NTNU – Trondheim
Norwegian University of
Science and Technology

IT2810 Web Development

Tutorial: Testing & linting

Methods and tools for producing professional software in teams

---

Eirik Fosse
Sondre Hjetland
Michael McMillan

November 18, 2016

Introduction

In this tutorial we will dive into the world of testing and linting. In the first chapter we start off with some arguments for why you should test your code. After this we go through some simple steps on how to configure a testing environment using Mocha and Node.js. Finally we present an example demonstrating the usefulness of testing your code.

In the second chapter we explain the concept of linting your code. We will explain what it is, and why you should use a linter. We will also help you configure a linter for your project, and set up rules that your code must pass to satisfy the linter. At last we demonstrate some simple code snippets and how they look before and after linting.

All the examples as well as the code presented in this tutorial is available on GitHub [3].

Contents

1 Testing

1.1 Motivation: Why bother with testing?

There is a number of reasons why collaborating on writing code is difficult. One of them is the fear associated with changing code that works. This fear comes from not knowing if a change will break any existing functionality. Ensuring that a change does not break any existing functionality entails starting the system and trying to run as many functions as possible in hopes of detecting an error. This is especially helpful if a team of developers are working together on a project, and you are not fully familiar with the entire code base.

As part of this course your web application had to fulfill 13 requirements at a minimum. Testing these requirements manually would require you to do the following after a change in the code, to confirm that everything worked as expected:

1. Start the web server ($\approx$ 5 seconds)
2. Fire up a web browser ($\approx$ 5 seconds)
3. Navigate to the web application in your browser ($\approx$ 3 seconds)
4. If necessary, log in to the application ($\approx$ 8 seconds)
5. 13$\times$Execute a requirement to see if it works ($\approx$ 10 seconds)

Total time spent $\approx$ 2 minutes and 31 seconds

It goes without saying that carrying out such a test manually takes a lot of time and patience. This unfortunate conclusion leads a lot of developers to simply pray that their code works instead of actually testing it. In turn, most will avoid changing code "that just works" even if it is desperately needed. The aim of this tutorial is to, among other things, enable you to test all your requirements in less than 300 milliseconds.

1.2 Getting started

What you learn in this tutorial is not only applicable to JavaScript applications. In fact, most languages have equivalent libraries and tools for testing and asserting correctness of the behaviour of the code. A test has one binary outcome: Fail or pass. The latter suggests that the test succeeded while the former suggests the code did not act as expected.

1.2.1 Installing a test runner: Mocha 3.1.2

To run the tests you will need a test runner. The test runner is responsible for finding all the tests you have written, running them and finally reporting their results. We will use a popular test runner called Mocha [1].

```
# Assuming you already have an npm project initialized
npm install mocha --save-dev
```

This command will install Mocha and conveniently include it as a dependency in your `package.json`.

1.2.2 Set up the test runner

Ideally you only want to type a single command to run the tests. `npm` lets you define custom scripts in your `package.json` that lets you run the tests by typing `npm test` in your terminal. Your `package.json` should look something like this.

```
{
  "name": "your-project-name",
  "scripts": {
    "test": "mocha test/"
  },
  "devDependencies": {
    "mocha": "^3.1.2"
  }
}
```

1.2.3 Writing your first test

A convention in software is to create a separate directory for the tests. Go ahead and create a directory called `test`. To ensure that the test runner works correctly we will create a test inside the `test` directory called `testTrue.js`.

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

describe('True', () => {
  it('should equal true', () => {
    assert.equal(true, true);
  });
});
```

`assert` is a library that is included in `node` (and almost all other programming languages). It will raise an exception, thus failing our test, if the condition is not met. We will always use `assert` to determine if our program behaves as expected.

`describe` and `it` are functions Mocha provides for writing the tests. They have no significant meaning other than providing structure to the test report. We will revisit these functions later.

1.2.4 Obtaining the test report

Finally we can start the test runner to see which tests fails and passes. It does not take a major in logic to understand that we expect the test to pass: True should be equal to true, it is after all a tautology. Start the test runner by typing `npm test` anywhere inside your project.

```
$ npm test
  True
    ✓ should equal true
  1 passing (8ms)
```

The output from the above command tells you that the test passed and that it took 8 milliseconds to run it. If you amend `testTrue.js` by changing the assertion to `assert.equal(true, false);` the test runner will yield the following report.

```
$ npm test
  True
    1) should equal true
  0 passing (12ms)
  1 failing

  1) True should equal true:
      AssertionError: true == false
      + expected - actual

      at Context.it (test/testTrue.js:5:12)
```

The report indicates that the test failed and directs our attention to `test/testTrue.js:5:12`. The two numbers trailing the file name (separated by colon) tells us that the assertion threw an exception at line 5 and character index 12.

1.3 Test-driven development

A popular methodology for writing tests is test-driven development (TDD). TDD has three rules [2] which on its face may seem counter intuitive. However, these three rules lock you in to a cycle that forces you to write code that is easily testable. Easily testable code implies that the design is decoupled and that it has sensible abstractions.

1. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
2. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.
3. You are not allowed to write any production code unless it is to make a failing unit test pass.

1.4 Example: Extracting places from tweets

Let's imagine you have been hired to create a system that extracts names of places from tweets on Twitter. The system has to regularly pull down tweets from an account called `@oslopolitiops` [4] and try to extract places contained in it's tweets.

1.4.1 Requirements

For example, the system should find "Brugata" given the the first tweet, and "Holmenkollveien" and "Frognerveien" given the second tweet.

(a) Brugata

(b) Holmenkollveien & Frognerveien

1.4.2 Write the first failing test

Abiding by the first law of TDD, we need to write a failing test before we write any code. This can be the most difficult and simultaneously the most fun part of the TDD method. Since there is no source code yet, we have to imagine how we want the system to look like. Create a file called `testPlaceExtractor.js` in the `test` directory.

```
1  const assert = require('assert');
2
3  describe('extractPlacesFromTweet', () => {
4    it('should return "Brugata" given a Tweet referencing Brugata', () => {
5      const tweet = new Tweet(
6        'To personer har blitt frastjålet gjenstander i Brugata.' +
7        'Gj.person ble sprayet med forsvarsspray i ansiktet. Vi ' +
8        'ser etter en rødmalt mann.'
9      );
10     const places = extractPlacesFromTweet(tweet);
11     assert.deepEqual(places, ['Brugata']);
12   });
13 });
```

In the test above we simply imagine that a class called `Tweet` exists. Presumably it represents a tweet. Furthermore, we imagine that a function called `extractPlacesFromTweet` returns an array of all the places it found. Finally we assert that the array returned only contains "Brugata".

1.4.3 Run the first test

We expect this test to fail: There is no class called `Tweet` nor a function called `extractPlacesFromTweet`.

```
$ npm test
  extractPlacesFromTweet
    1) should extract "Brugata" from a Tweet referencing Brugata
      0 passing (10ms)
      1 failing

  1) extractPlacesFromTweet should extract "Brugata" from a Tweet
      referencing Brugata:
     ReferenceError: Tweet is not defined
      at Context.it (test/testExtractPlacesFromTweet.js:5:23)
```

1.4.4 Change the error message

Abiding by the second law of TDD, we must now only write enough production code to change the error message reported by the test runner. The error that failed the test was `ReferenceError`: "Tweet is not defined". So let's define a Tweet class to change the error message. Create a file called `tweet.js` in the root of your project and require it in your `testPlaceExtractor.js`.

```
class Tweet {

}

module.exports = Tweet;
```

Run the test again. You will find that the error message has indeed changed after inspecting the output: `ReferenceError`: "extractPlacesFromTweet is not defined".

1.4.5 Change the error message again

The test is still failing, so we need to continue adding a minimal amount of code until the test passes. To fix this error we can create a file called `extractPlacesFromTweet.js` in the root of the project and include it from the test.

```
function extractPlacesFromTweet() {

}

module.exports = extractPlacesFromTweet;
```

Run the test again and observe that we now get
`AssertionError: "undefined == [ 'Brugata' ]"`.

1.4.6 Pass the first test

Finally it is the assertion that fails the test. To make the assertion succeed simply "hard code" the return statement of `extractPlacesFromTweet.js` to return an array containing the string "Brugata". Remember: The minimal amount of code to change the error message. The `extractPlacesFromTweet` function will then look something like this.

```
function extractPlacesFromTweet() {
  return ['Brugata'];
}

module.exports = extractPlacesFromTweet;
```

Run the test again.

```
$ npm test
  extractPlacesFromTweet
    ✓ should extract "Brugata" from a Tweet referencing Brugata
  1 passing (9ms)
```

1.4.7 Write the second failing test

To pass the previous test we naively hard coded the expected return value. It goes without saying that this will not work in the general case. In other words, the stupidity of the solution forces us to write a new failing test. Append the following case to the `testExtractPlacesFromTweet.js`. For simplicity in this report, `[...]` is substituted for the first test we wrote , but in your code you should keep the test.

```
const assert = require('assert');
const Tweet = require('../tweet');
const extractPlacesFromtweet = require('../extractPlacesFromtweet');

describe('extractPlacesFromTweet', () => {

  [ ... ]

  it('should return "Holmenkollveien" & "Frognerveien" given a Tweet with both',
      () => {
    const tweet = new Tweet(
      'Politiet er på vei til Holmenkollveien, nesten oppe ved ' +
      'Frognerveien hvor en bil skal ha kjøt ut i grøfta. Skal ' +
      'ikke være personskade.'
    );
    const places = extractPlacesFromTweet(tweet);
    assert.deepEqual(places, ['Holmenkollveien', 'Frognerveien']);
  });
});
```

1.4.8 Run the second test

When we run the second test we expect it to fail. Ideally ”Holmenkollveien” and ”Frognerveien” should be returned, instead we only get ”Brugata”.

```
$ npm test
  extractPlacesFromTweet
    ✓ should extract "Brugata" from a Tweet referencing Brugata
    1) should return "Holmenkollveien" & "Frognerveien" given a Tweet with both

  1) extractPlacesFromTweet should return "Holmenkollveien" & "Frognerveien" given
       a Tweet with both:

      AssertionError: ['Brugata'] deepEqual ['Holmenkollveien', 'Frognerveien']
      + expected - actual
```

1.4.9 Try to pass the second test

To pass this test we can return an array with ”Brugata” if ”Brugata” is contained in the tweet. In all other cases we can simply return an array with ”Holmenkollveien” and ”Frognerstra”.

```
function extractPlacesFromTweet(tweet) {
  if (tweet.text.includes('Brugata')) {
    return ['Brugata'];
  }
  return ['Holmenkollveien', 'Frognerveien'];
}

module.exports = extractPlacesFromTweet;
```

1.4.10 Changing the error message

Running the test yields the following error.

```
$ npm test
  extractPlacesFromTweet
    1) should extract "Brugata" from a Tweet referencing Brugata
    2) should return "Holmenkollveien" & "Frognerveien" given a Tweet with both

  1) extractPlacesFromTweet should extract "Brugata" from a Tweet referencing
      Brugata:
     TypeError: Cannot read property 'includes' of undefined

  2) extractPlacesFromTweet should return "Holmenkollveien" & "Frognerveien" given
       a Tweet with both:
     TypeError: Cannot read property 'includes' of undefined
```

1.4.11 Pass the second test

Because we are trying to access the `tweet.text` the code blows up. This is because the `Tweet` class has no field called `text`. To fix this simply add a constructor argument to the `Tweet` class with a corresponding field.

```
class Tweet {
  constructor(text) {
    this.text = text;
  }
}

module.exports = Tweet;
```

Run the tests again.

```
$ npm test
  extractPlacesFromTweet
    ✓ should extract "Brugata" from a Tweet referencing Brugata
    ✓ should return "Holmenkollveien" & "Frognerveien" given a Tweet with
        both
  2 passing (9ms)
```

1.4.12 Write the third test

Yet again we are forced to write another failing test. This poses an interesting question: "What input will our `extractPlacesFromTweet` not handle?". If a tweet contains Brugata and Holmenkollveien, both should be returned.

```
const assert = require('assert');
const Tweet = require('../tweet');
const extractPlacesFromtweet = require('../extractPlacesFromtweet');

describe('extractPlacesFromTweet', () => {
  [ ... ]
  [ ... ]
  it('should return "Brugata" & "Holmenkollveien" given a Tweet with both', () =>
      {
    const tweet = new Tweet(
      'Politiet er på vei til Holmenkollveien, nesten oppe ved ' +
      'Brugata hvor en bil skal ha kjøt ut i grøfta. Skal ' +
      'ikke være personskade.'
    );
    const places = extractPlacesFromTweet(tweet);
    assert.deepEqual(places, ['Holmenkollveien', 'Brugata']);
  });
});
```

Run the test again and observe the output.

```
$ npm test
 extractPlacesFromTweet
   ✓ should extract "Brugata" from a Tweet referencing Brugata
   ✓ should return "Holmenkollveien" & "Frognerveien" given a Tweet with both
   3) should return "Brugata" & "Holmenkollveien" given a Tweet with both
      AssertionError: [ 'Brugata' ] deepEqual [ 'Holmenkollveien', 'Brugata' ]
      + expected - actual
       [
      +  "Holmenkollveien"
         "Brugata"
       ]
```

The output confirms our suspicion: The `extractPlacesFromTweet` did not behave as expected, which in turn means that the code is not clever enough.

1.4.13 Pass the third test

To pass this test we can iterate over all the words and return the ones that contain "veien" or "gata".

```
function extractPlacesFromTweet(tweet) {
  const places = [];
  const words = tweet.text.split(' ');
  words.forEach(word => {
    if (word.includes('veien') || word.includes('gata')) {
      const place = word.replace('.', '').replace(',', '');
      places.push(place);
    }
  });
  return places;

module.exports = extractPlacesFromTweet;
```

Run the tests to confirm that the code works.

```
$ npm test
  extractPlacesFromTweet
    ✓ should extract "Brugata" from a Tweet referencing Brugata
    ✓ should return "Holmenkollveien" & "Frognerveien" given a Tweet with
        both
    ✓ should return "Brugata" & "Holmenkollveien" given a Tweet with both

  3 passing (9ms)
```

1.4.14 Refactor

3/3 tests are now passing. At this point we may want to improve our code, this is called "refactoring". And we can do so with confidence provided by the tests.

```
function extractPlacesFromTweet(tweet) {
  return tweet.text.split(' ')
  .filter(word => word.includes('veien') || word.includes('gata'))
  .map(place => place.replace(',', '').replace('.', ''));
}

module.exports = extractPlacesFromTweet;
```

Run the tests to confirm that it still works.

```
$ npm test
  extractPlacesFromTweet
    ✓ should extract "Brugata" from a Tweet referencing Brugata
    ✓ should return "Holmenkollveien" & "Frognerveien" given a Tweet with
        both
    ✓ should return "Brugata" & "Holmenkollveien" given a Tweet with both

  3 passing (9ms)
```

1.4.15 Conclusion

As promised in the introduction, we have managed to test the system in 9 milliseconds. The extraction of place names may still not be good enough, but by iterating and adding more tests and more code, we can verify that place names from all our previous test tweets are correctly extracted. Even though there is a lot of pieces missing, the core business logic of extracting places has a solid foundation.

2 Linting

When a team of developers are working on the same project, it can be hard to keep a consistent code base. Using a linter can help you solve this problem, and additionally make it easier for you to write clean code and even discover shallow bugs. It will raise awareness towards good practises, and help you learn to write better code. In this tutorial, we will use ESLint (`http://eslint.org/`) as our linter.

2.1 Rules

When configuring a linter, it expects a set of rules it can validate your code against. The complete set of possible rules can be found here: `http://eslint.org/docs/rules/`. An example of a rule is `no-unused-vars`, which disallows the use of unused variables. You can configure the set of rules yourself, but it is easier to extend an existing set of rules. You can think of `extend` just the way you extend classes in object oriented programming; you will inherit all the rules, but you can override them.

2.2 Demonstration

AirBnb has written a very popular guide on how to write JavaScript (`https://github.com/airbnb/javascript`). Each step in this guide is associated with a linter rule. You can configure ESLint to use all of these rules by simply declaring `{ "extends": "airbnb" }` in your config file (`.eslintrc`). The config file can be generated using ESlint's initializer in your project folder. We will use the initializer in the following steps to lint our project:

1. If your are not in a npm directory, run `npm init`.
2. Install ESLint with necessary peer dependencies for using Airbnb rules. It can be problematic to get the correct peer dependencies, but if you copy the code snippet from `https://www.npmjs.com/package/eslint-config-airbnb` into your terminal, the correct versions will be installed.
3. Run `node ./node_modules/eslint/bin/eslint --init` to initialize ESLint. This will give you the option to "Use a popular style guide". You can now select Airbnb, and it will generate an `.eslintrc` file for you, with rules from Airbnb.
4. `node ./node_modules/eslint/bin/eslint [options] [file|dir|glob]*`
   If you just want to lint all files in the project directory without any other specific options, you can run; `node ./node_modules/eslint/bin/eslint ./**`

Other steps you could (should) do:

- You can add a `.eslintignore` file, specifying files and folders to ignore, such as `package.json`.
- You can add the following line to the "scripts" section of your `package.json`:
  `"lint": "eslint ./**"`
  You can now run `npm run lint` to perform the linting.
- You can install a linting package in your IDE for real-time linting (Supported in most modern IDEs).
- If you are using Github, you can also specify that all linting must pass before you can push to the master branch.

2.2.1 Lets try some examples

```
function square(number){
    var squaredNumber = number**2;
    return squaredNumber;
}
var number=2;
var squaredNumber = square(number);
console.log(squaredNumber);
```

Listing 2.1: Before linting

Seems ok, doesn't it? Well, it does not comply to the linter's standards. When running `node ./node_modules/eslint/bin/eslint ./**`, the output is this:

```
1   1:24  error  Missing space before opening brace                       space-before-blocks
2   2:5   error  Expected indentation of 2 spaces but found 4             indent
3   2:5   error  Unexpected var, use let or const instead                 no-var
4   2:9   error  'squaredNumber' is assigned a value but never used       no-unused-vars
5   2:31  error  Infix operators must be spaced                           space-infix-ops
6   3:5   error  Expected indentation of 2 spaces but found 4             indent
7   6:1   error  All 'var' declarations must be at the top of the function scope  vars-on-top
8   6:1   error  Unexpected var, use let or const instead                 no-var
9   6:11  error  Infix operators must be spaced                           space-infix-ops
10  7:1   error  All 'var' declarations must be at the top of the function scope  vars-on-top
11  7:1   error  Unexpected var, use let or const instead                 no-var
12  7:5   error  'squaredNumber' is assigned a value but never used       no-unused-vars
```

It is possible to fix some of the errors by running the ESLint command with `--fix`. We will not do this now, but rather fix error by error. Below is the same code after correcting all the errors found by linting.

```
1 function square(number) {
2   const squaredNumber = number ** 2;
3   return squaredNumber;
4 }
5 const number = 2;
6 const squaredNumber = square(number);
7 console.log(squaredNumber);
```

Listing 2.2: After linting

This might seem unnecessary, but the example is very simplistic and only dips a toe into what ESLint is offering. You are still ensuring that you keep a consistent and clean code base. Lets do some more examples.

Example showing where you should put default parameters, and how you should use string templates instead of concatenating strings:

```
function printHelloMessage(greeting = "Hello", name){
    console.log(greeting + ", my good friend " + name);
}
printHelloMessage("Bye", "Trond");
```

Listing 2.3: Before linting

```
function printHelloMessage(name, greeting = 'Hello') {
  console.log(`${greeting}, my good friend ${name}`);
}
printHelloMessage('Trond');
```

Listing 2.4: After linting

Example showing how you should loop through an array:

```
const numbers = [1,2,3,4,5];
let sum = 0;
for (let num of numbers) {
  sum += num;
}
```

Listing 2.5: Before linting

```
const numbers = [1, 2, 3, 4, 5];
let sum = 0;
numbers.forEach(num => (sum += num));
```

Listing 2.6: After linting

Finally, we recommend that you run your JavaScript code through a transpiler before deploying your frontend code. Since the rules we use requires us to use newer ECMAScript functionality, a transpiler will transpile the code so that it works in older browsers.

Bibliography

[1] TJ Holowaychuk. Mocha. https://mochajs.org. [Accessed on November 18th 2016].

[2] Robert C. Martin. The three laws of tdd. http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd. [Accessed on November 18th 2016].

[3] michaelmcmillan. Github - it2810 linting & testing. https://github.com/michaelmcmillan/IT2810-linting-and-testing. [Accessed on November 18th 2016].

[4] Operasjonssentralen Oslo. Twitter - oslopolitiops. http://twitter.com/oslopolitiops. [Accessed on November 18th 2016].