React Native

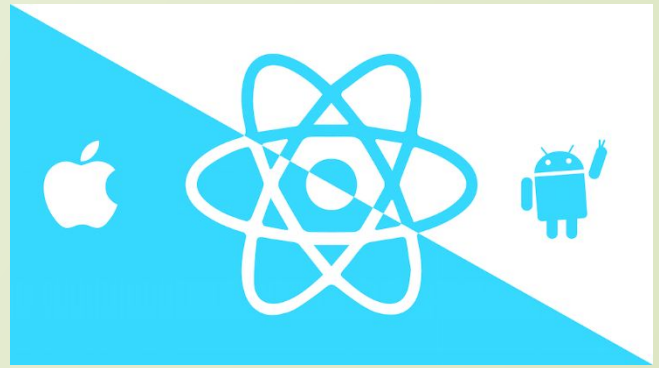
UNIT test, TDD, JEST and DETOX

- DEV {Education}
- Преподаватель –Эльмар Гусейнов

React Native

- **UNIT TEST is :**

software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.



React Native

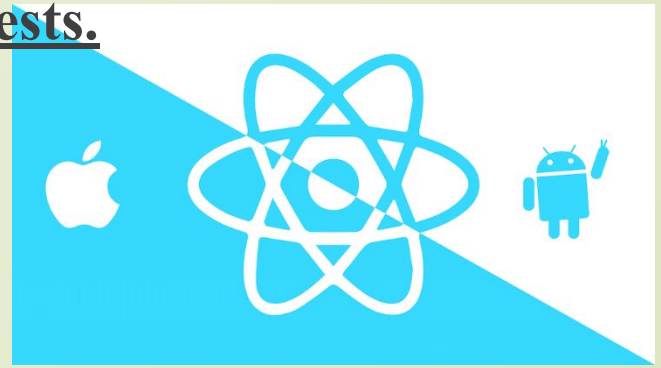
- When possible without UNIT TEST

- the project is not complicated. The application is placed on several screens (1-5-10), there is no complex logic, just text and media files

- the project is not long-playing. The project is made to order, the lead time is short, you will not need support or adding new functionality

- you have developers who never make mistakes :)

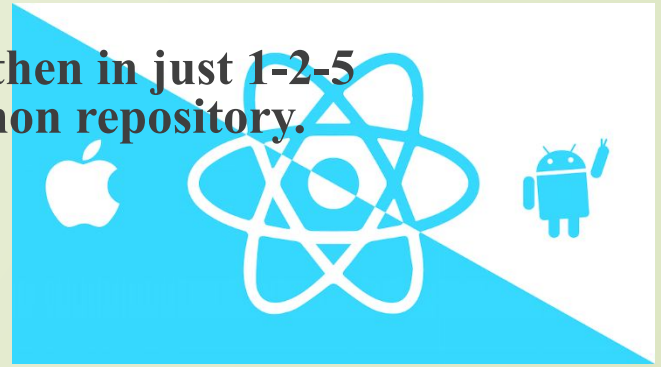
In other cases, it is desirable to cover the code with tests.



React Native

- Advantages of UNIT TESTs

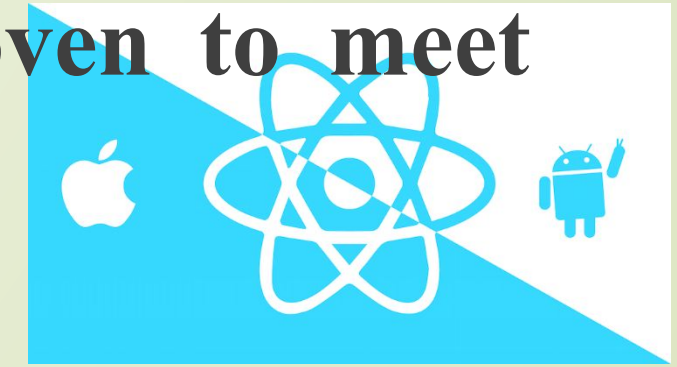
- you can, without fear, do code refactoring
- Code becomes more transparent.
- there is no need for some manual tests
- if you configure the build process correctly, the “bad” code will not get into the general repository
- Unit tests are the fastest. A few minutes are enough to test a large enough application.
- If you break up unit tests into different sets and have a set with smoke tests, then in just 1-2-5 minutes you can decide whether or not to commit the written code to a common repository.



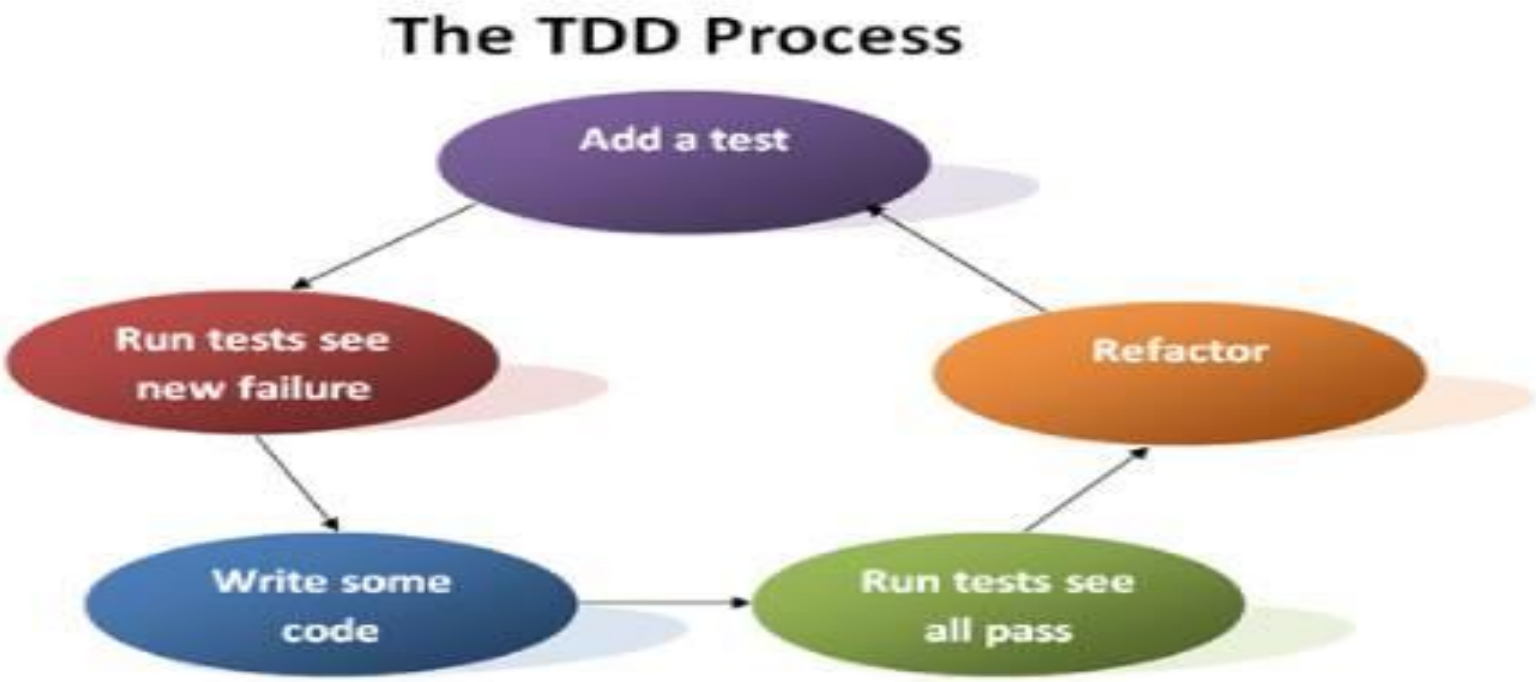
React Native

- TDD methodology

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved so that the tests pass. This is opposed to software development that allows software to be added that is not proven to meet requirements.

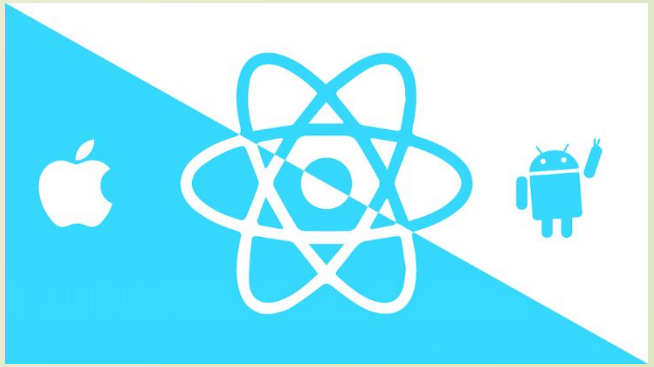


React Native



React Native

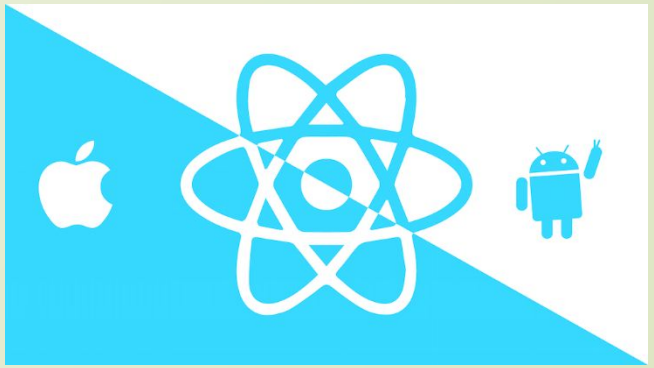
- TDD methodology algorithm:
- write a test and code for it for non-existent functionality (our future function) [“add test”+ “run test new see new failure”]
- write the functional itself (our function) [“Write some code”]
- testing functionality using our test [“Run test see all pass”]
- refactor code and again test it [“Refactor”]



React Native

- **Code coverage testing methods (main commonly used)**

- **Statement testing**
- **Decision testing**
- **Condition testing**
- **Multiple condition testing**



React Native

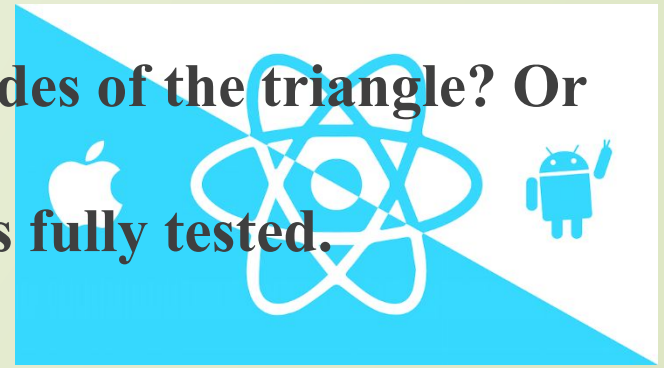
Statement testing (testing of operators)

- Statement testing assumes that for 100% coverage of the code it is necessary that each statement of the program be executed at least once
- For example, for 100% coverage of this code, one test is enough, where side1Length = 1, side2Length = 1, side3Length = 1;

```
function Myfunc(variables){
let side1Length = side1.Text;
let side2Length = side2.Text;
let side3Length = side3.Text;
if ((side1Length == side2Length) &&
(side2Length == side3Length))
 {
lblResult.Text = "triangle- equilateral !";
 } }
```

But what if the user does not enter anything in the fields for the sides of the triangle? Or enter different values? Or enter letters?

100% coverage of the code does not guarantee that the program is fully tested.



React Native

Decision testing

- **During decision testing (decision testing), it is necessary to draw up such a number of tests in which each condition in the program will accept both a true value and a false one.**

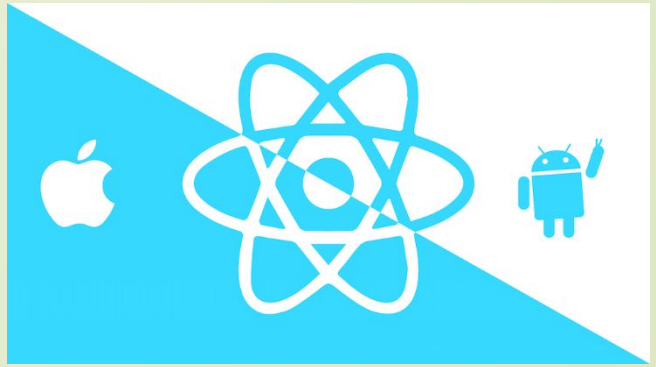
In the following example, 2 tests are enough for 100% coverage:

1 -> a = 3, b = 0, x = 4

2 -> a = 3, b = 1, x = 0

```
function myFunc(a, b,  x)
{
if ((a > 1) && (b == 0))
{x = x / a;}
if ((a == 2) || (x > 1))
{x++;}
}
```

But what if the developer made a mistake in the condition a == 2 (let's say you had to write a == 5)?



React Native

Condition testing

- During condition testing for 100% coverage of conditions, it is necessary that all conditions accept both false and true values.

In the following example, such a number of tests is necessary that conditions a> 1, b == 0, a == 2, x> 1 take both true and false values

```
function myFunc(a, b,  x)
{
if ((a > 1) && (b == 0))
{x = x / a;}
if ((a == 2)  ||  (x > 1))
{x++;}
}
```

That is, two tests are enough:

1 -> a = 2, b = 1, x = 2

2 -> a = 0, b = 0, x = 0

But at the same time, the line of code “x = x / a;” will not be executed even once, although the coverage will be 100%.

React Native

Multiple Condition testing

- **When testing multiple condition testing for 100%, full coverage of all conditions and all operators is required.**

That is, in the previous example, add another test: a = 3, b = 0, x = -5.

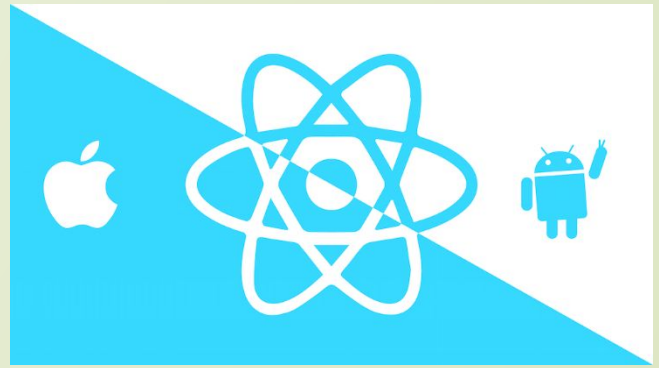
```
function myFunc(a, b,  x)
{
if ((a > 1) && (b == 0))
{x = x / a;}
if ((a == 2)  ||  (x > 1))
{x++;}
}
```

As a result, we get 3 tests:

1 -> a = 2, b = 1, x = 2

2 -> a = 0, b = 0, x = 0

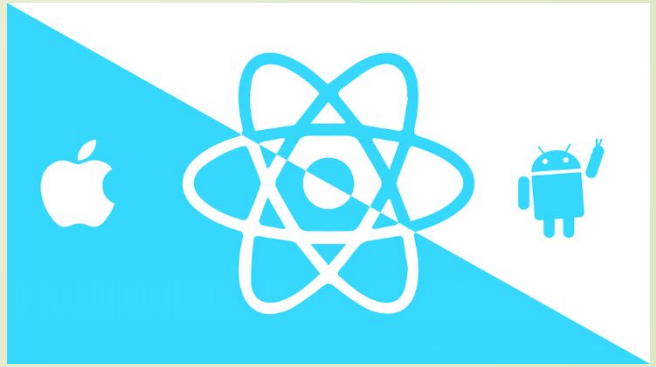
3 -> a = 3, b = 0, x = -5



React Native

- **JEST framework for testing**

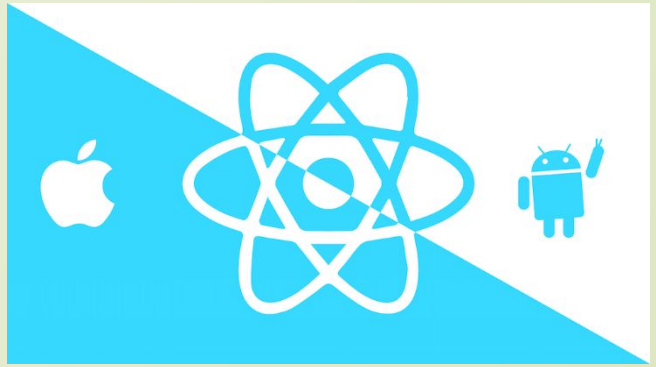
- **Official docs:** https://jestjs.io/docs/en/getting-started



React Native

JEST

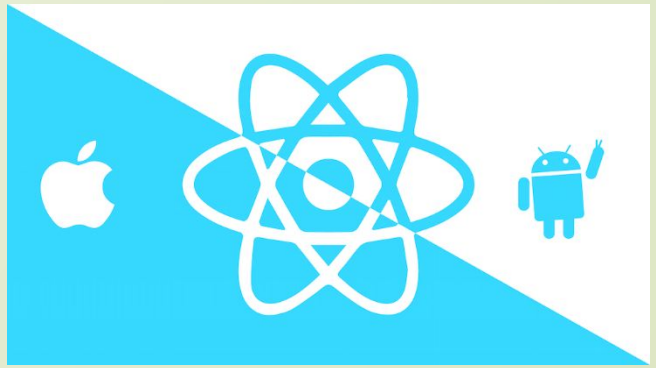
- **Official docs:** https://jestjs.io/docs/en/getting-started



React Native

JEST installation

- In your project : *npm install --save-dev jest*
- *In last versions of react-native is jest installed by default*



React Native
JEST methods using matchers

Common matchers

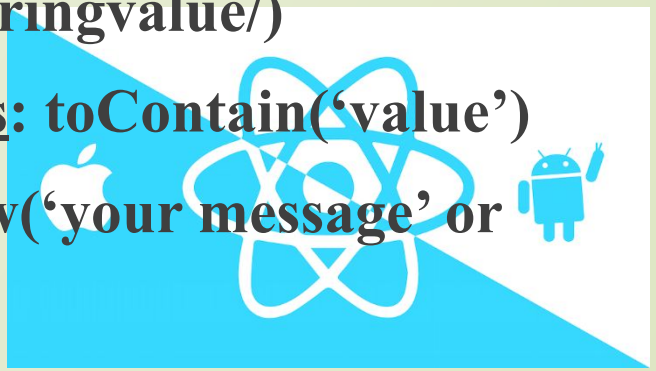
- toBe()
- toEqual()

Truthiness

- toBeNull() -matches only null
- toBeUndefined() -matches only undefined
- toBeDefined() -is the opposite of toBeUndefined
- toBeTruthy() -matches anything that an if statement treats as true
- toBeFalsy() - matches anything that an if statement treats as false
- **Full methods for "expected"** https://jestjs.io/docs/en/expect

Numbers

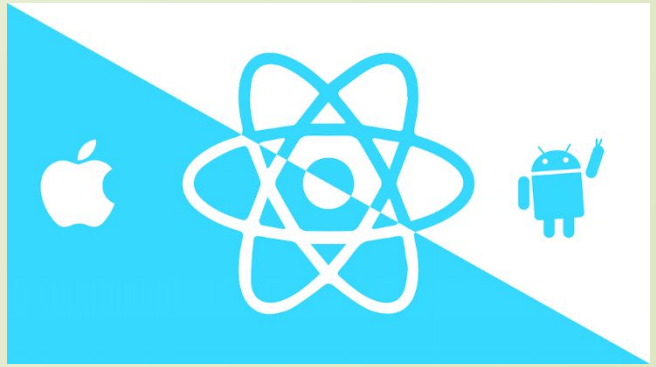
- toBeGreaterThan()
- toBeGreaterThanOrEqual()
- toBeLessThan()
- toBeLessThanOrEqual()
- toBe()
- toEqual()
- **Strings**: toMatch(/stringvalue/)
- **Arrays and iterables**: toContain('value')
- Exceptions: toThrow('your message' or /JDK/ -for example)



React Native

Structure of test file

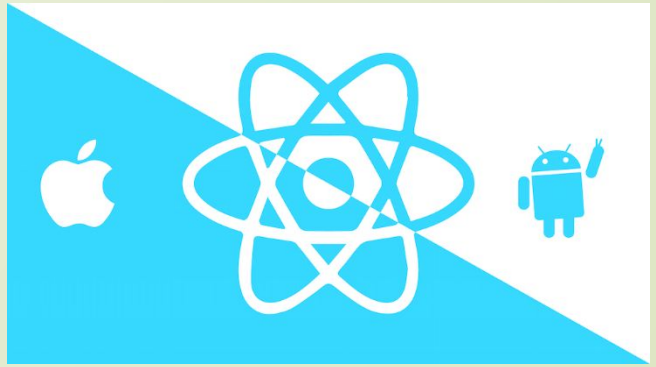
```
import statement……//(see on next page)
describe('Explanation of my tests', () => {

    it('explanation of my exactly unit test, for example, "snapshot test"', () => {
        // my test

        // expect(variable).toMatchSnapshot();
    })
It('my next test', ()=>{…… expect }
                                        }
```



React Native

Import statement of test file

```
import ‘react-native’
import React from 'react';
import Componentname from ‘../Componentfilename';
// Note: test renderer must be required after react-native.
import renderer from 'react-test-renderer';
```



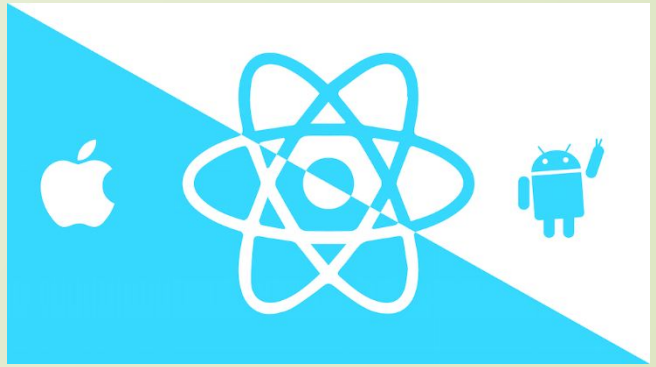
React Native

Snapshot testing

- Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly. Example take snapshot for my component Home.js:

```
describe('Test my component', () => {
    it('snapshot testing', () => {
        const mysnapshot1=renderer.create( <Home/>).toJSON()
        expect(mysnapshot1).toMatchSnapshot();
    })                                          }
```

Try command: npm run test



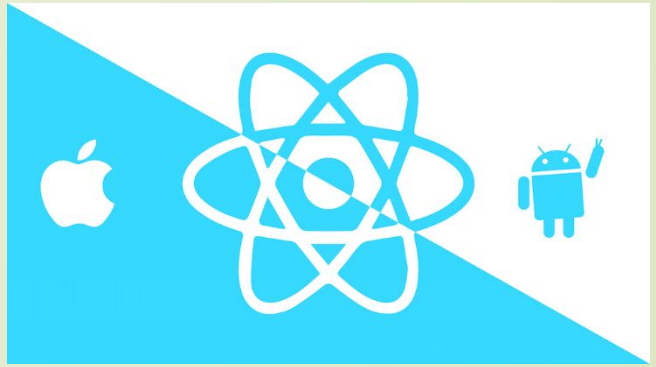
React Native

Function testing

- **Example** to test function myFunc from my component Home.js:

```
describe('Test my component', () => {
    it('function testing', () => {
        const myfunction=renderer.create( <Home/>).getInstance()
let variable1=myfunction.myFunc(myvalue)   // call myFunc from Home.js with value myvalue and store to variable1
        expect(variable1).toEqual(somevalue);
    })                                                    }
```

Try command: npm run test



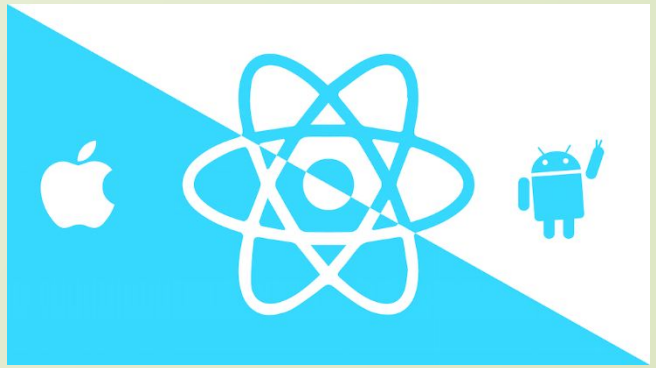
React Native

Find element testing

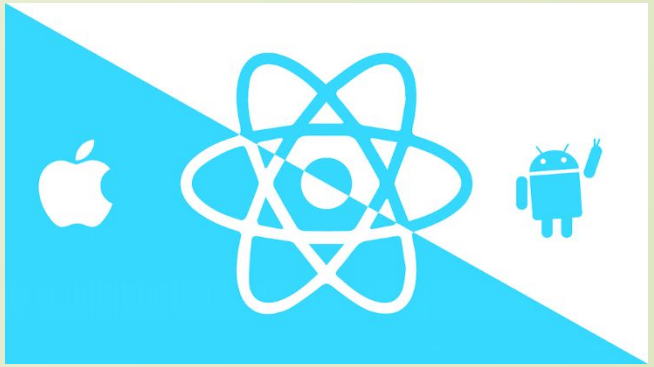
Example to find element in your component Home.js. Firstly, we need to add in your element testId={'usernamelabel'}, for example in your text element:

```
describe('Test my component', () => {
let findElement=function (tree, element){
console.log(tree)
let result=undefined
for (node in tree.children)
        { if (tree.children[node].props.testId==element)
                {result=true}
        }
                                        }
return result
it('finding element testing', () => {
        let tree=renderer.create( <Home/>).toJSON()
        expect(findElement(tree, 'usernamelabel')).toBeDefined();
                                }) }
```

Try command: npm run test



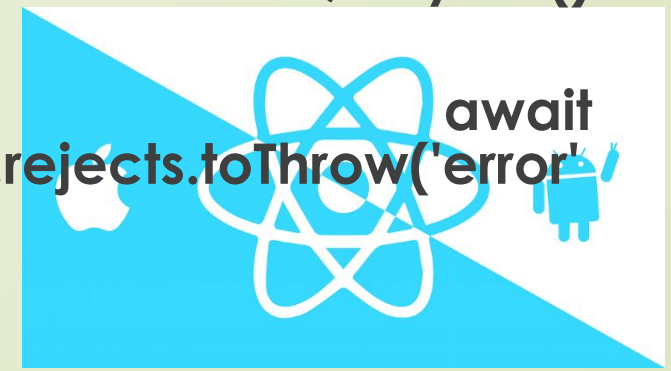
React Native Testing Part2



React Native
Testing a Async code
Async/await and resolves/rejects(combine)

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

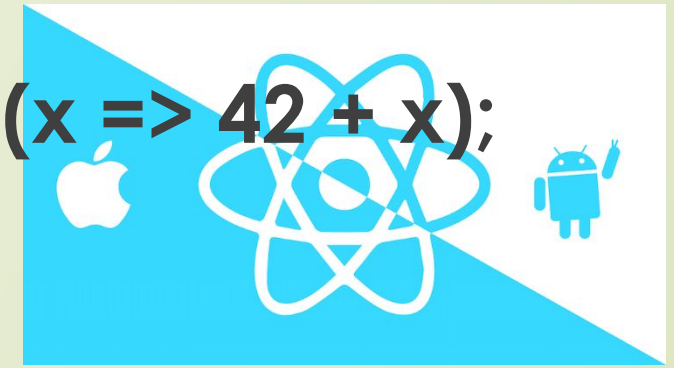
```
test('the data is peanut butter', async () =>
{
                                        await
expect(fetchData()).resolves.toBe('peanut
butter');
});

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



React Native
Mock function

- Mock functions are also known as "spies", because they let you spy on the behavior of a function that is called indirectly by some other code, rather than only testing the output.
- Create a mock function with **jest.fn()**

- **Example: const mockCallback = jest.fn(x => 42 + x);**
  or



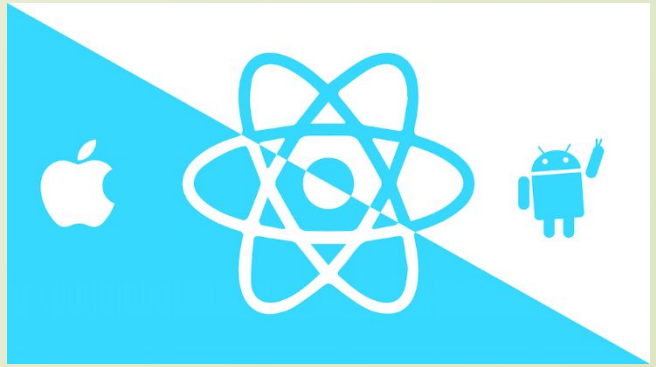
React Native Methods

- mockFn.getMockName()
- mockFn.mock.calls
- mockFn.mock.results
- mockFn.mock.instances
- mockFn.mockClear()
- mockFn.mockReset()
- mockFn.mockRestore()
- mockFn.mockImplementation(fn)
- mockFn.mockImplementationOnce(fn)
- mockFn.mockName(value)
- mockFn.mockReturnThis()
- mockFn.mockReturnValue(value)
- mockFn.mockReturnValueOnce(value)

Where mokeFN – your mock function name

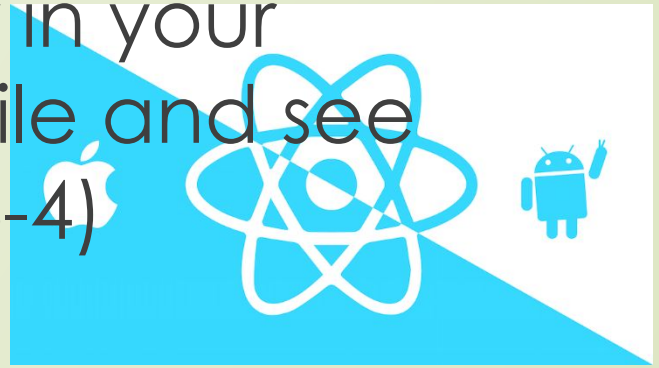
Examples and docs here:

https://jestjs.io/docs/en/mock-function-api



React Native
Jest: check test coverage

- 1. open package.json file
- 2. find row "test": "jest" and change to "test": "jest --coverage" (see screenshot1)
- 3. run your all tests and you can see in terminal coverage table and other info (see screenshot2)
  - Also all files saved to "coverage" folder in your project tree, you can open index.html file and see all info in your browser (see screenshot3-4)



React Native
Jest: check test coverage

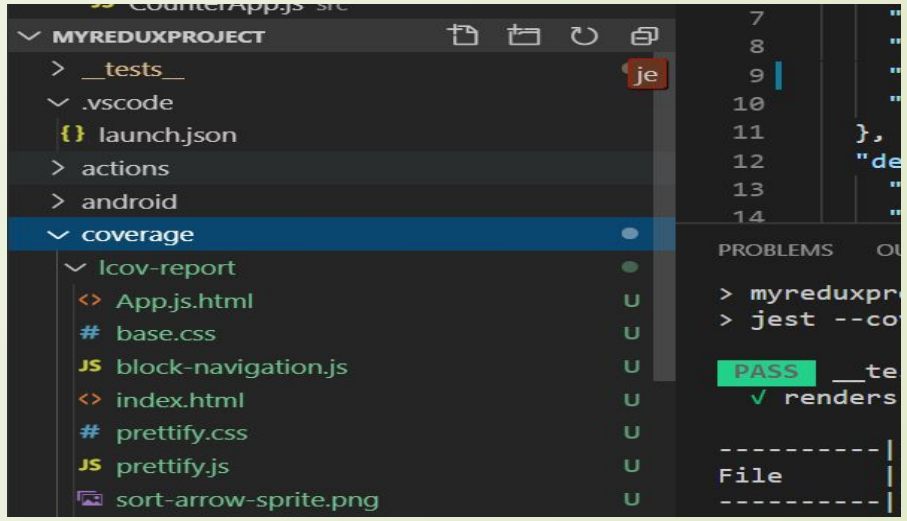
Scr. 1

```
{
  "name": "myreduxproject",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "android": "react-native run-android",
    "ios": "react-native run-ios",
    "start": "react-native start",
    "test": "jest --coverage",
    "lint": "eslint ."
  },
```

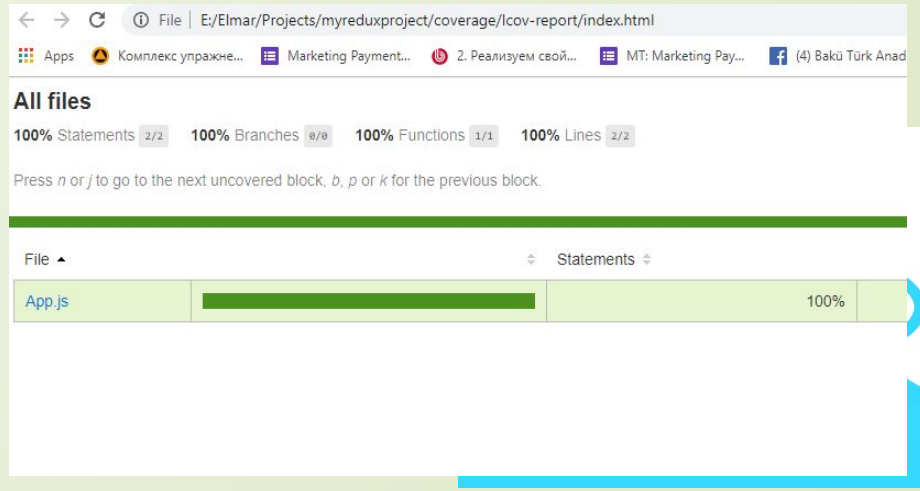
Scr. 2

```
> myreduxproject@0.0.1 test E:\Elmar\Projects\myreduxproject
> jest --coverage

 PASS  __tests__/App-test.js
  √ renders correctly (10ms)

----------|----------|----------|----------|----------|-------------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------|----------|----------|----------|----------|-------------------|
All files |      100 |      100 |      100 |      100 |                   |
 App.js   |      100 |      100 |      100 |      100 |                   |
----------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        2.954s, estimated 5s
Ran all test suites.
```

Scr. 3



Scr. 4



React Native Detox

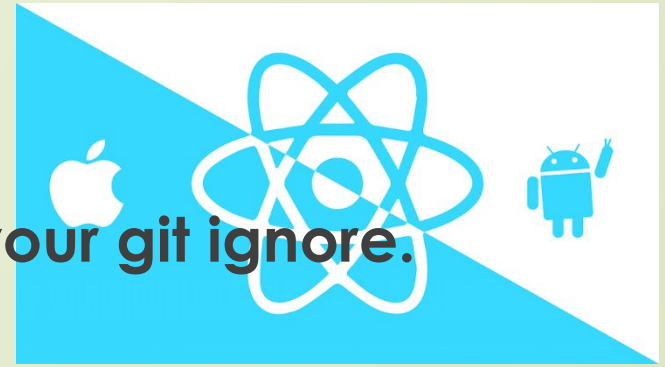
- Install detox cli: **npm install -g detox-cli**

- Install detox into your project:

npm install detox@12.11.1 --save-dev

For =>0.62 detox@16.2.1 --save-dev

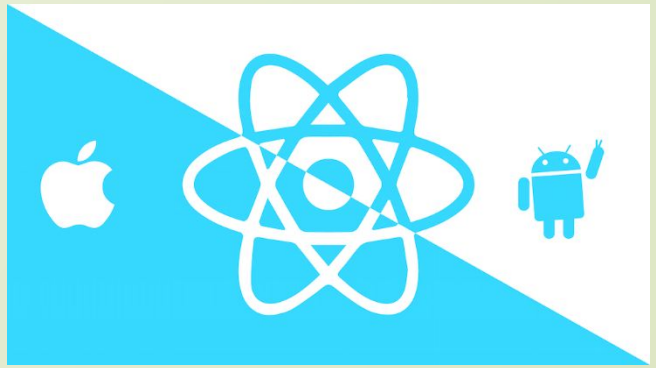
TIP: Remember to add the "node_modules" folder to your git ignore.



React Native Detox

- detox init -r jest
- In your root buildscript (i.e. build.gradle), register both google() and detox as repository lookup points in all projects:

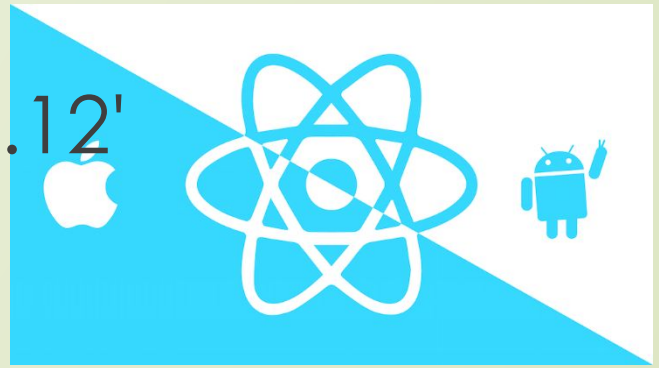
```
// Note: add the 'allproject' section if it doesn't exist
allprojects {
    repositories {
        // ...
        google()
        maven {
            // All of Detox' artifacts are provided via the npm module
            url "$rootDir/../node_modules/detox/Detox-android"
        }
    }
}
```



React Native Detox

- In your app's buildscript (i.e. app/build.gradle) add this in dependencies section:

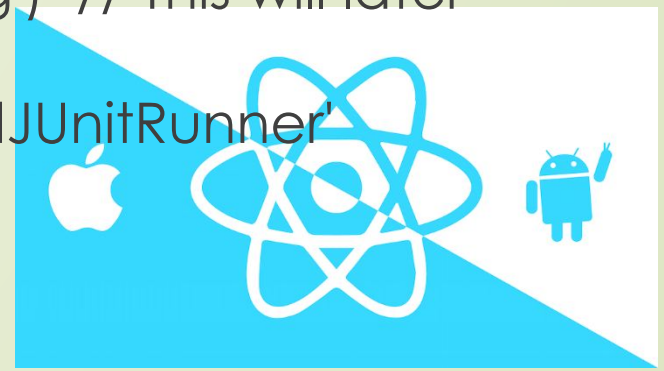
```
dependencies {
    // ...
    androidTestImplementation('com.wix:detox:+') { transitive = true }
    androidTestImplementation 'junit:junit:4.12'
}
```



React Native Detox

- In your app's buildscript (i.e. app/build.gradle) add this to the defaultConfig

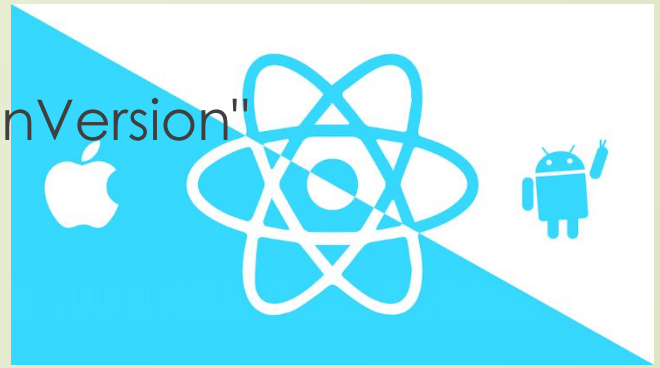
```
android {
  // ...


  defaultConfig {
      // ...
      testBuildType System.getProperty('testBuildType', 'debug')  // This will later be used to control the test apk build type
      testInstrumentationRunner 'androidx.test.runner.AndroidJUnitRunner'
  }
}
```



React Native Detox

- If your project does not already support Kotlin, add the Kotlin Gradle-plugin to your classpath in the root build-script (i.e.android/build.gradle):

```
buildscript {
    // ...
    ext.kotlinVersion = '1.3.10'

    dependencies: {
        // ...
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlinVersion"
    }
}
```

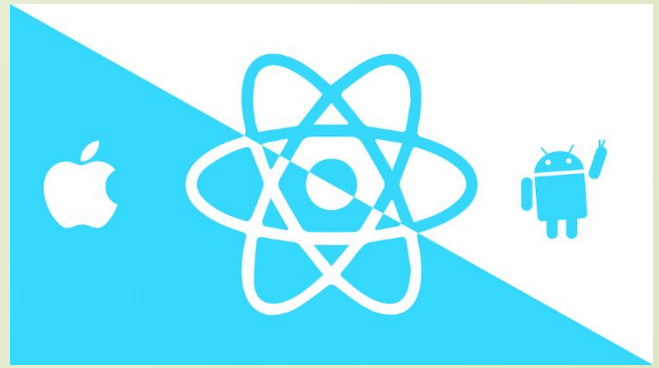


React Native Detox

- Create Android Test class

Add the file android/app/src/androidTest/java/com/[your.package]/DetoxTest.java and fill as in the detox example app for NR(on next page). Don't forget to change the package name to your project's.

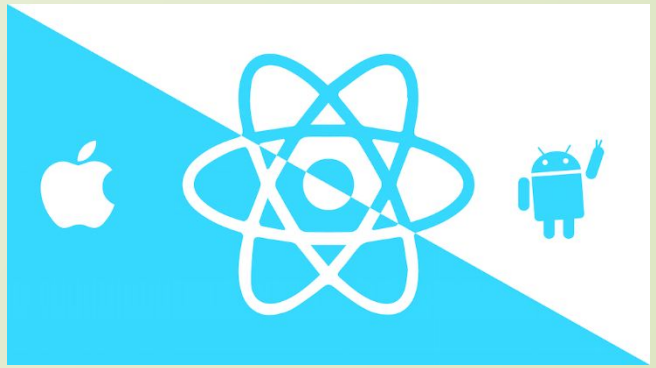
And add code below to your test file:



React Native Detox

And add code below to your test file:

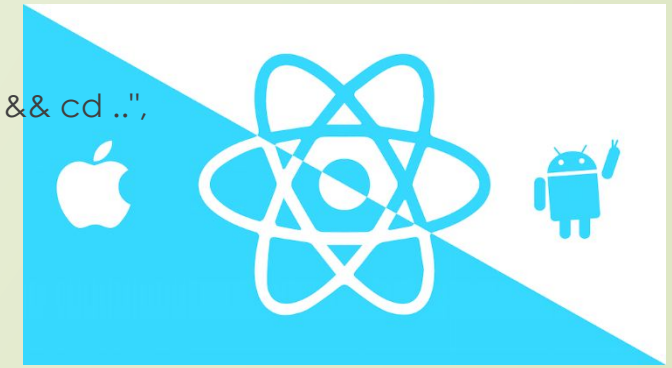
```
package com.example;

import com.wix.detox.Detox;

import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import androidx.test.ext.junit.runners.AndroidJUnit4;
import androidx.test.filters.LargeTest;
import androidx.test.rule.ActivityTestRule;

@RunWith(AndroidJUnit4.class)
@LargeTest
public class DetoxTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule<>(MainActivity.class, false, false);

    @Test
    public void runDetoxTests() {
```



React Native Detox

Insert into package.json file this code (“adb devices” in cmd):

```
 "detox": {
    "test-runner": "jest",
    "specs": "e2e",
    "configurations": {
      "android.emu.debug": {
        "binaryPath": "android/app/build/outputs/apk/debug/app-debug.apk",
        "build": "cd android && ./gradlew assembleDebug assembleAndroidTest -DtestBuildType=debug && cd ..",
        "type": "android.attached",
        "name": "emulator-5554"
            },
      "android.emu.release": {
        "binaryPath": "android/app/build/outputs/apk/release/app-release.apk",
        "build": "cd android && ./gradlew assembleRelease assembleAndroidTest -DtestBuildType=release && cd ..",
        "type": "android.attached",
         "name": "192.168.78.101:5555"

      }
  }
```



React Native Detox

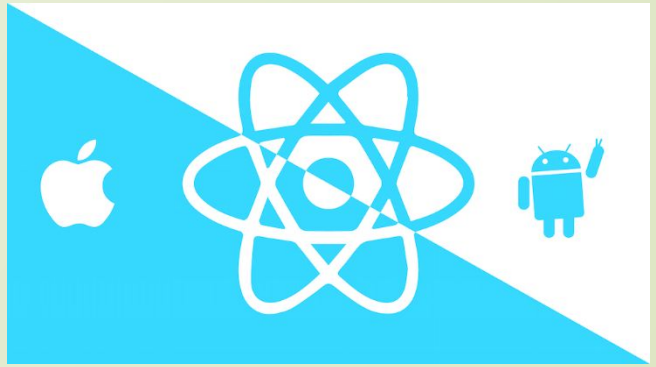
- Before run test:

./gradlew assembleAndroidTest

./gradlew assembleDebug

- Run test:

detox test -c android.emu.debug



React Native Detox

- How to make pause before any tap:

```
const sleep = duration =>
  new Promise(resolve => setTimeout(() => resolve(), duration));

it('should have welcome screen', async () => {
  await waitFor(element(by.id('mybutton')))
    .toBeVisible()
    .withTimeout(30000);
  await sleep(30000);
  await element(by.id('mybutton')).tap();

  await expect(element(by.id('mybutton'))).toBeNotVisible();
});
```

https://www.sitepoint.com/detox-react-native-testing-automation/