JEST DOCUMENTATION <h1>Introduction to JEST</h1> JEST is a framework used for JavaScript testing, it is built on top of ‘ Jasmine ‘ and is a popular framework for performing unit testing. Christoph Nakazawa, the brain of JEST developed it so that it provides support and simplicity in testing heavy web apps. JEST is mainly used for performing unit testing for REACT and React Native applications. Along with unit testing JEST can also be used for component testing. Unit testing is not very helpful for front-end software or components, as configuration of unit testing for front-end components requires more effort and is very time consuming. This is where JEST comes in handy, as it reduces the time consumption and complexity to a massive extent. <h1>Installation of JSET </h1><h2>•U4”är YARN</h2>
yarn add --dev jest
<h2>•U4”är å ÓÂö*ƒ*#ãÇVÃãÆÆ“æç Ò –ç7F ÆÂ Ò×6 ve-dev jest
<h1>Using JEST In NPM</h1> First open package.json file and change the configuration of the file as follows “Âð p> "scripts": {™™™™™“Â÷ ãÇ â ’ 'FW7B# "jest"™™™™™ ’ ÒÉ™™™™™™ <h2><a>First create a Test file</h2> Let's take a simple example like addition of two numbers as a test case, so create a file sum.js file “Ç7G&öæsç7VÒæ§3Â÷7G&öæsãÂð p> Function add(a,b){™™™™™“ÂöVÓãÂ÷ ãÇ ãÆVÓé™™™™™™ •&em> }™™™™™“Â÷ ãÇ é™™™™™™ //exporting add function™p> Module.exports = add“ÂöVÓé Now for testing create a file name sum.test.js ’ Ç7G&öæsç7VÒçFW7Bæ§3Â÷7G&öæsãÂ÷ ãÇ âòóÆVÓâ –× ÷ t your sum function const add= require('./add); test("adding 4and 2 expect to be 6",() => { expect(add(4,2)).toBe(6); }); In this file we take two parameters 4 and 2 and send it to the add function in sum.js to check if the function is working properly or not. We expect the output to be 6 if the output is 6 then the given function is working properly if not the test has failed i.e the taken function is not working properly. <h2>Execution of a test file</h2> We can execute the test file in two ways if we want to execute a single file we use the command
“Âð p>
npm test sum.test.js If we want to execute all the test files we use the command npm test <h1>JEST Matchers</h1> A matcher is used for creating assertions in combination with the expect keyword, we want to compare the output of our test to a value which is the output we are expecting. toBe : It matches the objects.
toEqual : It matches the value of the object to the return value.
<h2>Truthiness</h2> In tests you need to sometimes differentiate between undefined, null, false. toBeNull : matches only null
toBeUndefined : matches only undefined
toBedefined : It is the opposite of toBeUndefined
toBeTruthy : matches anything if the if statement is executed as true
toBeFalsy : matches anything if the if statement is executed as false
</

ul> Example: —FW7B, ‘null’ , () =>{ Const n = null</
p> except(n).toBeNull() }) <h2>Numbers</
h2>
toBeGreaterThan : If the return value of the
function is greater than the expected value</
li>
toBeGreaterThanOrEqual : If the return value of
the function is greater or equal to the value expected</
li>
toBeLessThan : If the return value of the function
is less than the expected value
toBeLessThanOrEqual :</
strong> If the return value of the function is less than or equal to the value expected</
li> Example: —FW7B, ‘ four minus 2 ‘, () =>{</
p> Const n = 4 - 2 except(n).toBeGreaterThan(1) }) <h2><a
id="_g4k1rs76cnzw">Strings</h2>
toMatch: </
strong>It is used check strings against regular expression
</
ul> Example: —FW7B, ‘ there is a christ in christopher ‘ , ()
=>{ except(‘christopher’).toMatch(/christ/) }) <h2><a
id="_eahc146s0ggf">Arrays and iterables</h2>
toContain : </
em>You can check if an item is present in an array or not iterables
</
ul> Example: “Â÷ ãÇ éconst anime = [‘ Naruto ‘, ‘ Baruto
‘, ‘Zero’, ‘Suzaku’, ‘Goku’] —FW7B, ‘ Naruto is mentioned in the list ‘ , () =>{</
p> except(anime).toContain(‘Naruto’) }) <h1></
a>Testing Asynchronous Code</h1> In JavaScript we can run code asynchronously,
so when we test JEST needs to know when the testing is completed so that it can move
to the next text. JEST has several ways to handle this <h2><a
id="_dkcwj4cr97bh">Promises </h2> •&WGW n a promise from your test, and Jest
will wait for that promise to resolve. If the promise is rejected, the test will fail.</
p> Example: ™math.js export
function multiply(a, b) { —&WGW n new Promise((resolve, reject) => { ™
if (typeof a !== 'number' || typeof b !== 'number') { ™—&V¦V7B†æPw Error('Both
arguments must be numbers')); ™ } else { ™—&W6öÇ`e(a * b); ™ }
  —Ò“³Â÷ ãÇ çÓÂ÷ ãÇ é“Â÷ ãÇ ãÇ7G&öæsé“Â÷7G&öæsãÂ÷ ãÇ ãÇ7G&öæsæÖ F,çFW7Bæ
strong> import { multiply} from './math'; test('multiply should resolve with
the correct result', () => { ’ &WGW n multiply(2, 3).then((result) => { ™
expect(result).toBe(6); —Ò“³Â÷ ãÇ çÒ“³Â÷ ãÇ çFW7B,vÐultiply should reject with an
error for non-numeric arguments', () => { expect.assertions(1);
return multiply(2, 'not a number').catch((error) => {
expect(error.message).toBe('Both arguments must be numbers'); });</
p> }); <h2>Async and wait</h2> “Â÷ ãÇ é
Alternatively, you can use async and await in your tests. To write an async test, use the
async keyword in front of the function passed to test. Example:</
strong> ™math.js export function divideAsync(a, b) {</
p> —&WGW n new Promise((resolve, reject) => { ™if (b === 0) { ™•
reject(new Error('Cannot divide by zero')); ™} else { ™—&W6öÇ`e(a / b);</
p> ™} ’ Ò“³Â÷ ãÇ çÓÂ÷ ãÇ é“Â÷ ãÇ ãÇ7G&öæsé“Â÷7G&öæsãÂð
p> math.test.js import { divideAsync } from './math';</
p> test('divideAsync should resolve with the correct result', async () => {
const result = await divideAsync(10, 2); expect(result).toBe(5); });</

p> test('divideAsync should reject with an error for division by zero', async () => { expect.assertions(1); try {   await divideAsync(10, 0); } catch (error) {   expect(error.message).toBe('Cannot divide by zero'); } }); <h2>Callback</h2> Example: ™math.js export function subtractAsync(a, b, callback) { —6WEF–ÖV÷WB„’ ÒfwC° {   ™if (typeof a !== 'number' || typeof b !== 'number') {   ™–6 ÆÆ& 0k(new Error('Both arguments must be numbers'));   ™ } else {   ™–6 ÆÆ& 0k(null, a - b);   ™ } ’ ÒÂ “³Â÷ ãÇ çÓÂ÷ ãÇ é“Â÷ ãÇ ãÇ7G&öæsé“Â÷7G&öæsãÂð p> math.test.js import { subtractAsync } from './math'; test('subtractAsync should return the correct result via callback', (done) => { —7V'G actAsync(10, 3, (error, result) => {   ™expect(error).toBeNull();   ™expect(result).toBe(7);   ™done(); ’ Ò“³Â÷ ãÇ çÒ“³Âð p> test('subtractAsync should return an error via callback for non-numeric arguments', (done) => { —7V'G actAsync(10, 'not a number', (error, result) => {   ™expect(error).toBeInstanceOf(Error);   ™expect(result).toBeUndefined();   ™done(); —Ò“³Â÷ ãÇ çÒ“³Â÷ ãÆƒ ãÆ id="_cuonlazeh4v9">Using Mock in JEST</h1> Mock functions allow you to test the connectivity between two codes by actually erasing the actual implementation of a function. There are two ways to create mock functions: Either by creating a mock function to use in the test code. Writing a manual mock to override a module dependency. All mock functions have this special .mock property which is where the data about how the function has been called and what the function returned is kept. The .mock property also tracks the value of this for each call. ™™™™™