the Master Course

{CODENATION}

Introduction to Javascript Testing

{CODENATION}

Learning Objectives

To understand what testing is and why we would do it.

To know what unit testing is and be able to run unit tests.

Testing

What does testing mean to you?

Testing

Checking that our code does what it is supposed to do.

CODENATION

{CODENATION}

Testing

Why would we want to write automated tests for our code?

Testing

For peace of mind that our code does what we expect it to.

{CODENATION}

Testing

Different types of testing:

Unit Tests
Integration Tests
Functional Tests.

Unit Testing

A unit is typically a module, a function, an object, a variable, an array etc.

A unit test checks the input and/or output of these units to make sure we get back what we expect!

Unit Testing

So suppose we have a function that adds two numbers. We expect the function to return the sum of the two numbers. We can write a test that will check this does in fact happen.

Unit Testing

To write automated tests we need to make use of third-party modules.

We need a test runner!

Unit Testing

A test runner is software that will run our JS tests for us.

The code you write to test your code, is basically what is happening when you submit an answer on Edabit, Codewars etc.

Unit Testing

We are going to use **Jest**, developed by the Facebook team.

{ CN }®

Unit Testing

Jest will run our tests for us, and also has methods we can use to write them.

{CN}®

Unit Testing

Everything is packaged nicely for us and it works straight out of the box. No additional config required. **Ace.**

Unit Testing

Open a new folder in VS Code called jest-demo. Inside this folder create a new app.js file.

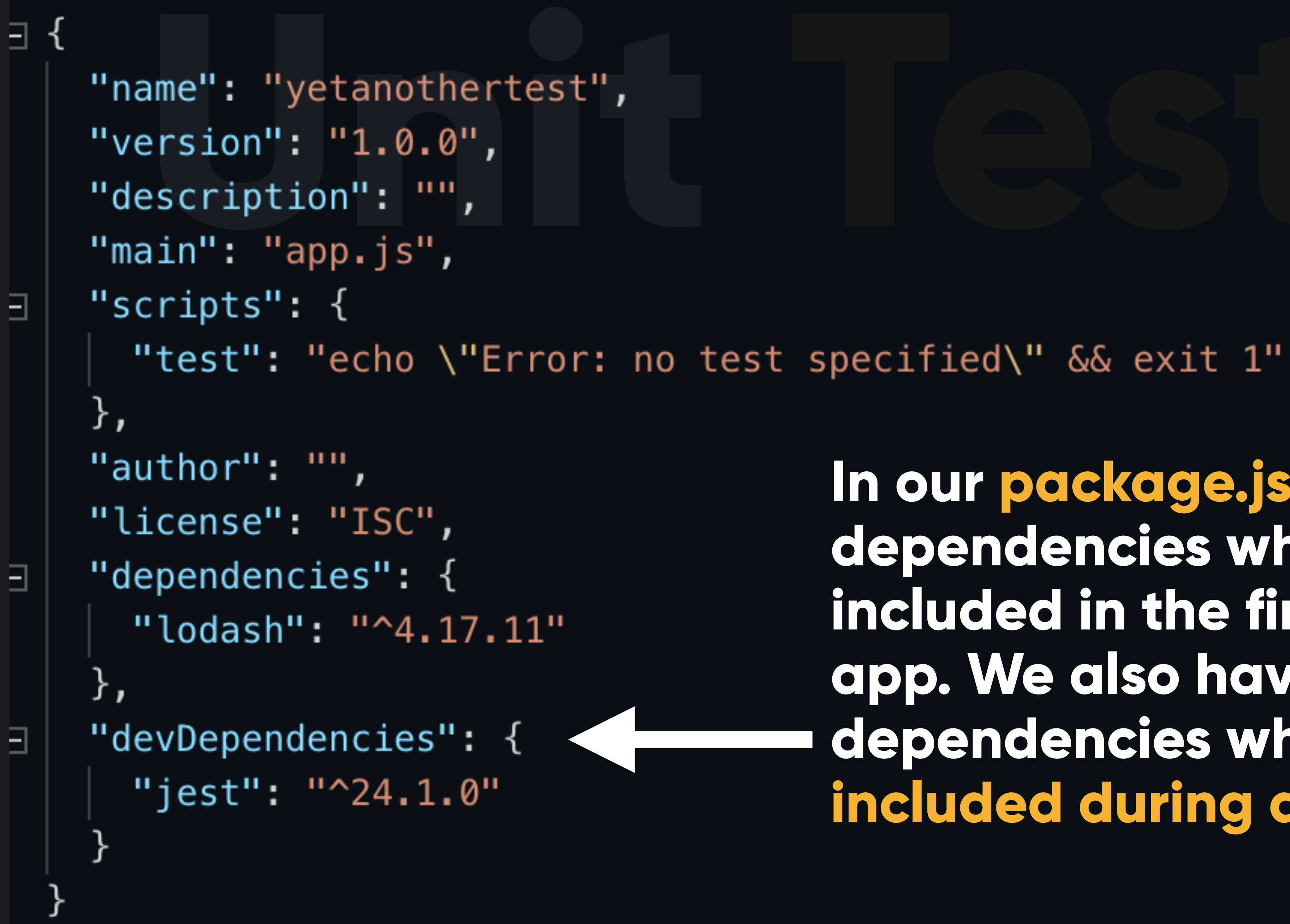
Remember to run npm init -y so we can package our project.

Unit Testing

Let's install jest in our project with NPM.
Enter the command:

```
npm install --save-dev jest
```

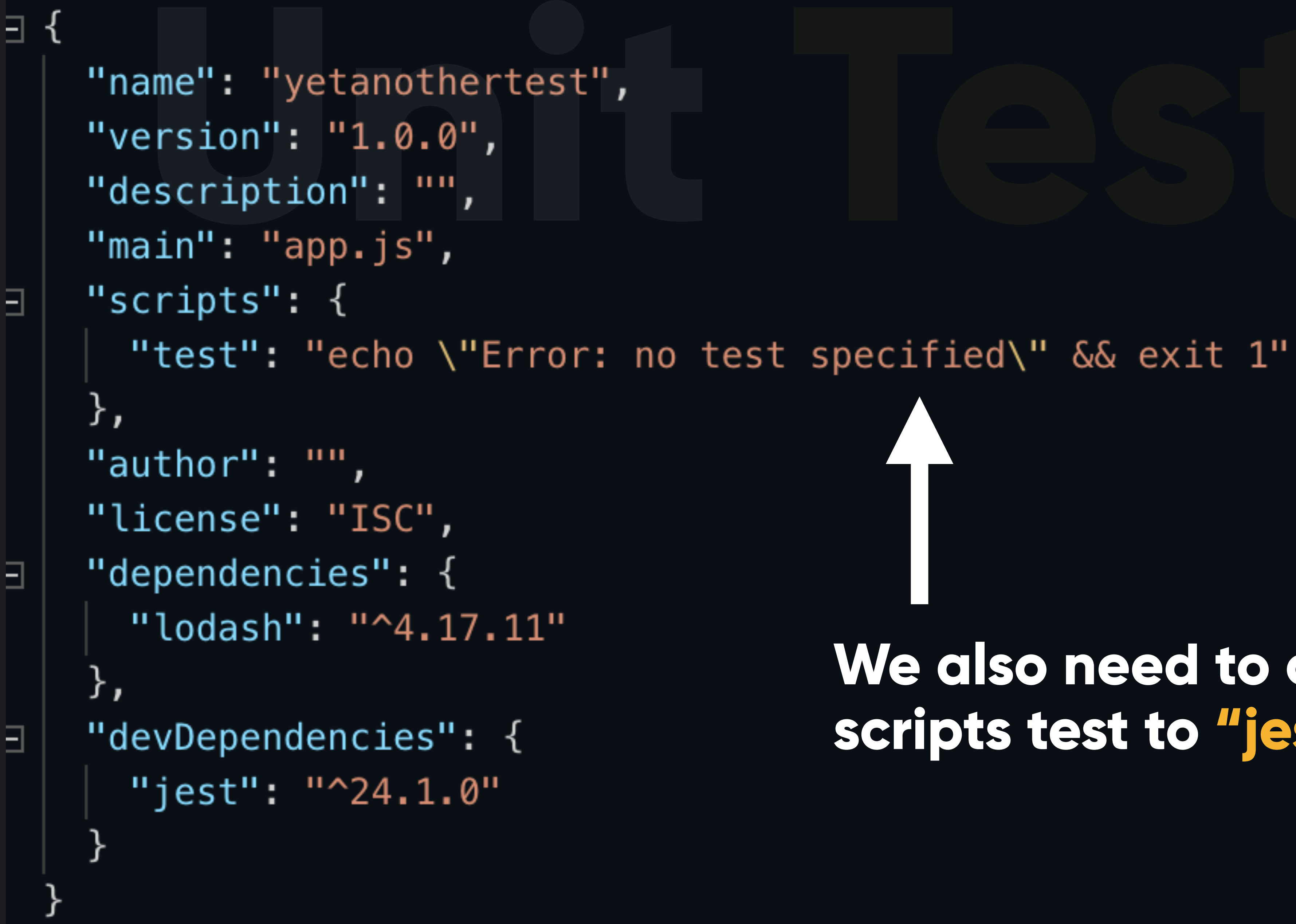
This installs it in our devDependencies.
Check your package.json file.



```json
{
  "name": "yetanothertest",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.11"
  },
  "devDependencies": {
    "jest": "^24.1.0"
  }
}
```

In our **package.json** file we have dependencies which are included in the final build of our app. We also have dependencies which are **only included during development.**

Unit Testing



```
{
  "name": "yetanothertest",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.11"
  },
  "devDependencies": {
    "jest": "^24.1.0"
  }
}
```

We also need to change our scripts test to "jest"

```json
{
  "name": "yetanothertest",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "jest"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.11"
  },
  "devDependencies": {
    "jest": "^24.1.0"
  }
}
```

Later, when we run npm test node will know we are referring to jest.

Unit Testing

When we create new **files for our tests** we give them the same name as the file we are testing, but include **.test** in the name.

CODENATION

Unit Testing

Example	**Example**	**Example**
app.js	main.js	ben.js
app.test.js	main.test.js	ben.test.js

{CODENATION}

Unit Testing

We can make our project folder cleaner by keeping our **test files** in a sub-folder called **tests**.

{CODENATION}

Unit Testing

Create a new **tests** sub-folder. Inside this folder create a new file called **app.test.js**

{CODENATION}

Unit Testing

Inside you app.js file **write a function** that adds two numbers together and **returns the sum** of those numbers.

Unit Testing

```
const add = (num1, num2) => {
        return num1 + num2;
}
```

{CODENATION}

Unit Testing

We're nearly set up. We need to **export** our functions, variables, arrays etc from our app.js file.

Unit Testing

Then we need to require our app.js file in our test file. Can you remember how?

List our functions etc in an object called module.exports = { } at the end of our app.js file.

require our app.js file in the test file using the require method by
const app = require('../app.js')

{CODENATION}

Unit Testing

The test( ) method

Takes two parameters:

1. A string which describes the test
2. A function where we make our assertions.

{CODENATION}

Unit Testing

```
test('should equal 5 when passed 2 and 3', ( ) => {
        //We make our assertions here.
});
```

{CODENATION}

Unit Testing

The expect( ) function:

We use the expect( ) function inside our test( ) method, and we use it every time we want to test a value.

{CODENATION}

Unit Testing

We use expect( ) along with a "**matcher**" function to assert something about a value. Let's have a look.

{CODENATION}

Unit Testing

```
test('should equal 5 when passed 2 and 3', ( ) => {

    expect(app.add(2,3)).toBe(5);

});
```

Function to test from our app.js file

A matcher function

Unit Testing

There are lots of matcher functions. Here's a few common ones:

.toBe( )
.toHaveLength( )
.toEqual( )
.toContain( )
.toBeDefined( )
.toHaveBeenCalled( )

We can even add .not to our matcher functions

.not.toBe( )

{CODENATION}

Read the docs!

If you want to understand how each **matcher function** works.

Say I have an array in my app.js file:

```
let myArray = ['Dan', 'Stuart', 'Ben'];
```

...and I want to check whether it contains 'Stuart'

```
 test('should contain Stuart', ( ) => {

       expect(app.myArray).toContain('Stuart');

 });
```

{CODENATION}

```
test('should contain Stuart', ( ) => {

    expect(app.myArray).toContain('Stuart');

});
```

It almost reads in perfect English.

{CODENATION}

Unit Testing

Challenge: write a test which will check your add function works as expected. I'll leave the next slide up to give you a few hints.

{CODENATION}

Unit Testing

```
test('A string to describe the test', ( ) => {

    expect(your function call here).toEqual(some value);

});
```

{CODENATION}

Unit Testing

Now we have set up our tests. It is time to run them! Exciting.

Using the command npm test in the console.

{CODENATION}

Unit Testing

```javascript
const app = require('../app.js');


test('should equal 5 when 2 and 3 are passed', () => {
    expect(app.add(2,3)).toBe(5);
})

test('should contain Stuart in myArray', () => {
    expect(app.myArray).toContain('Stuart');
})
```

```
Dans-MacBook-Pro:LearningJest dan$ npm test

> learningjest@1.0.0 test /Users/dan/codenation/LearningJest
> jest

 PASS  tests/app.test.js
  ✓ should equal 5 when 2 and 3 are passed (4ms)
  ✓ should contain Stuart in myArray

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.824s
Ran all test suites.
```

Unit Testing

Let's make the first test fail on purpose just to see what that looks like. I'll change the .toBe( ) value from 5 to 6.

{CODENATION}

Unit Testing

```
Dans-MacBook-Pro:LearningJest dan$ npm test

> learningjest@1.0.0 test /Users/dan/codenation/LearningJest
> jest

 FAIL  tests/app.test.js
  × should equal 5 when 2 and 3 are passed (5ms)
  ✓ should contain Stuart in myArray

  ● should equal 5 when 2 and 3 are passed

    expect(received).toBe(expected) // Object.is equality

    Expected: 6
    Received: 5

      3 |
      4 | test('should equal 5 when 2 and 3 are passed', () => {
    > 5 |     expect(app.add(2,3)).toBe(6);
        |                          ^
      6 | })
      7 |
      8 | test('should contain Stuart in myArray', () => {

      at Object.toBe (tests/app.test.js:5:26)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.808s
Ran all test suites.
npm ERR! Test failed.  See above for more details.
```

Unit Testing

This gives us a really nice report of what went wrong. If we know the test SHOULD pass, we will need to fix our code.

{CODENATION}

Unit Testing

Unit tests should be dead simple. Don't try to over-complicate things!

{CODENATION}

You can group tests together in a describe( ) block

describe( ) takes two parameters. A string and a function.

```
describe('description for the test group', ( ) => {

//Grouped tests go here

});
```

This is very much a game of red and green.

Extra info: test( ) has an alias it( ) and they both do the same thing.

```
it('should contain Stuart', ( ) => {
    expect(app.myArray).toContain('Stuart');
});
```

{CODENATION}

it( ) reads quite nicely

```
it('should contain Stuart', ( ) => {

    expect(app.myArray).toContain('Stuart');

});
```

"It should contain Stuart. I expect this to contain Stuart."

Challenges

Create functions and test on the following…

- To make sure what is returned is not 'null'
- A value that is truthy
- A value that is not falsy
- Create a function that creates an object with 2 properties, test to makes sure that the objects properties are equal to your test function
- A function that will return items in an array with 6 or more characters
- Can you refractor any of your code?

Challenges

Create functions and test on the following…

- **Convert a number to a string**
- **Display the correct planet with the number order it is away from the sun**
  (planet(3) //will return 'Earth'**)**
- **Count the amount of students present in the class. With an array or boolean values, count how many students are present (true = present)**
  ([true, true, true, false, true] //will return 5**)**
- **Square every digit and concatenate them (must return an integer)**
  (squareDigi(34) //will return 916**)**

{CODENATION}

Challenges

Create functions and test on the following…

- **Given a year return back the century it is in**

  (century(1705) //will return 17**)**

- **With an array of ones and zeroes, convert the equivalent binary value to an integer**

  (binary([0, 0, 0, 1]) //will return 1**)**

  (binary([0, 1, 0, 0]) //will return 4**)**

{CODENATION}

Revisiting Learning Objectives

To understand what testing is and why we would do it.

To know what unit testing is and be able to run unit tests.