



TDD (Test Driven Development)

Chapter 5 - Topic 2

**Selamat datang di Chapter 5 Topic 2 online course
React Native dari Binar Academy!**





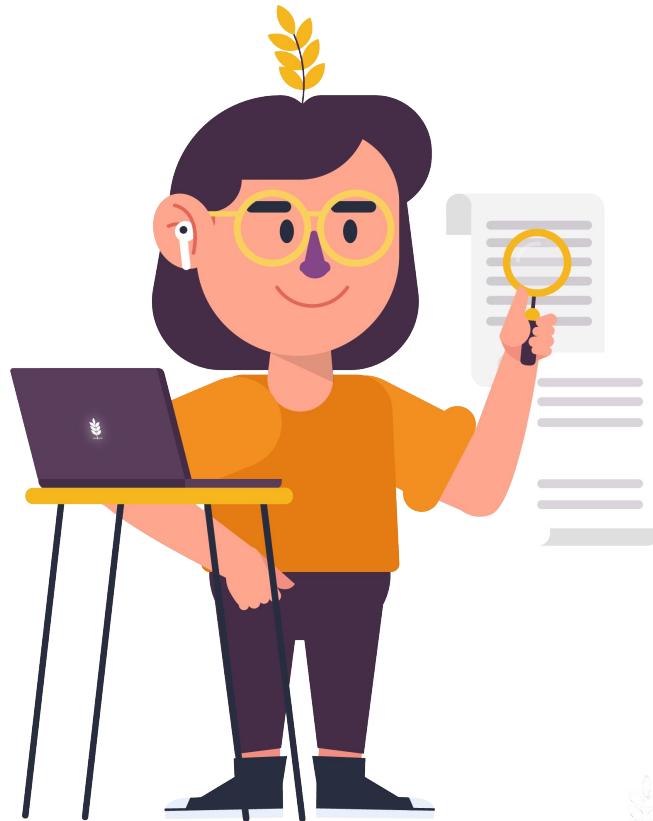
Halooo~

Ketemu lagi di Chapter 5!

Oke, di topic pertama kemarin kamu sudah belajar tentang media handling. Nahh di Chapter 5 Topic 2 hari ini, kamu akan belajar tentang TTD alias Test Driven Development.

Wah, apa tuh?

Cuci sepatu bayarnya pake poin, kalau mau tahu langsung ya kepoin!





Detailnya, kita bakal bahas hal-hal berikut ini:

1. Software Requirement
2. Test Driven Development
3. JEST
4. ENZYME





Guys, sebelum mulai, kalian perlu tahu satu hal ini nih. Pada awal pembuatan aplikasi, kamu akan diberikan **software requirements** atau daftar yang berisi spesifikasi dari aplikasi yang akan dibuat.

Hm.. untuk apa ya?





Jadi dari requirement tersebut, kamu akan menuliskan baris code yang nantinya akan menyelesaikan satu per satu spesifikasi yang diminta.





Setelah selesai menulis code, kamu bisa tes aplikasi yang telah kamu buat, apakah berjalan sesuai dengan requirement yang ada atau tidak.





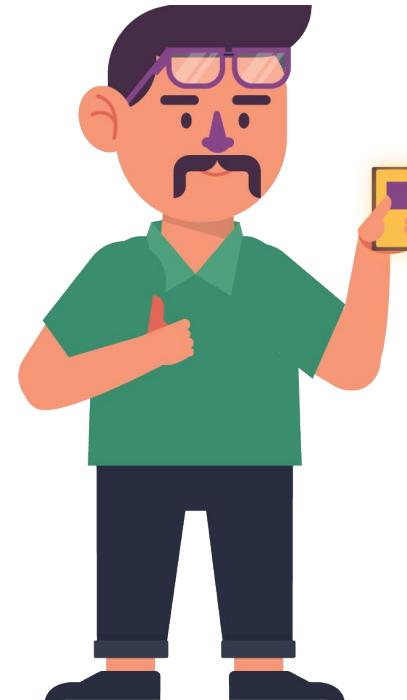
Kalau aplikasi sudah berjalan dengan baik dan benar, maka aplikasi bisa kamu berikan kepada pengguna untuk dicoba lebih jauh.





Sampai di sini kayaknya nggak ada masalah yang muncul ya. Tapi, hal yang sebelumnya kita bahas hanya dapat terjadi jika requirement yang diberikan nggak pernah berubah.

Sayangnya hal tersebut sangat jarang terjadi dalam proses pengembangan aplikasi yang sesungguhnya.





Pada era **Agile Software Development** kayak sekarang ini, requirement akan selalu berubah dengan cepat sehingga pendekatan testing secara manual dapat memunculkan beberapa masalah.

Masalah seperti apa sih? Lihat di slide selanjutnya yuk~





Masalah yang bisa muncul akibat testing manual, cekidot!

1. Perubahan pada code, penambahan fitur baru ataupun mengembangkan fitur yang sudah ada, berpotensi untuk menimbulkan bugs baru.
2. Code yang memiliki bugs sering kali nggak terdeteksi di awal, sehingga akan terus terbawa hingga ke tahap production.
3. Jika bugs muncul, maka proses untuk menemukan dan menyelesaikan bugs secara manual dapat menghabiskan waktu yang relatif lama dan menyulitkan.





Dengan adanya masalah di atas, kamu perlu skenario untuk melakukan proses pengujian (testing).

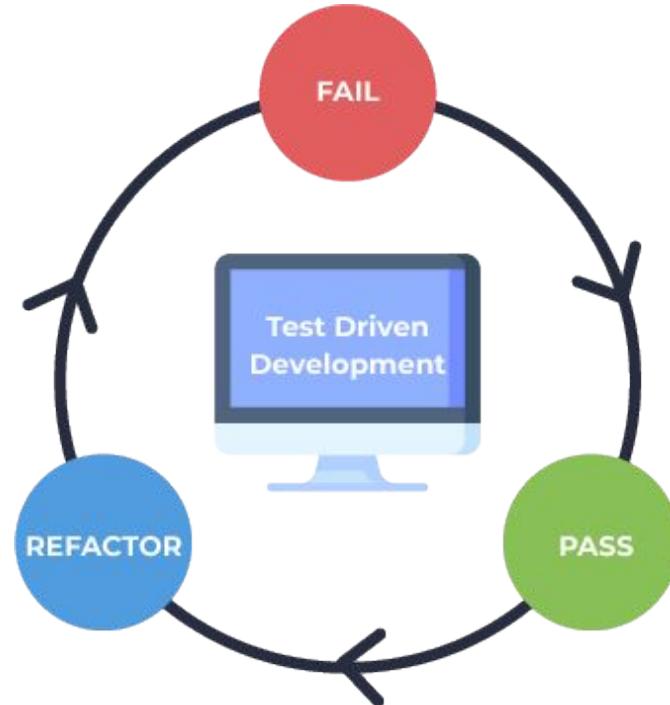
Nah, proses yang akan kamu pelajari adalah **TDD (Test Driven Development)**.





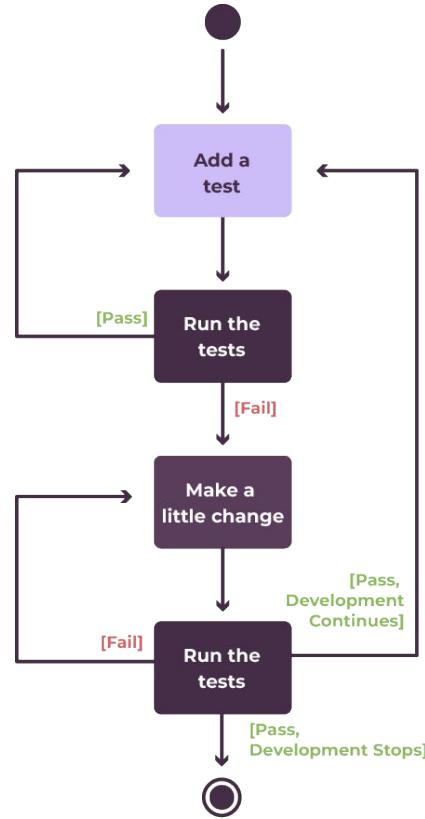
Biar familiar, yuk kenalan sama TDD dulu

TDD merupakan **proses testing yang bersifat black box**, yang pada proses pembuatannya, kamu hanya perlu menguji input dan output yang diharapkan tanpa mengetahui lebih jauh bagaimana suatu fitur akan diimplementasikan.



Bagaimana cara melakukan TDD?

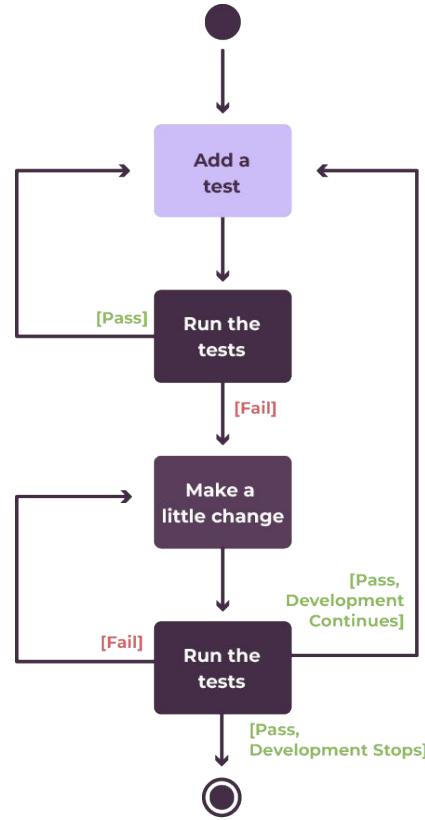
Langkah pertama adalah dengan menuliskan test lebih dulu, kemudian jalankan tes. Jika ternyata tes gagal, buatlah sedikit perubahan, kemudian coba jalankan lagi tesnya.





Lanjutannya~

Lakukan perubahan seminimal mungkin pada code agar tes yang dibuat berhasil. Jika belum berhasil, tambahkan lagi implementasi dari code hingga akhirnya tes yang telah dibuat berhasil dilewati.





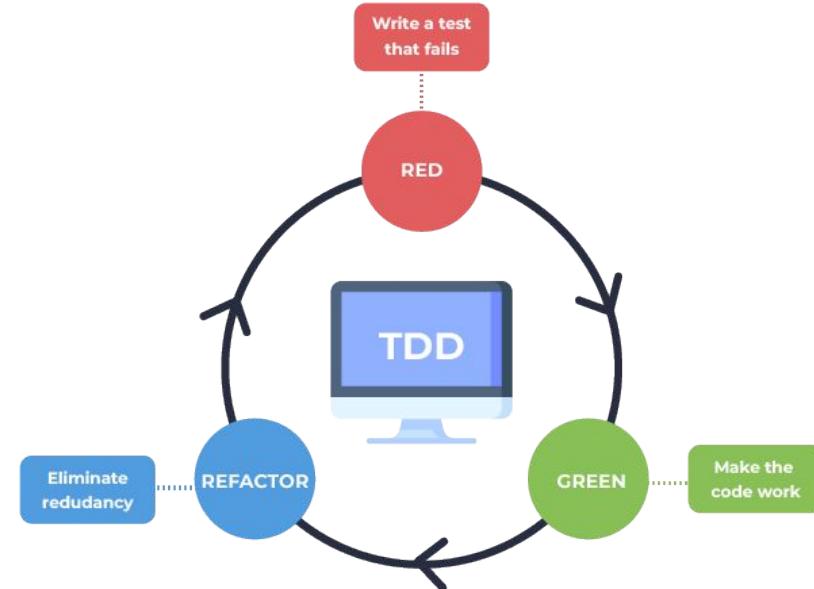
Setelah membahas secara test secara umum, selanjutnya kita akan membahas contoh penerapan dari TDD.

Are you ready?



Secara umum, TDD terdiri dari tiga siklus utama. Apa saja?

1. **RED:** Menuliskan test code dengan ukuran kecil yang membuat code menjadi gagal (fails) karena belum ada implementasinya.
2. **GREEN:** Menuliskan implementasi dari test code dengan seminimal mungkin sehingga test code dapat dijalankan dengan berhasil.
3. **REFACTOR:** Jika code yang dibuat pada proses mengimplementasikan test code masih berantakan, maka pada tahap ini code akan dirapikan dan dibersihkan.



Next, kita akan belajar tentang **Jest**, yaitu salah satu framework untuk melakukan testing pada aplikasi yang udah dibuat.

Selengkapnya tentang cara kerja Jest, yuk lanjut ke slide berikutnya~





Jest? Apa itu ya?

Jest adalah sebuah **framework pengujian berdasarkan Javascripts yang berfokus pada kesederhanaan.**

Jest dapat menguji perangkat lunak yang dibuat menggunakan Babel, TypeScript, Node, React, Angular, Vue.





Hal yang perlu kamu tahu sebelum membahas Jest lebih jauh adalah..

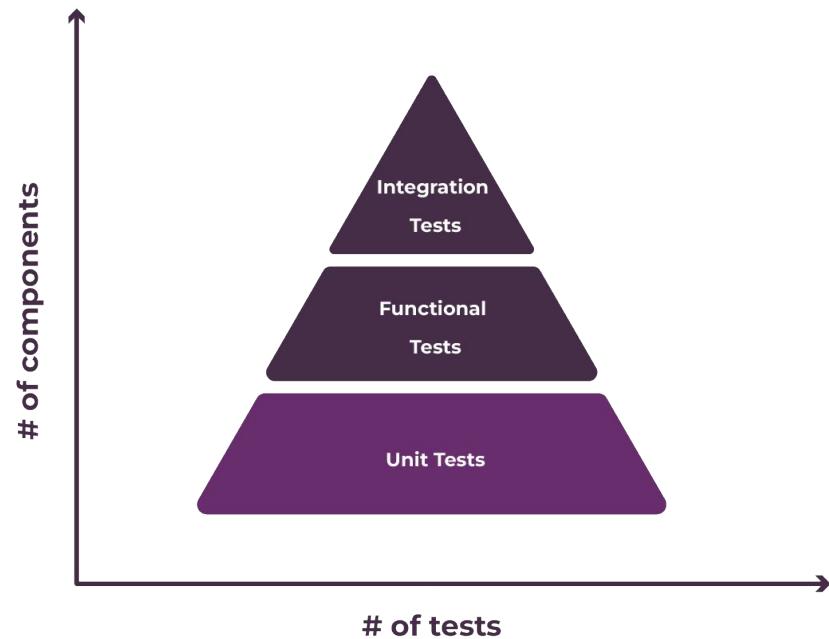
Oke, hari ini kita akan belajar untuk melakukan unit test dengan menggunakan Jest, tapi ups.. sebelum itu kamu harus paham dulu nih tentang **jenis-jenis pengujian (testing)**.

Ada 3 jenis testing yang bisa kamu baca detailnya di slide berikutnya.



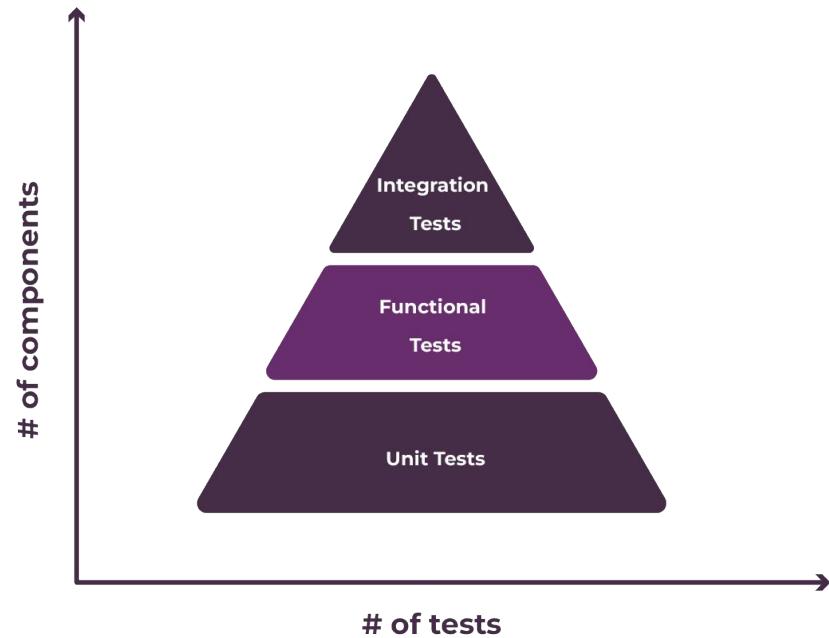
1. Unit Testing

Unit Testing mengacu pada test yang dilakukan pada potongan-potongan code atau unit, sehingga kamu dapat memastikan bahwa potongan code tersebut dapat berjalan dengan semestinya, sesuai dengan yang kamu harapkan.



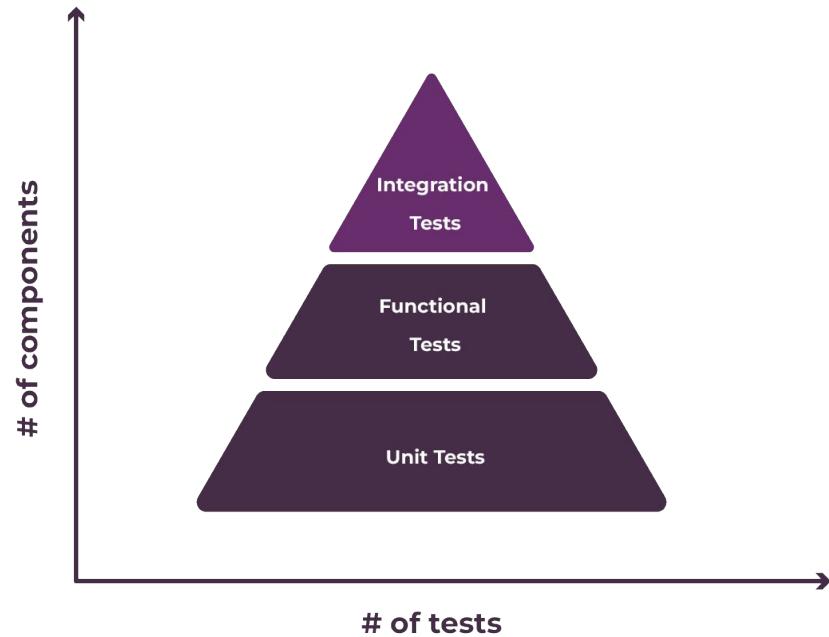
2. Functional Testing

Functional Testing mengacu pada test yang dilakukan untuk melihat behavior dari component.



3. Integration Testing

Integration Testing mengacu pada test yang dilakukan kepada keseluruhan sistem dari aplikasi, seperti layaknya end-user menggunakan aplikasi tersebut.





Contoh testing terhadap Code

Salah satu contoh pengujian yang paling mudah kita temui adalah pada website hackerrank.

Website tersebut menunjukkan suatu hasil expect yang harus dipenuhi berdasarkan input yang disediakan (Test Case).

Sample Input

```
a = 2  
b = 3
```

Sample Output

```
5
```

Explanation

```
2 + 3 = 5.
```

Test case 0

Test case 1

Compiler Message

Success

Input (stdin)

```
1 100  
2 1000
```

Download

Expected Output

```
1 1100
```

Download



Kalo udah paham soal konsep Jest dan jenis-jenis testing yang ada, habis ini langsung aja kita pelajari cara menggunakan Jest yuk~





1. Persiapkan project React Native

Kamu bisa pakai Project-mu yang sudah ada atau bikin project React Native baru khusus untuk belajar video.

Kita akan coba membuat project React Native baru.



```
npx react-native init rnmedia
```



```
admins-MacBook-Pro-2:documents 161552.mikhael$ npm install --save-dev jest
```

2. Install Jest

Umumnya kamu harus melakukan install pada project JavaScript yang ingin kamu unit test seperti pada gambar di samping, namun mulai dari versi React Native 0.38, Jest sudah tersedia secara default saat menjalankan. Cek pada file **package.json** kamu seperti pada gambar di samping.

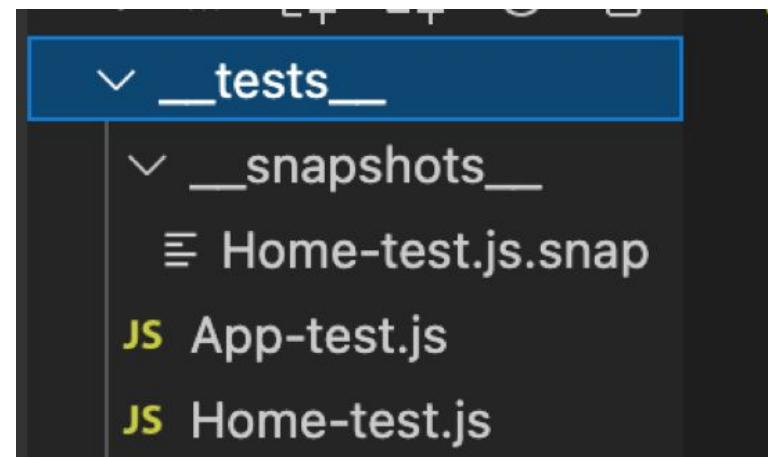


```
{
  "scripts": {
    "test": "jest"
  },
  "jest": {
    "preset": "react-native"
  }
}
```



3. Pahami kegunaan folder `_tests_` pada root project folder

Kalau kamu lihat di root folder React Native-mu, akan ada satu folder bernama `_tests_` . Folder itu adalah folder yang kamu pakai untuk menyimpan file yang akan kamu uji (test).



4. Pahami code testing

Pada dasarnya, function yang kamu gunakan untuk melakukan uji adalah test() dan expect (), seperti pada gambar di samping. Sederhananya kamu bisa memahami function tersebut yang bertujuan untuk menguji sesuatu dengan mengexpect hasil sesuatu.

contoh :

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

JS App-test.js ●

```
1 test('something', () => {
2   expect('something')
3 })
```

PROMPT bungferdy@Ferdlys-MacBook-Pro % yarn add @types/jest --dev
yarn add v1.22.4
[1/4] ⚡ Resolving packages...
[2/4] 🏛 Fetching packages...
[3/4] ⚡ Linking dependencies...
warning "@react-native-community/eslint-config > @typescript-eslint/eslint-plugin > tsutils@3.17.1" has unmet peer dependency "typescript@>2.8.0 || > 3.2.0-dev || > 3.3.0-dev || > 3.4.0-dev || > 3.5.0-dev || > 3.6.0-dev || > 3.6.0-beta || > 3.7.0-dev || > 3.7.0-beta".
[4/4] ⚡ Success: 1 package saved.
0/1863 [] 0/1863

5. Pahami Snapshot Testing

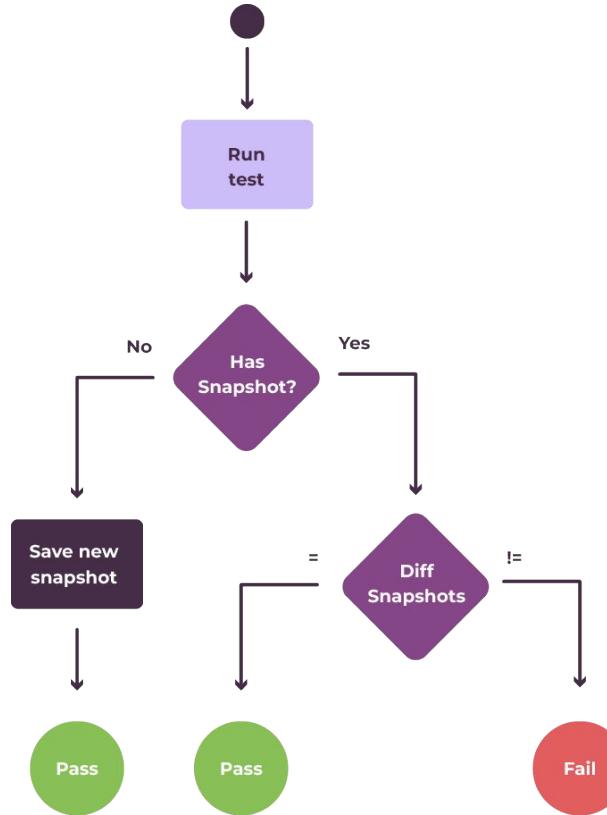
Snapshot testing adalah tools yang berguna banget kalau kamu mau memastikan UI-mu nggak berubah secara tiba-tiba.



Lanjutannya~

Kasus uji snapshot biasa merender komponen UI, mengambil snapshot, lalu membandingkannya dengan file snapshot referensi yang disimpan di samping pengujian.

Pengujian akan gagal jika kedua snapshot tidak cocok: perubahan tidak terduga, atau snapshot referensi perlu diperbarui ke versi baru komponen UI.





6. Buat 1 file komponen sederhana

Untuk melakukan snapshot testing, buatlah file komponen sederhana seperti contoh di samping bernama `Home.js`.

File tersebut bertujuan agar kamu bisa melakukan pengujian dan memastikan ketika UI di render file tersebut nggak akan mengalami perubahan yang tiba-tiba.



```
src > JS Home.js > Home
1 import React, { Component } from 'react';
2 import { Text, View } from 'react-native';
3
4 export default class Home extends React.Component {
5   render() {
6     return (
7       <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
8         <Text style={{fontSize : 10}}>Home Component</Text>
9       </View>
10    )
11  }
12 }
```



7. Buat File Home-test.js

Selanjutnya, buatlah satu file bernama `Home-test.js` di dalam folder `__tests__`.

Perhatikan code di samping, kamu akan memanggil component Home yang sudah kamu buat ke file ini dan fungsi renderer yang kamu gunakan untuk menyimpan hasil hasil renderer ke dalam variabel snap yang bertipe konstan.

```
__tests__ > js Home-test.js > ...
1 import 'react-native';
2 import React from 'react';
3 import Home from '../src/Home';
4 import renderer from 'react-test-renderer';
5
6 test('Home Snapshot', () => {
7   const snap = renderer.create(
8     <Home />
9   ).toJSON();
10  expect(snap).toMatchSnapshot()
11 })
```



Lanjutannya~

File tersebut akan di-convert menjadi json, sehingga kamu bisa mencocokkan hasil file home-mu dengan function `toMatchSnapshot()`.

```
__tests__ > JS Home-test.js > ...
1  import 'react-native';
2  import React from 'react';
3  import Home from '../src/Home';
4  import renderer from 'react-test-renderer';
5
6  test('Home Snapshot', () => {
7    const snap = renderer.create(
8      <Home />
9    ).toJSON();
10   expect(snap).toMatchSnapshot()
11 })
```



8. Pahami Command npm test

Setelah memiliki setidaknya 1 file yang kamu tes di folder `__tests__`, langkah selanjutnya adalah dengan mengeksekusi file tersebut untuk menguji hasilnya.

Berikut langkah yang harus kamu lakukan:

```
admins-MacBook-Pro-2:UnitTest 161552.mikhael$ npm test Home-test.js
> UnitTest@0.0.1 test /Users/161552.mikhael/Documents/AlatTempur/UnitTest
> jest "Home-test.js"

Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#b
rowers-data-updating
PASS  __tests__/views/Home-test.js
  ✓ Home Snapshot (53 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        0.536 s, estimated 4 s
Ran all test suites matching /Home-test.js/i.
admins-MacBook-Pro-2:UnitTest 161552.mikhael$
```



- Jalankan **npm test**, dengan command ini kamu dapat melakukan testing untuk seluruh file yang terdapat di folder `_tests__`

```
admins-MacBook-Pro-2:UnitTest 161552.mikhael$ npm test Home-test.js
> UnitTest@0.0.1 test /Users/161552.mikhael/Documents/AlatTempur/UnitTest
> jest "Home-test.js"

Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#b
rowers-data-updating
PASS  __tests__/views/Home-test.js
  ✓ Home Snapshot (53 ms)

  Test Suites: 1 passed, 1 total
  Tests:       1 passed, 1 total
  Snapshots:  1 passed, 1 total
  Time:        0.536 s, estimated 4 s
  Ran all test suites matching /Home-test.js/i.
admins-MacBook-Pro-2:UnitTest 161552.mikhael$
```



- Jalankan **npm test <fileName>** untuk melakukan test terhadap spesific file yang akan di test.

Jika file yang diuji melewati hasil pengujian, maka akan menampilkan informasi passed. Jika tidak, maka akan menampilkan informasi failed, sehingga kamu perlu memperbaiki file tsb

- Jalankan **npm test -- -u**, untuk mengupdate file yang diubah.

```
admins-MacBook-Pro-2:UnitTest 161552.mikhael$ npm test Home-test.js
> UnitTest@0.0.1 test /Users/161552.mikhael/Documents/AlatTempur/UnitTest
> jest "Home-test.js"

Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#b
rowers-data-updating
PASS  __tests__/_views/Home-test.js
  ✓ Home Snapshot (53 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        0.536 s, estimated 4 s
Ran all test suites matching /Home-test.js/i.
admins-MacBook-Pro-2:UnitTest 161552.mikhael$
```



9. Function and State

Kalau sebelumnya kamu melakukan tes pada code UI yang kamu buat, bagaimana jika kamu melakukan pengujian terhadap unit terkecil yang terdapat di file, seperti contohnya function dan state.

Perhatikan code di samping, terdapat sebuah data yang diinisialisasikan di state dan memiliki fungsi change.

```
import React, { Component } from 'react';
import { Text, View, StyleSheet } from 'react-native';
import Profile from './Profile'

export default class Home extends React.Component {
  constructor() {
    super();
    this.state = {
      data: ''
    }
  }

  change(x) {
    this.setState({ data: x * 10 })
  }

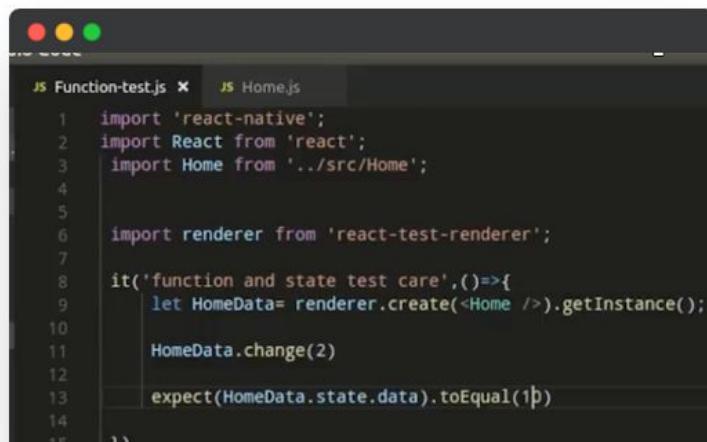
  render() {
    return (
      <View style={styles.text}>
        <Profile/>
        <Text style={{ fontSize: 25 }}>Home Component</Text>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  text: {
    flex: 1, alignItems: 'center', justifyContent: 'center'
  }
})
```

10. Function and State

Selanjutnya, untuk melakukan testing terhadap nilai state dan fungsi tersebut, buatlah 1 file baru, misal bernama **home-function.js**.

Pada file ini kita menggunakan function `getInstance()` bertujuan untuk kita dapat menggunakan function `change` yang terdapat pada file `Home`.

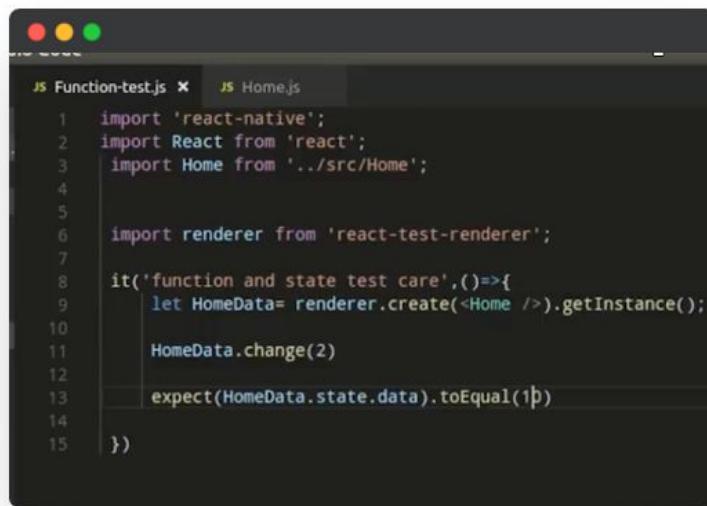


```
JS Function-test.js x JS Home.js
1 import 'react-native';
2 import React from 'react';
3 import Home from '../src/Home';
4
5
6 import renderer from 'react-test-renderer';
7
8 it('function and state test care', ()=>{
9   let HomeData= renderer.create(<Home />).getInstance();
10
11   HomeData.change(2)
12
13   expect(HomeData.state.data).toEqual(1)
14
15 })
```



Tujuan testing adalah agar kamu dapat melakukan pengecekan seandainya ada function yang, tanpa kamu sadari, nilai variabelnya nggak sengaja terpengaruh oleh beberapa function lain.

Kesalahan itu akan menyebabkan bug pada production, sehingga perlu kamu minimalisir. Psst.. jangan lupa jalankan npm test setiap kamu akan menguji file yaa.

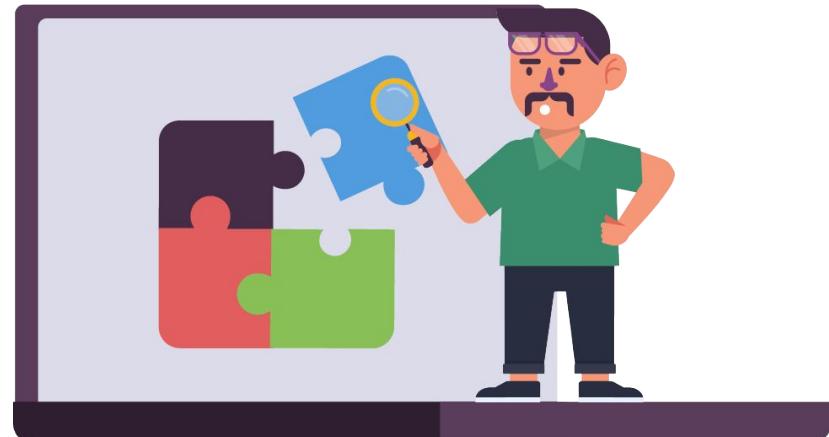


```
JS Function-test.js x JS Home.js
1 import 'react-native';
2 import React from 'react';
3 import Home from '../src/Home';
4
5
6 import renderer from 'react-test-renderer';
7
8 it('function and state test care', ()=>{
9   let HomeData= renderer.create(<Home />).getInstance();
10
11   HomeData.change(2)
12
13   expect(HomeData.state.data).toEqual(1)
14
15 })
```

11. Integration Test

Saat mengkoding suatu software systems yang lebih besar, yang setiap bagian dari sistemnya perlu berinteraksi satu sama lainnya.

Dalam unit testing, jika component bergantung pada API, terkadang kamu perlu melakukan pengujian di bagian integrasi untuk memastikan nggak ada perubahan tiba-tiba terjadi pada API saat di-production.





12. Buat suatu file dummy response API

Contoh:

Buatlah 1 file di `_tests_` yang bernama `homeDummyData.js`, yang berisi replika dari hasil respon suatu API dengan method Get seperti di samping.

```
export const data = {
  "count": 1118,
  "next": "https://pokeapi.co/api/v2/pokemon?offset=110&limit=10",
  "previous": "https://pokeapi.co/api/v2/pokemon?offset=90&limit=10",
  "results": [
    {
      "name": "electrode",
      "url": "https://pokeapi.co/api/v2/pokemon/101/"
    },
    {
      "name": "exeggute",
      "url": "https://pokeapi.co/api/v2/pokemon/102/"
    },
    {
      "name": "exeggutor",
      "url": "https://pokeapi.co/api/v2/pokemon/103/"
    },
    {
      "name": "cubone",
      "url": "https://pokeapi.co/api/v2/pokemon/104/"
    },
  ]
}
```



13. Buat file testing integration

Pastikan kamu sudah menginstall axios-mock-adapter dan Axios seperti pada chapter mengenai API, kemudian perhatikan pada file di samping.

```
import MockAdapter from 'axios-mock-adapter';
import axios from 'axios';
import { ApiTest, postLoginApi } from '../../../../../src/Api';
import { data, dataLogin } from '../dummy/data-pokemon';

describe('Api', () => {
  it('Api Pokemon test', async () => {
    let mock = new MockAdapter(axios);
    mock.onGet('https://pokeapi.co/api/v2/pokemon?offset=100&limit=10').reply(200, data);

    //act
    let res = await ApiTest();
    //assert
    expect(res.status).toEqual(200)
    expect(res.data).toEqual(data)
  });
  //post
  it('api login', async () => {
    let mock = new MockAdapter(axios);
    const loginBody = {
      "email": "ianvaldear2001@gmail.com",
      "password": "makandurianBR012"
    };
    mock.onPost('http://code.aldipee.com/api/v1/auth/login').reply(200, dataLogin);

    //act
    let res = await postLoginApi(loginBody);
    //assert
    expect(res.data).toEqual(dataLogin)
    expect(res.status).toEqual(200)
  });
})
```



Dengan menggunakan mock kamu seakan-akan melakukan fetching terhadap API tersebut dengan function OnGet.

Selanjutnya bandingkan data replika yang kamu buat dengan hasil response dari API:

- | 1. | Status |
|------------------|--------|
| 2. Data Response | |

Response

```
import MockAdapter from 'axios-mock-adapter';
import axios from 'axios';
import { ApiTest, postLoginApi } from '../../../../../src/Api';
import { data, dataLogin } from '../dummy/data-pokemon';

describe('Api', () => {
  it('Api Pokemon test', async () => {
    let mock = new MockAdapter(axios);
    mock.onGet('https://pokeapi.co/api/v2/pokemon?offset=100&limit=10').reply(200, data);

    //act
    let res = await ApiTest();
    //assert
    expect(res.status).toEqual(200)
    expect(res.data).toEqual(data)
  });
  //post
  it('api login', async () => {
    let mock = new MockAdapter(axios);
    const loginBody = {
      "email": "ianvaldear2001@gmail.com",
      "password": "makandurianBR012"
    };
    mock.onPost('http://code.aldipee.com/api/v1/auth/login').reply(200, dataLogin);

    //act
    let res = await postLoginApi(loginBody);
    //assert
    expect(res.data).toEqual(dataLogin)
    expect(res.status).toEqual(200)
  });
})
```



14. Buat file dummy dari response post

Selanjutnya, dengan hal yang sama kamu akan melakukan pengujian terhadap API post.

Kamu bisa pakai API apapun yang gratisan, contohnya pada code di samping, kita akan menguji integrasi API login kita.

Buatlah replica login data seperti contoh di samping atau sesuaikan dengan response API post yang kamu pakai.

```
export const dataLogin = {
  "user": {
    "role": "user",
    "isEmailVerified": true,
    "email": "lanvaldear2001@gmail.com",
    "name": "Aldi Daniela Pranata",
    "id": "60f96e94920dcbeec63c9172"
  },
  "tokens": {
    "access": {
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9",
      "expires": "2021-08-12T05:38:40.528Z"
    },
    "refresh": {
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9",
      "expires": "2021-09-11T05:08:40.529Z"
    }
  }
}
```



15. Buat file testing integration(2)

Seperti pada contoh sebelumnya, kamu membuat file mocking integration untuk melakukan fetching API. Tujuannya agar kamu bisa memastikan nggak ada perubahan response API yang kamu terima.

Jangan lupa, jalankan npm test untuk menguji hasil dari file yang telah kamu buat.

```
import MockAdapter from 'axios-mock-adapter';
import axios from 'axios';
import { ApiTest } from '../src/Api';
import { data } from '../dummy/data-pokemon';

describe('Api', () => {
  it('Api Pokemon test', async () => {
    let mock = new MockAdapter(axios);
    mock.onGet(`https://pokeapi.co/api/v2/pokemon?offset=100&limit=10`).reply(200, data);

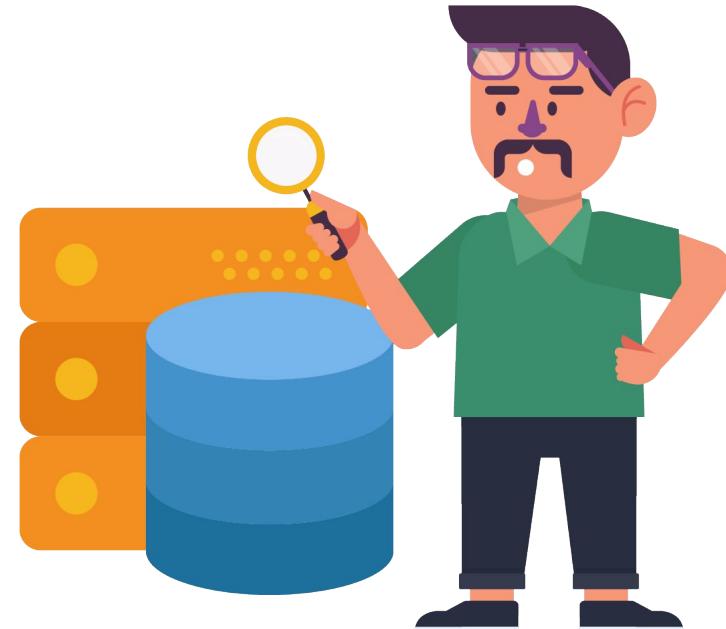
    //act
    let res = await ApiTest();
    //assert
    expect(res.status).toEqual(200)
    expect(res.data).toEqual(data)
  });
})
```

16. Review Integration Test

Integration test ini digunakan untuk memastikan nggak ada perubahan data yang tiba-tiba dari response API.

Contoh:

Kamu menerima suatu data bertipe Array. [...], kemudian array tersebut kamu eksekusi dengan map() function. Jika ada suatu kondisi yang mengubah nilai variabel array [...] dari response API, mungkin dikarenakan data di database kosong menjadi `undefined` atau `null`.





Hal tersebut akan mengakibatkan aplikasi bug/crash saat di production karen null atau undefined tidak dapat menerima function map().

Nah, itulah salah salah satu alasan kamu perlu menerapkan integration test, guys.





Selain Jest, ada juga utilitas pengujian lain, yaitu **Enzyme**. Jika digabungkan, penggunaannya kedua unit test ini akan sangat membantu proses testing.

Kayak apa cara kerjanya? Ayo cari tahu di slides berikutnya.





Enzyme? Apa lagi ini ya?

Enzyme adalah paket utilitas pengujian yang awalnya dikembangkan oleh Airbnb, sebelum akhirnya ditransfer ke pengembang lain.

Enzyme dianggap dapat membuat pengujian komponen React lebih mudah karena Enzyme memungkinkan kita untuk mengeksekusi tes, **bahkan untuk elemen yang lebih kecil sekalipun.**

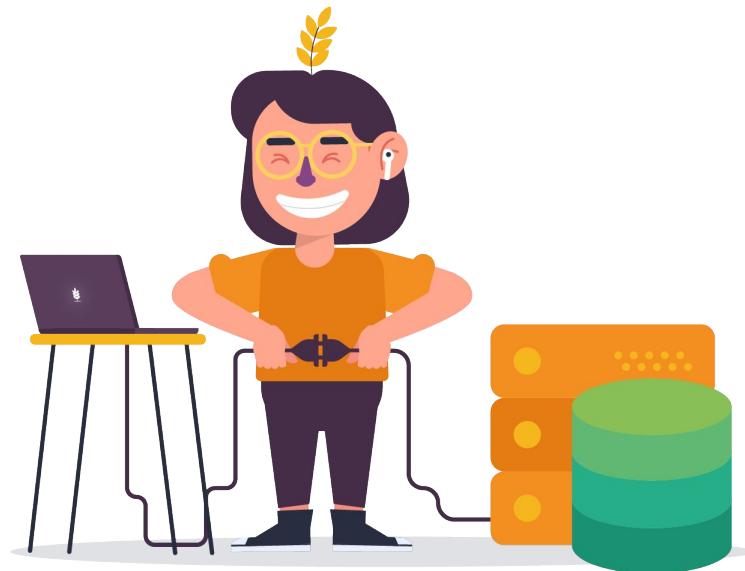




Lalu bagaimana ya cara Setup Enzyme?

Melakukan testing dengan Jest tanpa dibantu oleh Enzyme adalah hal yang sudah cukup. Tapi dengan menggunakan Jest dan Enzyme kamu **bisa melakukan testing sampai ke bagian yang lebih kecil**, yaitu sampai ke elemen yang ada di source code kamu.

Makanya nih kalau kamu tahu cara setup Enzyme. Cekidot!





**Setup Enzyme, mulai dari mana?
Ikuti langkahnya:**

1.

Instalasi

Jalankan command `npm i jest-environment-enzyme`
`jest-enzyme`
`--save-dev enzyme enzyme-adapter-react-16`

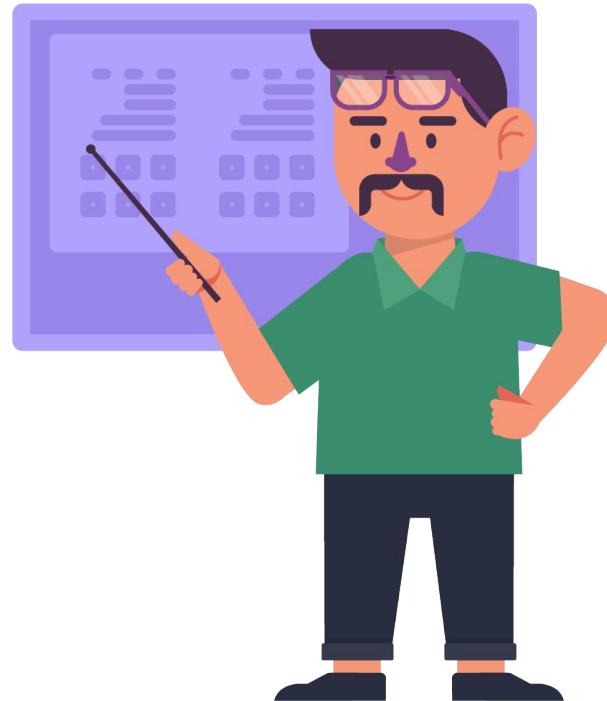




Opsional: Jika error saat pertama kali run, tambahkan npm install --save-dev react-dom enzyme

Sumber: :

<https://enzymejs.github.io/enzyme/docs/guides/react-native.html>





2. Membuat File

Buatlah file sederhana di mana ada proses untuk input dan button yang menampilkan info menggunakan alert dari informasi yang kamu input.

```
import React, {useState} from 'react'
import { View, Text, TextInput, TouchableOpacity, StyleSheet, Alert } from 'react-native'

export default function App() {
  const [name, setname] = useState('')
  return (
    <View style={styles.container}>
      <Text>Absen Nama :</Text>
      <TextInput
        testID='input-name'
        onChangeText={(name => setname(name))}
        style={styles.input}
        placeholder='Name'
        value={name}
      />
      <TouchableOpacity
        testID='button-absen'
        onPress={() => Alert.alert('nama', name)}
        <View style={styles.button}>
          <Text style={styles.textButton}>Absen</Text>
        </View>
      </TouchableOpacity>
    </View>
  )
}
```



3. Membuat file Test

Buatlah file test sederhana seperti gambar di samping, dengan perbedaan satu file komponen yang akan kamu tes nantinya kamu tambahkan ke function shallow.

Di samping adalah contoh sederhana untuk melakukan snapshot testing.

```
import 'react-native'
import React from 'react'
import Adapter from 'enzyme-adapter-react-16'
import {shallow, configure} from 'enzyme'
import renderer from 'react-test-renderer'
import App from '../src/App'
configure({adapter: new Adapter(), disableLifecycleMethods: true})

const profileWrapper = shallow(<App />

jest.mock('@react-navigation/native', () => ({
  useNavigation: component => component,
}))

jest.mock('react-native/Libraries/Animated/src/NativeAnimatedHelper')
jest.mock('native-base')

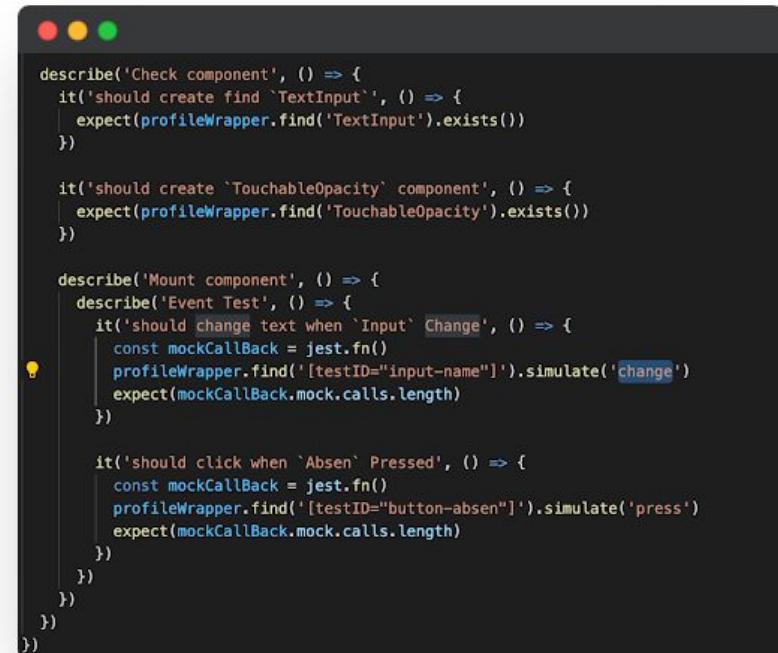
describe('App Screen', () => {
  it('should renders correctly', () => {
    renderer.create(<App />
  })

  it('should renders `App Screen` module correctly', () => {
    expect(profileWrapper).toMatchSnapshot()
  })
})
```



Dengan enzyme, kamu bisa mengakses level yang lebih kecil untuk elemen pada komponen, seperti di samping.

Kamu juga dapat melakukan pengecekan apakah ada component textinput dan touchableopacity, hingga melakukan pengujian terhadap function text input dan touchableopacity.



```
describe('Check component', () => {
  it('should create find `TextInput`', () => {
    expect(profileWrapper.find('TextInput').exists())
  })

  it('should create `TouchableOpacity` component', () => {
    expect(profileWrapper.find('TouchableOpacity').exists())
  })
}

describe('Mount component', () => {
  describe('Event Test', () => {
    it('should change text when `Input` Change', () => {
      const mockCallBack = jest.fn()
      profileWrapper.find('[testID="input-name"]').simulate('change')
      expect(mockCallBack.mock.calls.length)
    })

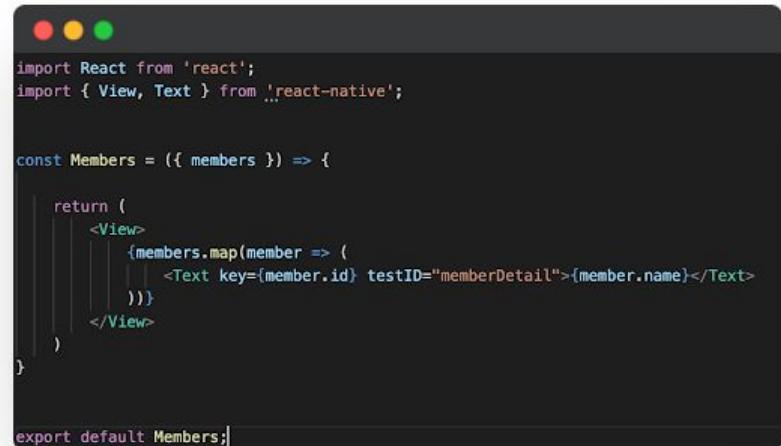
    it('should click when `Absen` Pressed', () => {
      const mockCallBack = jest.fn()
      profileWrapper.find('[testID="button-absen"]').simulate('press')
      expect(mockCallBack.mock.calls.length)
    })
  })
})
```



Kita coba dengan contoh lain yuk

Buatlah suatu file component yang memiliki beberapa component. Pada contoh di samping component text akan dipanggil sesuai jumlah members.length.

Setiap members memiliki id key dan name key, yang dirender menggunakan testID='memberDetail'



```
import React from 'react';
import { View, Text } from 'react-native';

const Members = ({ members }) => {

  return (
    <View>
      {members.map(member => (
        <Text key={member.id} testID="memberDetail">{member.name}</Text>
      ))}
    </View>
  )
}

export default Members;
```



Implementasi Jest + Enzyme

Berdasarkan gambar di samping, kamu bisa menguji, apakah setiap component kecil tersebut akan menampilkan informasi sesuai dari data apa saja yang tersedia.

Sehingga jika ada perubahan yang mengakibatkan salah satu komponen gagal menampilkan informasi yang sesuai, maka kamu dapat mengetahuinya lebih dulu.

```
import React from 'react';
import { shallow } from 'enzyme';
import toJson from 'enzyme-to-json';

import Members from './Members';

const members = [{ id: 1, name: 'Daphne' }, { id: 2, name: 'Margret' }, ];

describe('Members Component', () => {
  it('should render without issues', () => {
    const component = shallow(
      <Members members={members} />
    );
    expect(component.length).toBe(1);
    expect(toJson(component)).toMatchSnapshot();
  });

  it('should render all item in members', () => {
    const wrapper = shallow(
      <Members members={members} />
    );
    expect(wrapper.find({ testID: 'memberDetail' }).length).toBe(2);
  });

  it('should render correct names', () => {
    const wrapper = shallow(
      <Members members={members} />
    );
    wrapper.find({ testID: 'memberDetail' }).forEach((node, index) => {
      expect(node.props().children).toBe(members[index].name);
    });
  });
});
```



Manfaat pengujian terhadap code

Untuk lebih mengetahui function bawaan atau apa saja yang bisa kamu lakukan dengan Jest+Enzyme, dibutuhkan waktu ekstra untuk mencoba dan memahaminya dari dokumentasi resmi.

Kamu bisa eksplor Jest+Enzyme di [link berikut ini](#)





Berikut beberapa manfaat yang kamu bisa dapatkan dengan melakukan pengujian terhadap code :

1. Kamu dapat menjamin atau meningkatkan code quality, karena code yang kamu tulis sudah melewati tahap pengujian.
2. Membuat kamu lebih yakin terhadap code yang akan kamu deploy di production.
3. Meminimalisir resiko terjadinya bug. Risiko nggak bisa kamu hindari, namun kamu bisa meminimalisir risiko terjadi bug ataupun crash.



Saatnya kita Quiz!





1. Apa Keuntungan kita melakukan pengujian code?

- A. Akan mempercepat proses pembuatan aplikasi.
- B. Akan rentan terhadap keamanan aplikasi.
- C. Akan meminimalisir bug yang terjadi di production.



2. Pernyataan dibawah ini yang benar adalah, kecuali..

- A. Jest sudah tersedia secara default di project react native version 0.38 ++
- B. unit testing merupakan bagian dari TDD (test driver development)
- C. menerapkan pengujian akan mempercepat proses development



3. Apa dasar function dalam melakukan pengujian dengan jest?

- A. test('something') .. to expect('something')
- B. axios to mock('api')
- C. renderer().tojson equals.(json)



4. Sejauh ini apa yang kita pahami dari Unit Test?

- A. Melakukan pengujian terhadap unit terkecil seperti component, function atau pun integrasi terhadap API
- B. Melakukan pengujian dengan memposisikan diri sebagai user dari aplikasi tersebut
- C. Melakukan perencanaan pengujian dengan membuat test case dari aplikasi

Terima Kasih!



Next Topic

loading...