Keep calm and use Jest

Holger Grosse-Plankermann

codecentric

Whoaml?

Developer/Consultant/Whatever

Taming the Web since the 2000

Compiled Mozilla for bonus features

Podcaster http://autoweird.fm

@holgergp

http://github.com/holgergp

Holger Grosse-Plankermann

AGENDA

Where are we heading?

Unit Tests in JavaScript

Where do I come from?

Enter Jest

Why another library?

Snapshot Tests

Golden Master on Steroids

Mocking

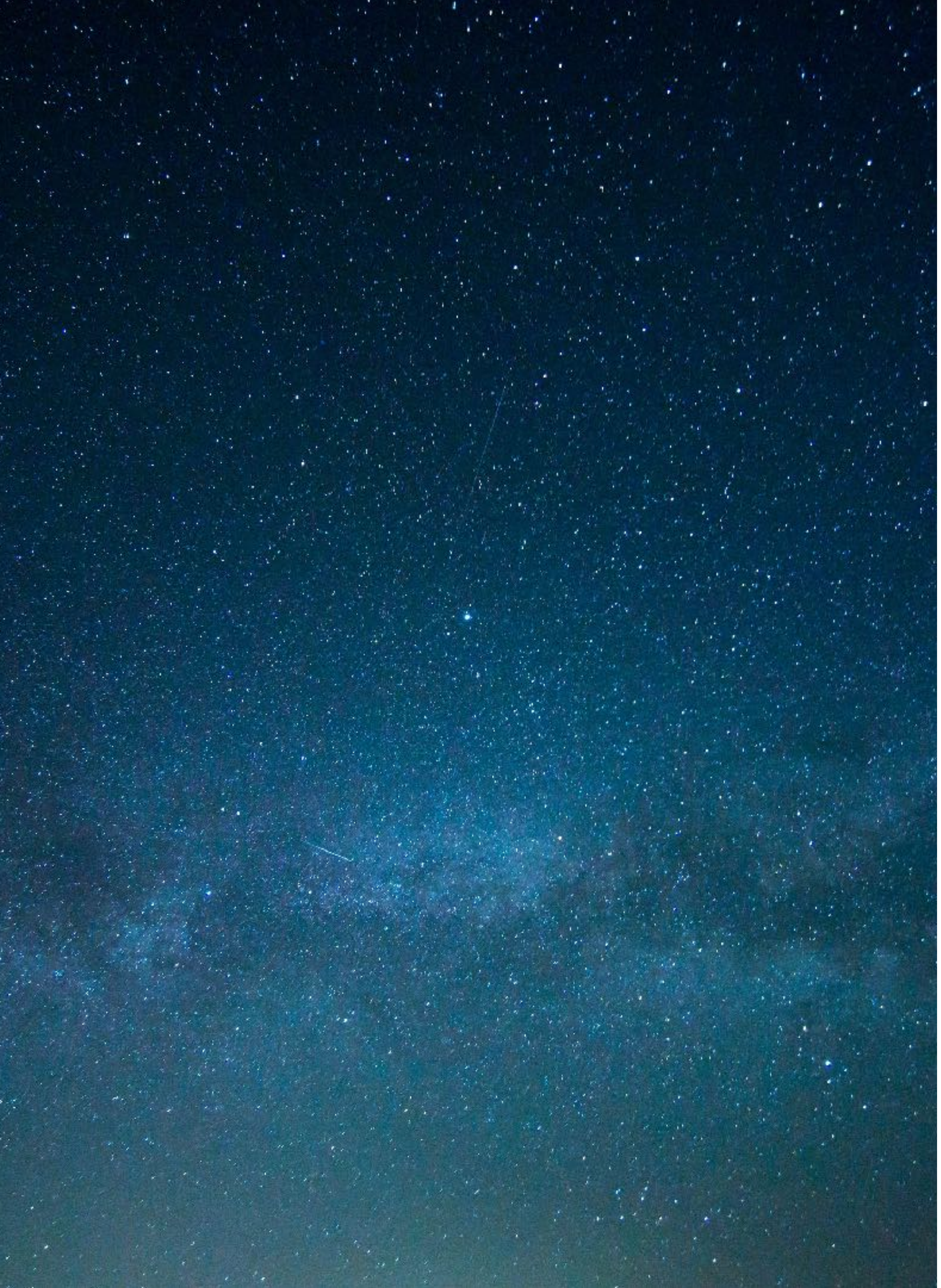
Minimizing scope

Wait there's more

Async Code, IDE Support, ….

(Unit-) Testing JS

Where do I come from



Phase 1

- JavaScript === alert(“Foo“)
- Simple Formish Web Apps
- Seemingly no need for thorough testing like in the backend
- „Selenium is enough“

Phase 2

- Everybody wanted to build the new Myspace
    - … with jQuery
- But frontend skills were still lacking
- *Let's do complicated backend stuff and finish the frontend with some sprinkles of jQuery …*
    - … And some more
    - jQuery Jenga
- Too „complicated" to properly test our frontend code

Phase 3

- In the age of SPAs the need for proper testing on the client arose
  - Alongside the explosion of the ecosystem
- Many different approaches (Hey it's Javascript!l
  - Jasmine
  - Mocha
  - QUnit
  - .....
- But: Terribly to set up!

And now?

- What is needed to setup a TDD environment for ES6?
- Anybody tried to do a GDOCR with ECMAScript?
    - How did it feel?
- Somehow testing JS felt/feels/is subpar compared to Java/Maven
- Debugging tests in an IDE is still a thing!
- Console.log is still a thing!

- But there's help!

Enter Jest

A library that puts the fun in testing

Some Jest Facts

What the fuzz?

Jest is a testing library made by Facebook. Its aim is to be ‚delightful'. Something you wouldn't normally attribute to testing

- Easy Setup
- Ease of use
- Familiar Features
- Batteries included

Let's get started

I want to write the first test!

Create-React-App

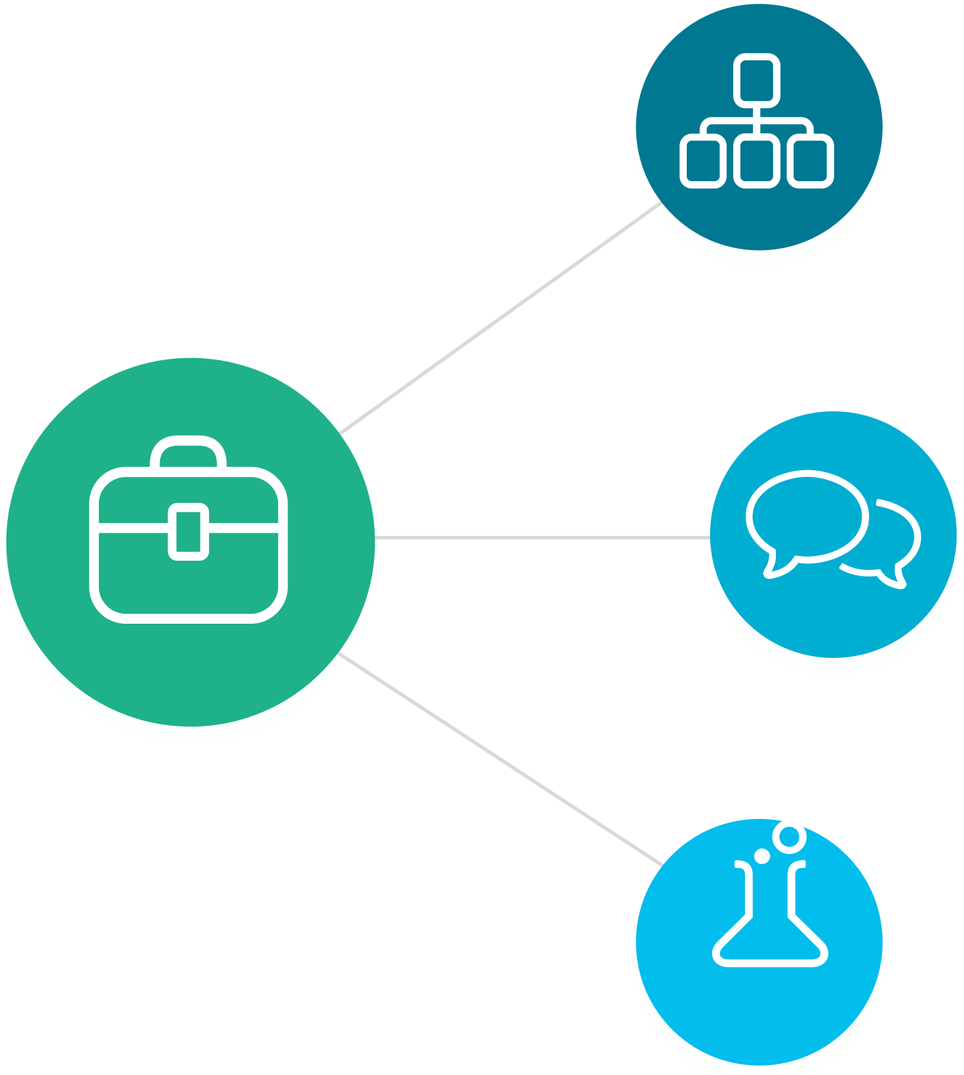
Probably the easiest way to get going is to just use create-react-app

repl.it

Use an online repl might be even easier.

npm install jest

Come on let's get our hands dirty



Let’s write our first test

Install dependencies
Let’s go simple first

Write a simple test
Very simple

Run it
In your terminal

Be amazed

```
~/d/s/t/jestStarter git:(master) 1A > yarn add -d jest jest-cli
```

```
const add = (a, b) => {
  return a + b;
}
describe('add' ,() => {
    it('should compute the sum', () => {
        expect(add(1,2)).toBe(3);
    });
});
```

```
~/d/s/t/jestStarter git:(master) 1A > jest
 PASS  ./add.test.js
  add
    ✓ should compute the sum (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.149s
Ran all test suites.
```

Let’s look at the test

```
const add = (a, b) => {
  return a + b;
}
describe('add' ,() => {
 it('should compute the sum', () => {
  expect(add(1,2)).toBe(3);
 });
});
```

Should look familiar

- Jasmine and mocha users shouldn’t be surprised
- JUnit people will see that there is some testing going on.

So no worries

Let’s look at the test some more

```
describe('add' ,() => {
 it('should compute the sum', () => {
  expect(add(1,2)).toBe(3);
 });
```

```
test('Simple Test' ,() => {
 expect(true).toBe(true);
});
```

BDD Style

This gives a nice descriptive scoped structure of your test.
As a bonus Jest can print the results nicely

AAA Style

A more traditional way of writing your tests Possible, but less expressive than the BDD style.

What can we expect?

- `.toBe(value)`
- `.toHaveBeenCalled()`
- `.toBeCloseTo(number, numDigits)`
- `.toBeDefined()`
- `.toBeFalsy()`
- `.toBeGreaterThan(number)`
- `.toBeGreaterThanOrEqual(number)`
- `.toBeLessThan(number)`
- `.toBeLessThanOrEqual(number)`
- `.toBeInstanceOf(Class)`
- `.toBeNull()`
- `.toBeTruthy()`
- `.toBeUndefined()`
- `.toContain(item)`
- `.toContainEqual(item)`
- `.toEqual(value)`
- `.toHaveLength(number)`
- `.toMatch(regexpOrString)`
- `.toMatchObject(object)`
- ...

A great number of expects is already built in

We will have a look at some more in a second

SetUp/TearDown

- `afterAll(fn)`
- `afterEach(fn)`
- `beforeAll(fn)`
- `beforeEach(fn)`
- `describe(name, fn)`
- `describe.only(name, fn)`
- `describe.skip(name, fn)`
- `test(name, fn)`
- `test.only(name, fn)`
- `test.skip(name, fn)`

All the lifecycle methods you would expect are there

and more

Ease of use

Simple to install and simple to use

The CI

Jest features a very comfortable command line interface

You almost don't miss the IDE :)

What I like most are:

Built in Watch mode

Restarts your tests when tests/source changes

Built in Code Coverage analysis

How good are you testing at your fingertips

Let's fail a test

Start Jest in Watch mode
This is built in!

Fail test!
Oh my!

Watch it fail
In your terminal

Be amazed

```
~/d/s/t/jestStarter git:(master) 1A > jest —watch
```

```
const add = (a, b) => {
  return a - b;
}
```

```
 FAIL  ./add.test.js
  add
    × should compute the sum (5ms)

  ● add › should compute the sum

    expect(received).toBe(expected)

    Expected value to be (using ===):
      3
    Received:
      -1

      at Object.<anonymous> (add.test.js:5:37)
          at Promise (<anonymous>)
          at <anonymous>
      at process._tickCallback (internal/process/next_tick.js:188:7)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        2.518s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Let’s add a feature!

- Hack some new feature
- Start Jest in coverage mode
- Watch the statistics
  In your terminal
- We need some more tests!

```
const multiply = (a,b) => {
    return a * b;
}
```

```
~/d/s/t/jestStarter git:(master) 1A > jest –coverage
```

```
~/d/s/t/jestStarter git:(master) 1M 1A > jest --coverage --verbose
 PASS  ./add.test.js
  add
    ✓ should compute the sum (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.206s, estimated 2s
Ran all test suites.
----------|----------|----------|----------|----------|----------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
All files |       75 |      100 |       50 |       75 |                |
 add.js   |       75 |      100 |       50 |       75 |              6 |
----------|----------|----------|----------|----------|----------------|
```

There is more

- Run only certain tests
  - Pick tests by name
  - Pick your test(s) by regex
  - Typeahead
- Very polished

Snapshot Testing

Golden master on steroids

What is Snapshot testing?

Golden Master on steroids!

Run your code

Capture the output

Store the output in a file

Change your code

Rerun your code

Compare result to snapshot

Simple example

- Run your code
- Capture the output
- Store the output in a file
- Change your code
- Rerun your code
- Compare result to snapshot

```
export const stringProducer = (a) => {
    return "myString" + a;
}

const myString = stringProducer('Reactjs');
       expect(myString).toMatchSnapshot();
```

```
exports[`stringProducer should prepend the given string 1`] = `"myStringReactjs"`;
```

```
export const stringProducer = (a) => {
    return "yourString" + a;
}
```

```
~/d/s/t/jestStarter      master  1M 4A > jest
 FAIL  ./foo.test.js
  ● stringProducer › should prepend the given string

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot 1.

    - "myStringReactjs"
    + "yourStringReactjs"

      at Object.<anonymous> (foo.test.js:6:26)
          at Promise (<anonymous>)
          at <anonymous>
      at process._tickCallback (internal/process/next_tick.js:188:7)

 › 1 snapshot test failed.
```

So what is this useful for?

Test that would require some effort to expect

React components (that should not change)

Things you would not test otherwise

React example

Run your code

Capture the output

Store the output in a file

Change your code

Rerun your code

Compare result to snapshot

codecentric

```
render() {
    const { value } = this.props;
    return (
      <div className="Text">
        {value}
      </div>
    );
  }
```

```
const renderedApp = renderer.create(<Text value='test'/>).toJSON();
     expect(renderedApp).toMatchSnapshot();
```

```
exports[`TextComponent renders 1`] = `
<div
  className="Text"
>
  test
</div>
`;
```

```
return (
      <div className="Text">
        <p>{value}</p>
      </div>
    );
```

```
~/d/s/t/j/jestTalkApp      master  1M 14A > jest
 FAIL  client/components/TextComponent.spec.jsx
  • TextComponent › renders

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot 1.

    - Snapshot
    + Received

      <div
        className="Text"
      >
    -   test
    +   <p>
    +     test
    +   </p>
      </div>
```

That being said

Besides snapshot tests being awesome

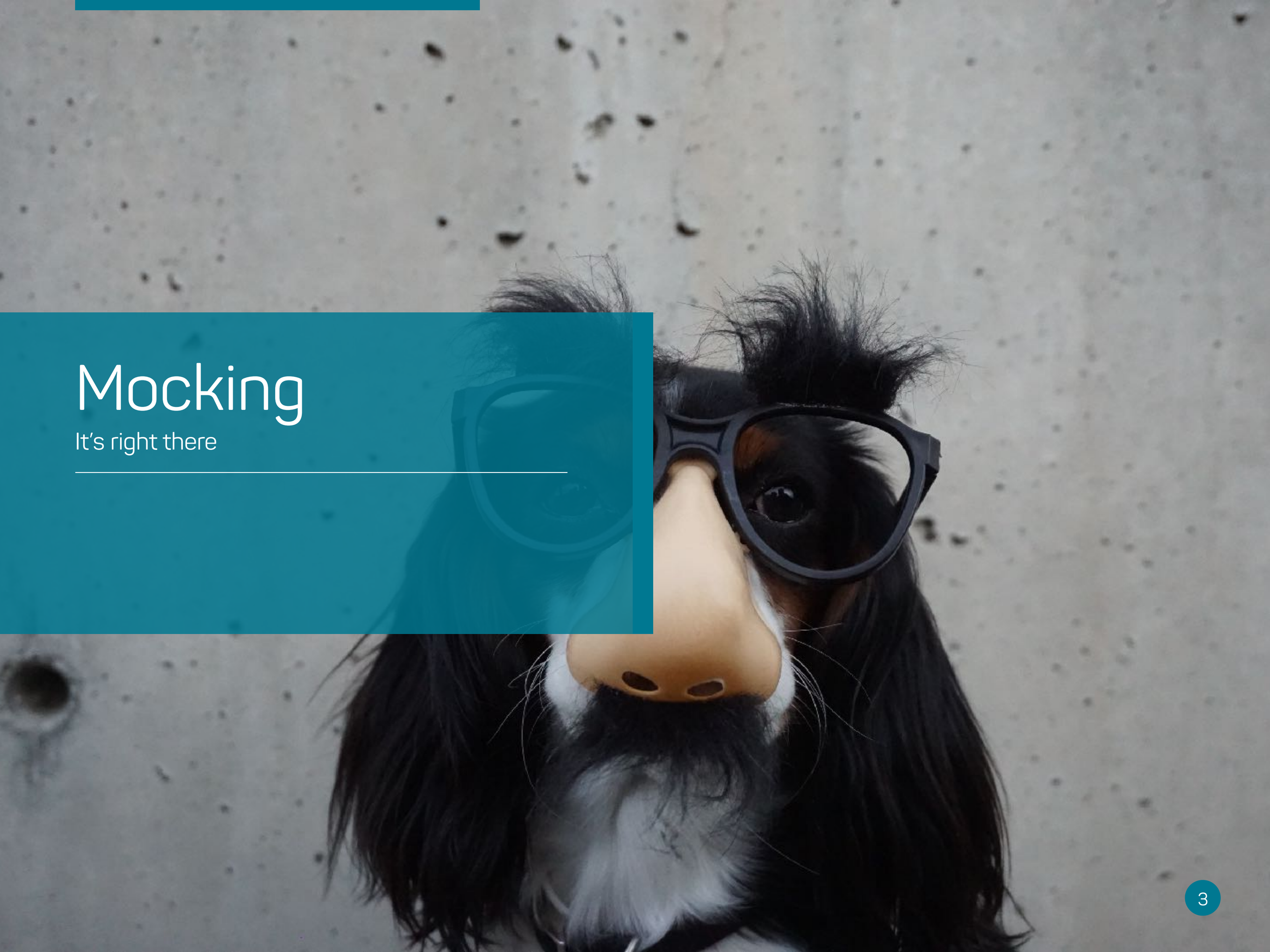
Snapshot test may be brittle

You NEED to review your snapshot changes

Prefer unit tests

Mocking

It's right there

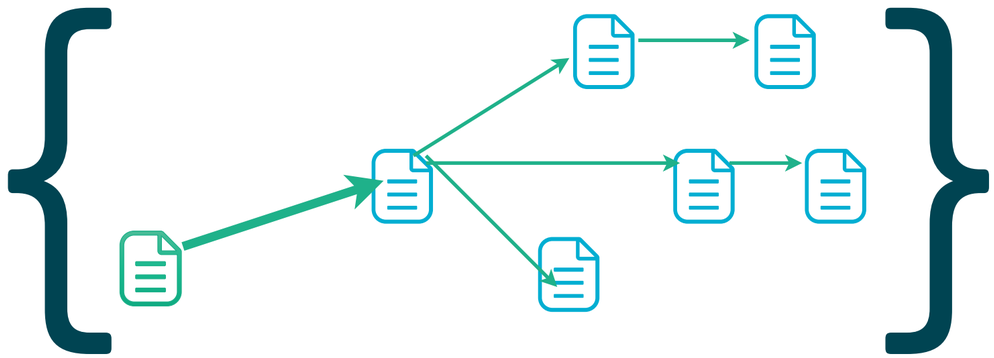


Mocking? I've seen that!

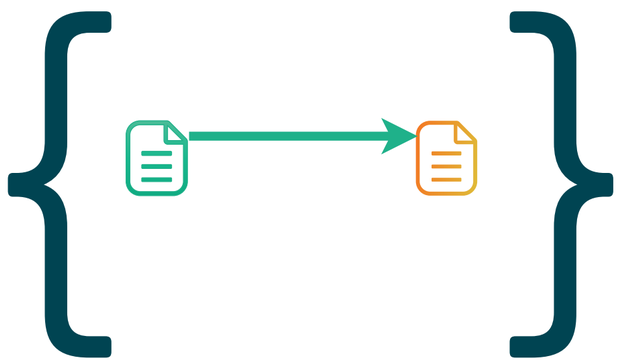
- There are already lots of great mocking libs out there
  - Sinon ftw
- There's no need to reach out for third party libs
- The integrated mocking lib will suit your needs

Mocking

In a unit test it is often unfeasible to kickstart your whole dependency graph



Mocking lets you narrow down the scope of your test



Show me more

Write here your great subtitle

Mock Functions

Lets you replace more complex logic with logic right for your test

Manual Mocks

Lets you replace whole modules for your test

Testing a complicated service

Say you want to use a service that talks over the net

But you don't want to actually test the http communication

A test might look like this

```
import {getPosts} from '../service/httpService';

export let doSomethingWithRemoteData = () => {
    const postData  = getPosts();
    return postData.posts.map(it =>it.toUpperCase());
};
```

```
  render() {
        const {posts} = this.state;
        return (
            <div style={{textAlign: 'center'}}>
                <h1>Hello World</h1>
                <List values={posts}/>
            </div>
        );
    }

    componentDidMount() {
        const posts = doSomethingWithRemoteData() || [];
        this.setState({posts});
    }
```

```
it('calling Service correctly', () => {
        dependency.doSomethingWithRemoteData = jest.fn();
        const renderedList = renderer.create(<App />).toJSON();

        expect(dependency.doSomethingWithRemoteData).toHaveBeenCalled();
        expect(dependency.doSomethingWithRemoteData).toBeCalled();

        expect(dependency.doSomethingWithRemoteData.mock.calls[0][0])
            .toBe('param');
        expect(dependency.doSomethingWithRemoteData).toBeCalledWith('param');

    });
```

Mocking return values

If you want to control the values your mock returns

```
    it('renders with defined data', () => {
        dependency.doSomethingWithRemoteData = jest.fn()
            .mockReturnValueOnce(['test']);

        const renderedApp = renderer.create(<App/>).toJSON();
        expect(renderedApp).toMatchSnapshot();
    });
```

If you want to control the implementation of your mock

```
it('renders with defined return function', () => {
        dependency.doSomethingWithRemoteData = jest.fn()
            .mockImplementation(() => (['mockImplementation']));

        const renderedApp = renderer.create(<App/>).toJSON();
        expect(renderedApp).toMatchSnapshot();
    });
```

There is more at:
https://facebook.github.io/jest/docs/en/mock-functions.html

Mocking a whole module

Let's revisit our ‚service'

```
import {getPosts} from '../service/httpService';

export let doSomethingWithRemoteData = (someParam) => {
    const postData  = getPosts();
    return postData.posts.map(it =>it.toUpperCase());
};
```

It uses a tiny ‚http lib'

```
export const getPosts = () => ({ posts: ['complicated', 'JSON', 'Response'] });
```

You can replace this with your own stub implementation

```
import { doSomethingWithRemoteData } from './serviceUser.js';

jest.mock('./httpService', () => {
  const getPostMock = () => {
    return {
      posts: ['fake', 'data']
    };
  };
  return {
    getPosts: getPostMock
  };
});

describe('serviceUser shoukd', () => {
  it('talk to backendService', () => {
    const data = doSomethingWithRemoteData();
    expect(data).toEqual(['FAKE', 'DATA']);
  });
});
```

My usage

MockFns vs. Manual mocks

- **I use MockFns when**
  - I need some fine-grained control of a dependency
  - I only need to know how and if something was called

- **I use Manual Mocks when**
  - I want to replace a whole module in a test
  - Maybe its an my way for the current test
  - I18n could be such an example

- **It's perfectly fine to mix them**
  - If it is important how a larger dependency might be called

Wait! There’s more

Some more advanced concepts

Working with ES6

Most of you would want to work with ES6 in place in your tests

We need babel for that

A .babelrc needs to be present

```
  "devDependencies": {
    "babel-core": "^6.26.0",
    "babel-jest": "^21.2.0",
    "babel-preset-env": "^1.6.1",
```

```
{
    "presets": ["env"]
}
```

Testing asynchronous code

- Let's revisit our http lib, now it's async!
- You can test this via the well known *done*
- You can just return a Promise (and skip done)
- You can use an expectation
- **You can use an async/await!**
  If you have babel in place

```
export const getPosts = () => {
    return new Promise((resolve, reject) => {
        resolve({ posts: ['complicated', 'JSON', 'Response'] });
    });
}
```

```
  it('talk to the backend using done', done => {
    getPosts().then(data => {
      expect(data).toEqual(response);
      done();
    });
  });
```

```
  it('talk to the backend using promise returns', () => {
    return getPosts().then(data => {
      expect(data).toEqual(response);
    });
  });
```

```
  it('talk to the backend using expectations', () => {
    expect(getPosts()).resolves.toEqual(response);
  });
```

```
  it('talk to the backend using async await', async () => {
    const posts = await getPosts();
    expect(posts).toEqual(response);
  });
```

Error handling asynchronous code

- Our Http Client now rejects the promise

```
export const getPostsRejecting = () => {
    return new Promise((resolve, reject) => {
        reject('myError');
    });
}
```

- You can test this via the well known *done*

```
it('reject using done', done => {
      getPostsRejecting().catch(error => {
        expect(error).toEqual('myError');
        done();
      });
    });
```

- You can just return a Promise (and skip done)

```
 it('reject using promise returns', () => {
     return getPostsRejecting().catch(error => {
       expect(error).toEqual('myError');
     });
   });
```

- You can use an expectation

```
    it('reject using expectations', () => {
      expect(getPostsRejecting()).rejects.toEqual('myError');
    });
```

- You can use an async/await!
  If you have babel in place

```
    it('reject using async await', async () => {
        try {
            const posts = await getPostsRejecting();
        }
        catch (e){
            expect(e).toEqual('myError');
        }
    });
```

codecentric

IDE Support

It just works

- **I use it mainly in IntelliJ/Webstorm and it just works**
  - Use a Run Config like any other test
  - In EAP even coverage is shown
- **Works fine in VSCode**
  - Some more config
    - Not that polished than IntelliJ
    - But fine for me
  - I use the Jest extension
- **I often use it on the command line**
  - Alongside the IDE
  - The watch feature shines there

Coming to an end

Some closing words

Topics not covered

But should have

- **Testing DOM Nodes with Enzyme**
  - If you need fine-grained expectations on the output
- **Timer tests**
  - You can tweak the time
  - Useful with setTimeout based code
- **Works with Typescript**
  - Need to have a precompiler in place

Try Jest yourself

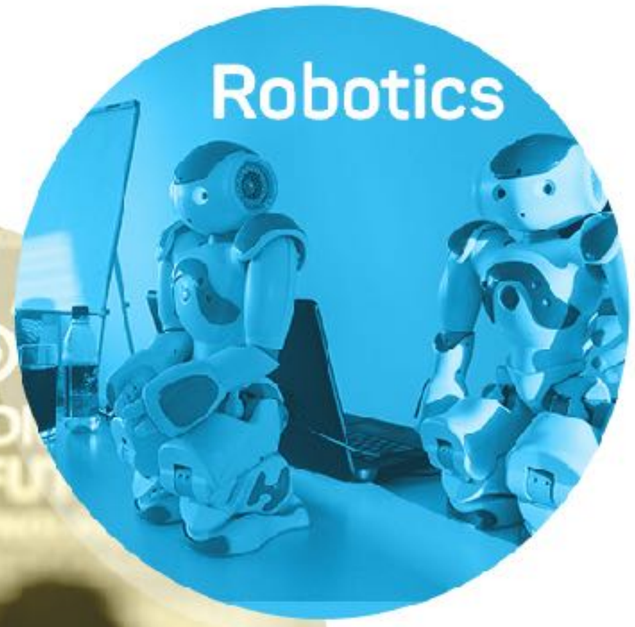
Easy to use and setup

Powerful features set yet familiar

Polished product (even the CLI)


Thanks!

Consulting and so much more



International Events

Meetups

+1 Time

Trainings

Webinars

Social Media

Writing

Knowledge Leadership

Blog

Knowledge Sharing

Stay connected

Thanks for listening!

Our mission – to promote agile development, innovation and technology – extends through everything we do.

Address

codecentric AG
Hochstraße 11
42697 Solingen

Contact Info

E-Mail: info@codecentric.de
www.codecentric.de

Telephone

Telefon: +49 (0) 212. 23 36 28 0
Telefax: +49 (0) 212.23 36 28 79

Picture Links

- https://unsplash.com/photos/TyQ-0lPp6e4
- https://unsplash.com/photos/D56TpVU8zng
- https://unsplash.com/photos/RpDA3uYkJWM
- https://unsplash.com/photos/TVKVskRlkb8
- https://unsplash.com/photos/6dvxVmG5nUU
- https://unsplash.com/photos/8xAA0f9yQnE
- https://unsplash.com/photos/TyQ-0lPp6e4
- https://unsplash.com/photos/ofUXJt4EssI
- https://unsplash.com/photos/YeAE2e2QJrM
- https://unsplash.com/photos/w0HH-NUTvVc
- https://unsplash.com/photos/QyCH5jwrD_A
- https://unsplash.com/photos/Q6jX7BbPE38
- https://unsplash.com/photos/0YbeoQ0X89k