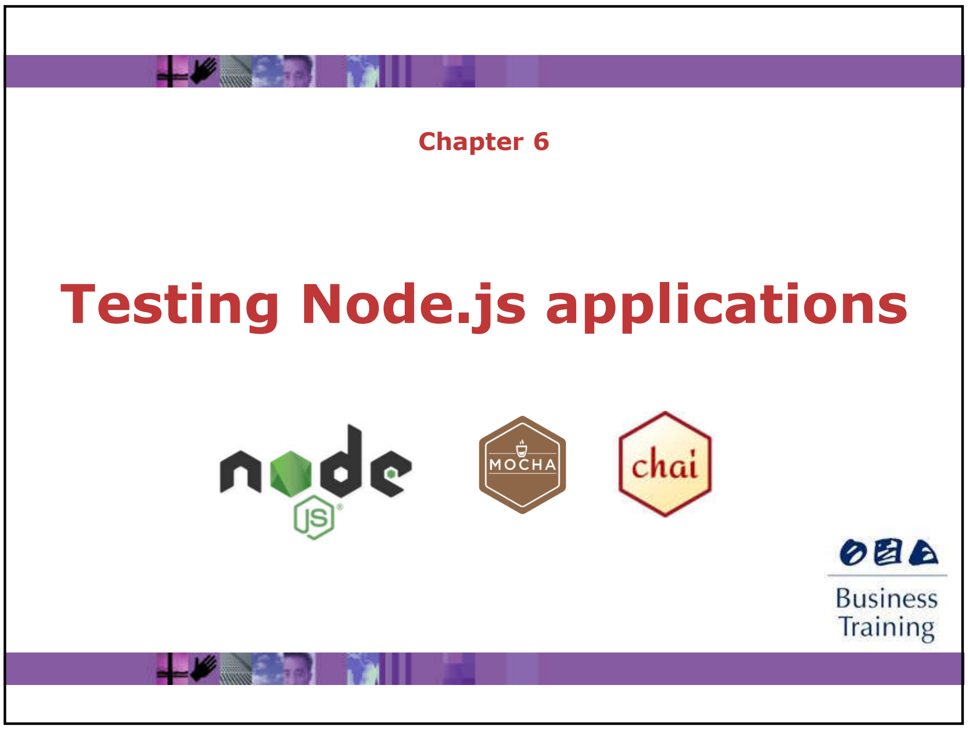
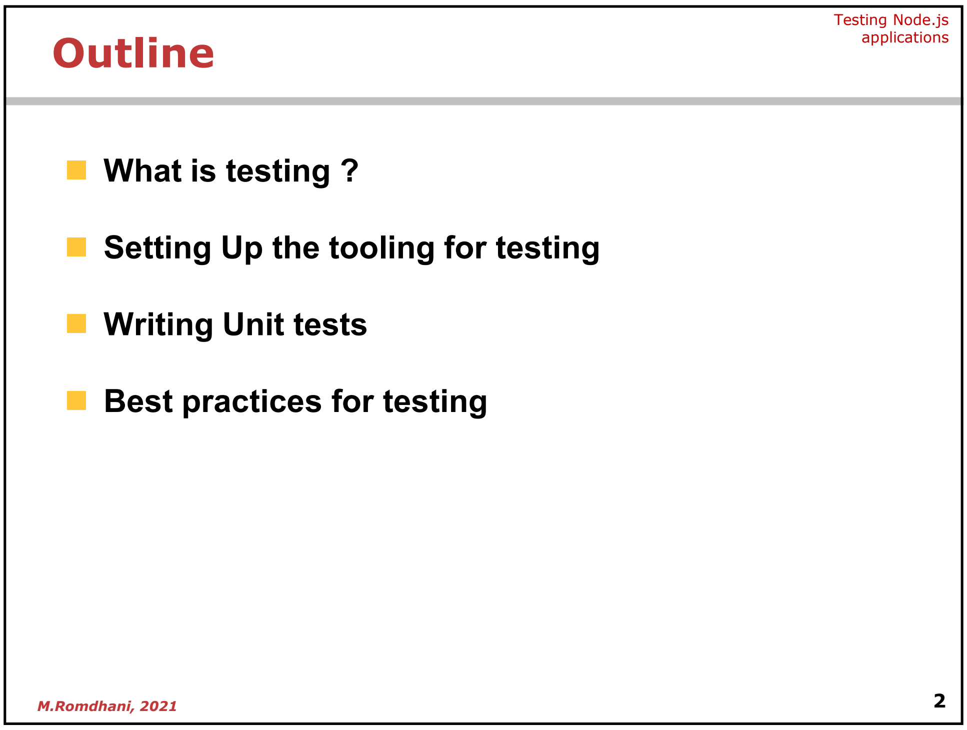


Chapter 6

Testing Node.js applications

Business Training

1



Outline

- What is testing ?
- Setting Up the tooling for testing
- Writing Unit tests
- Best practices for testing

M.Romdhani, 2021 2

2

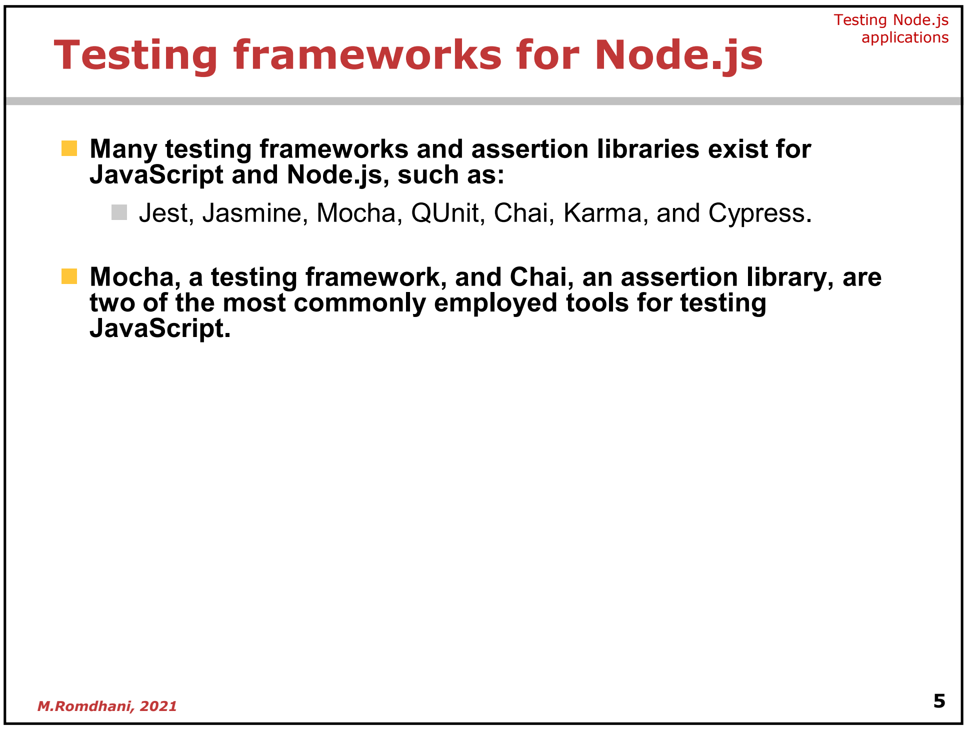
What is testing ?

3

Why testing is important ?

- **With the aid of Test-Driven Development (TDD), software engineers can build strong, adaptable code that is less likely to error when additional features are included.**
- **Types of Testing**
  - Unit tests
    - test individual 'units' of code such as functions, methods, modules, or classes. These tests are, computationally, very cheap and can be performed quickly.
  - Integration testing
    - Integration testing tests across multiple, different modules to ensure they successfully work together.
  - E2E testing
    - It operates within the browser (or a browser-like environment) to replicate the experience a user would have with the application — think scrolling, clicking, navigating through the app, typing, etc.

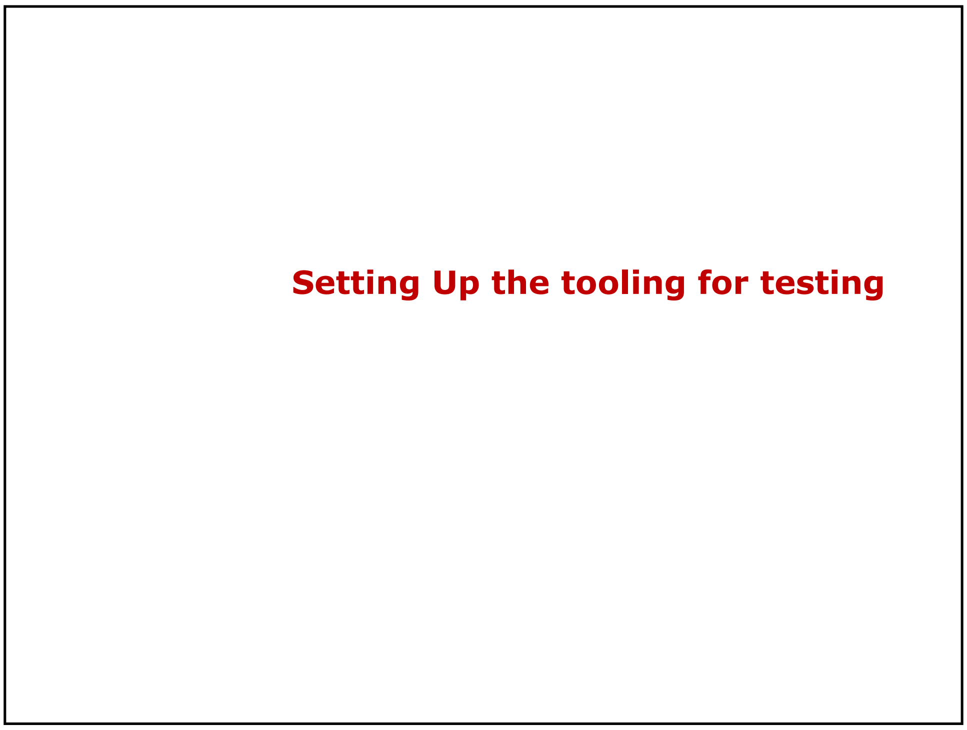
4



Testing frameworks for Node.js

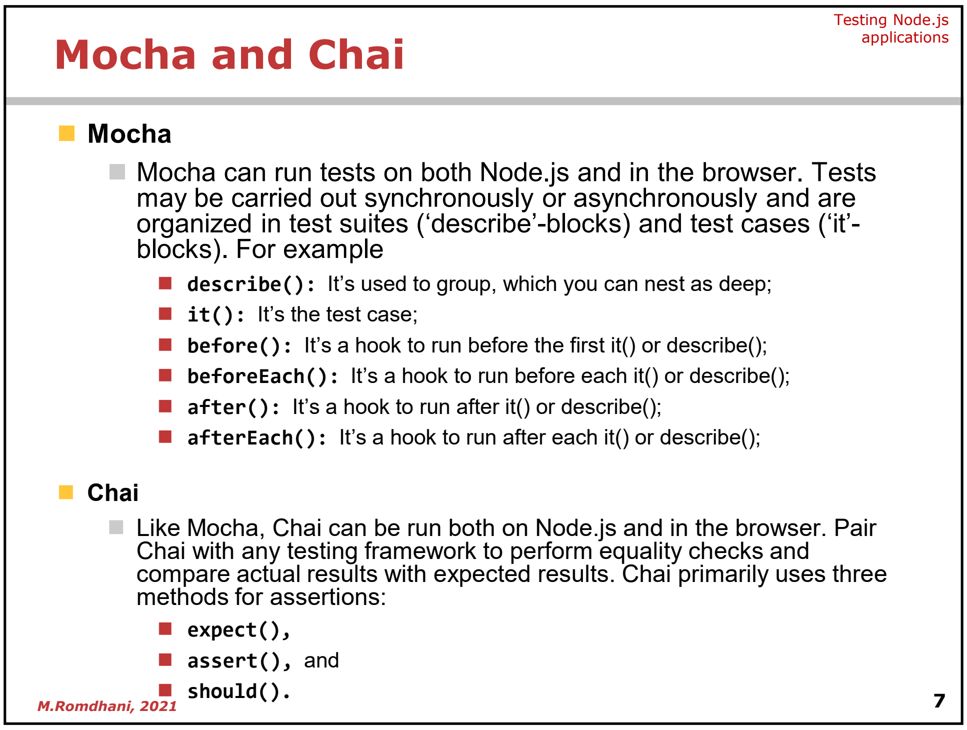
- **Many testing frameworks and assertion libraries exist for JavaScript and Node.js, such as:**
  - Jest, Jasmine, Mocha, QUnit, Chai, Karma, and Cypress.
- **Mocha, a testing framework, and Chai, an assertion library, are two of the most commonly employed tools for testing JavaScript.**

5



Setting Up the tooling for testing

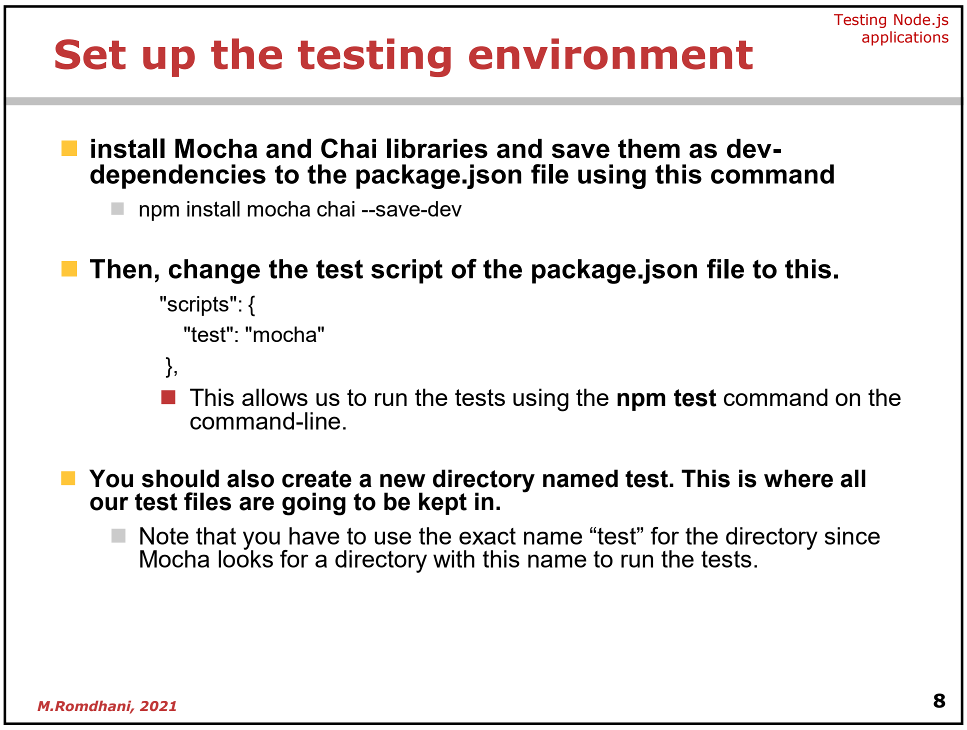
6



Mocha and Chai

- **Mocha**
  - Mocha can run tests on both Node.js and in the browser. Tests may be carried out synchronously or asynchronously and are organized in test suites (‘describe’-blocks) and test cases (‘it’-blocks). For example
    - `describe():` It’s used to group, which you can nest as deep;
    - `it():` It’s the test case;
    - `before():` It’s a hook to run before the first it() or describe();
    - `beforeEach():` It’s a hook to run before each it() or describe();
    - `after():` It’s a hook to run after it() or describe();
    - `afterEach():` It’s a hook to run after each it() or describe();
- **Chai**
  - Like Mocha, Chai can be run both on Node.js and in the browser. Pair Chai with any testing framework to perform equality checks and compare actual results with expected results. Chai primarily uses three methods for assertions:
    - `expect(),`
    - `assert(),` and
    - `should().`

7



Set up the testing environment

- **install Mocha and Chai libraries and save them as dev-dependencies to the package.json file using this command**
  - npm install mocha chai --save-dev
- **Then, change the test script of the package.json file to this.**

```
"scripts": {
   "test": "mocha"
 },
```

  - This allows us to run the tests using the **npm test** command on the command-line.
- **You should also create a new directory named test. This is where all our test files are going to be kept in.**
  - Note that you have to use the exact name “test” for the directory since Mocha looks for a directory with this name to run the tests.

8

Writing tests

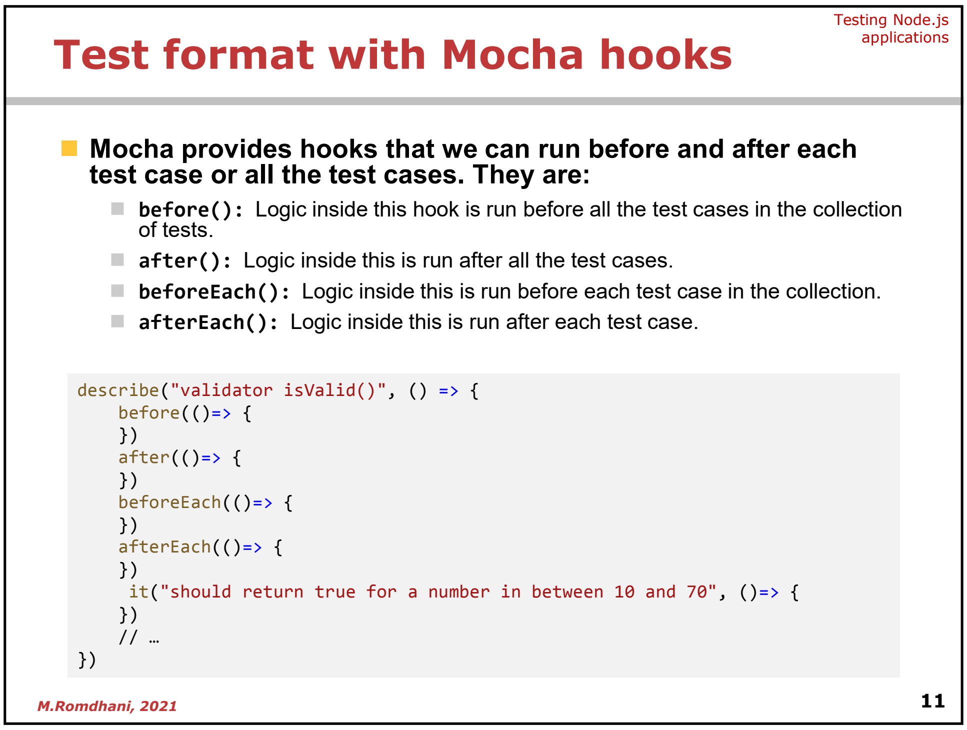
9

Basic test format of mocha

- **A typical test file in Mocha takes the following form.**

```
//a collection of test cases that test a specific component
describe("validator isValid()", () => {

    //test a function for a specific case
    it("should return true for a number in between 10 and 70", ()=> {

    })

    it("should return false when the number is less than or equal to 10", () => {

    })

    it("should return false when the number is greater than or equal to 70", () => {

    })
})
```

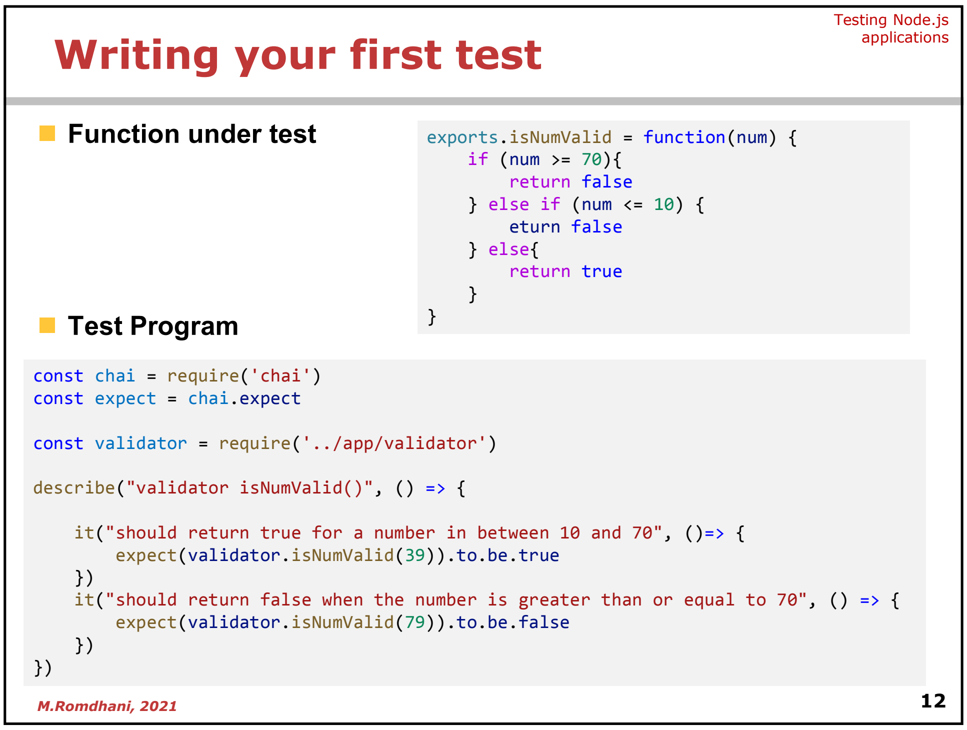
10



Test format with Mocha hooks

- **Mocha provides hooks that we can run before and after each test case or all the test cases. They are:**
  - **`before():`** Logic inside this hook is run before all the test cases in the collection of tests.
  - **`after():`** Logic inside this is run after all the test cases.
  - **`beforeEach():`** Logic inside this is run before each test case in the collection.
  - **`afterEach():`** Logic inside this is run after each test case.

```
describe("validator isValid()", () => {
    before(()=> {
    })
    after(()=> {
    })
    beforeEach(()=> {
    })
    afterEach(()=> {
    })
     it("should return true for a number in between 10 and 70", ()=> {
    })
    // …
})
```



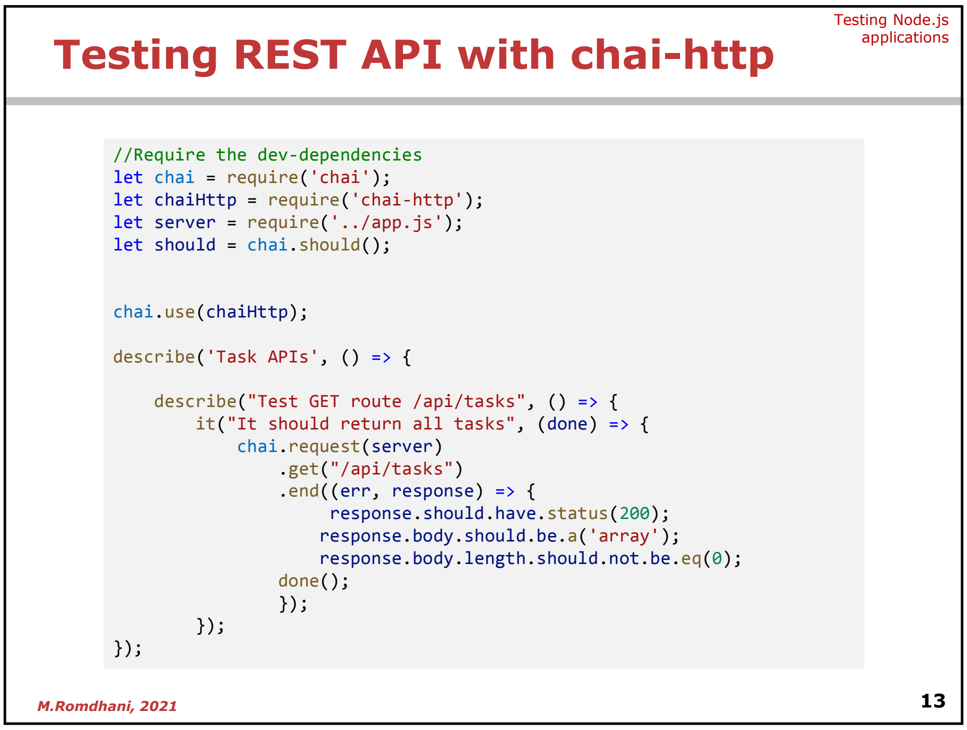
Writing your first test

- **Function under test**

```
exports.isNumValid = function(num) {
    if (num >= 70){
        return false
    } else if (num <= 10) {
        eturn false
    } else{
        return true
    }
}
```

- **Test Program**

```
const chai = require('chai')
const expect = chai.expect

const validator = require('../app/validator')

describe("validator isNumValid()", () => {

    it("should return true for a number in between 10 and 70", ()=> {
        expect(validator.isNumValid(39)).to.be.true
    })
    it("should return false when the number is greater than or equal to 70", () => {
        expect(validator.isNumValid(79)).to.be.false
    })
})
```



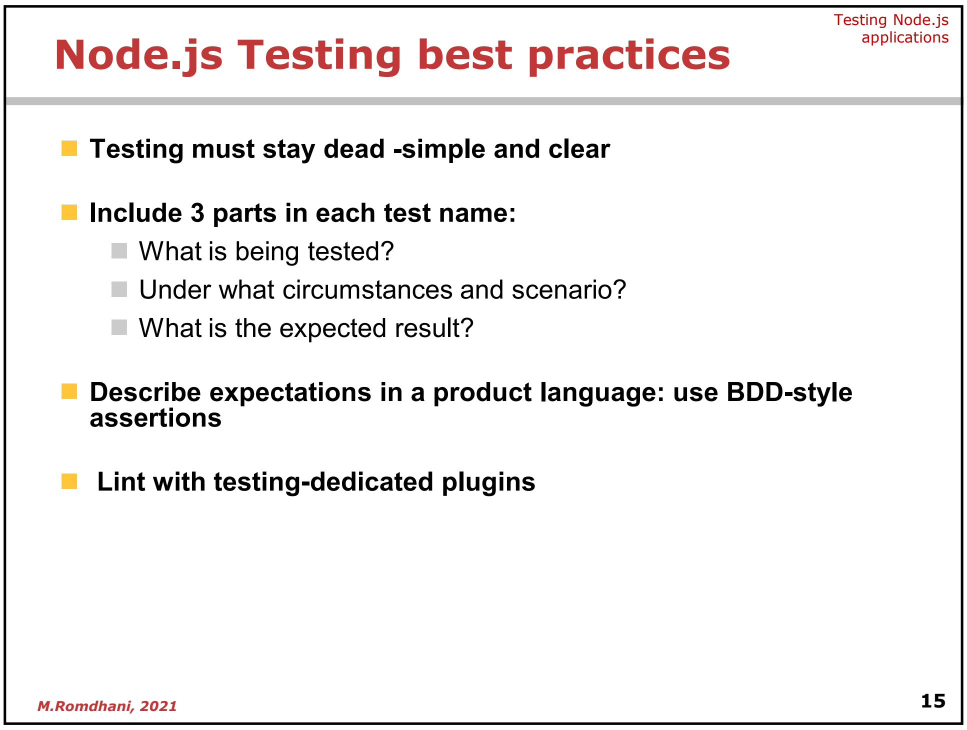
Testing REST API with chai-http

```
//Require the dev-dependencies
let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../app.js');
let should = chai.should();


chai.use(chaiHttp);

describe('Task APIs', () => {

    describe("Test GET route /api/tasks", () => {
        it("It should return all tasks", (done) => {
            chai.request(server)
                .get("/api/tasks")
                .end((err, response) => {
                     response.should.have.status(200);
                    response.body.should.be.a('array');
                    response.body.length.should.not.be.eq(0);
                done();
                });
        });
});
```

13

Best practices for testing

14



Testing Node.js applications

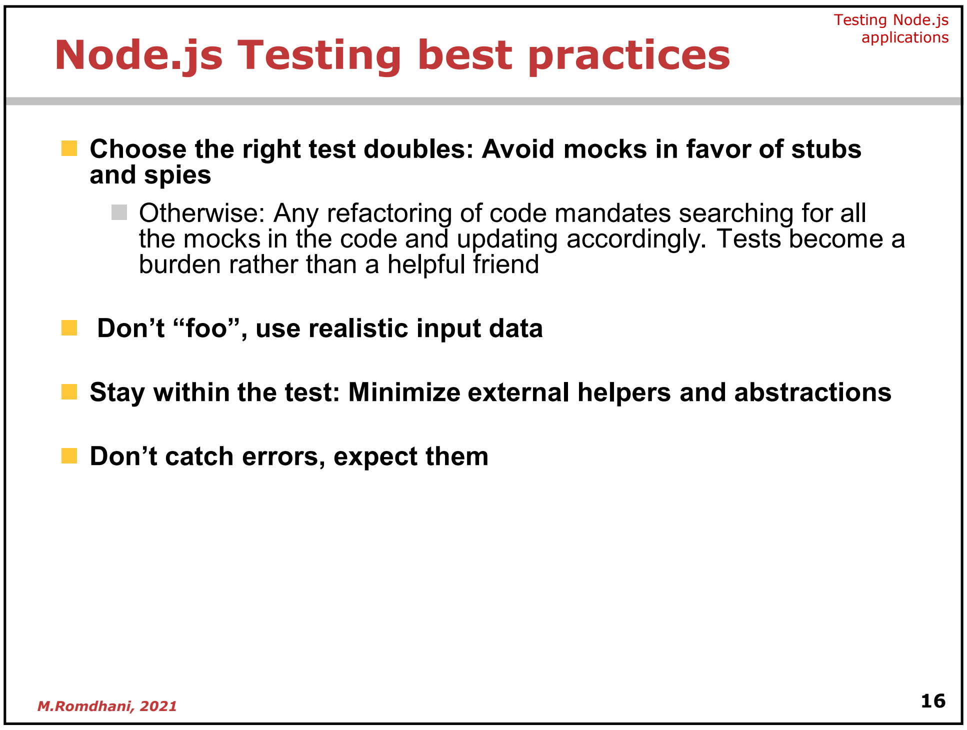
Node.js Testing best practices

- **Testing must stay dead -simple and clear**
- **Include 3 parts in each test name:**
  - What is being tested?
  - Under what circumstances and scenario?
  - What is the expected result?
- **Describe expectations in a product language: use BDD-style assertions**
- **Lint with testing-dedicated plugins**

M.Romdhani, 2021

15

15



Testing Node.js applications

Node.js Testing best practices

- **Choose the right test doubles: Avoid mocks in favor of stubs and spies**
  - Otherwise: Any refactoring of code mandates searching for all the mocks in the code and updating accordingly. Tests become a burden rather than a helpful friend
- **Don't "foo", use realistic input data**
- **Stay within the test: Minimize external helpers and abstractions**
- **Don't catch errors, expect them**

M.Romdhani, 2021

16

16