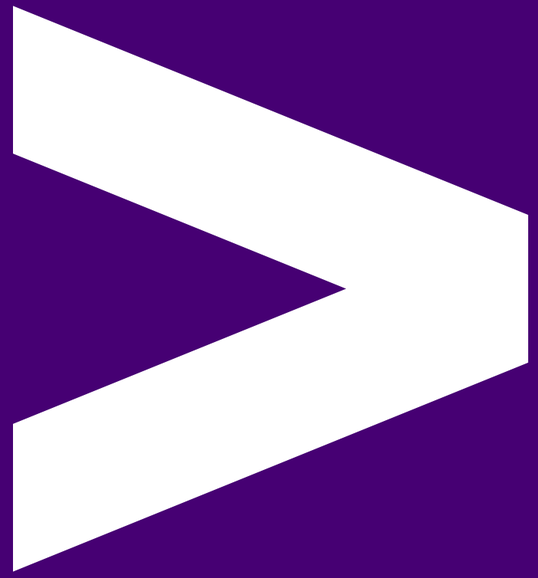
Next Gen Engineering
School of Tech - INDIA

Unit Testing



Overview

- Why do we test?
- How do we test in TypeScript?
- Using Jest
- Happy tests
- Unhappy tests

Objectives

- See the value of unit testing
- Run some pre-written example tests to check code
- Understand the nesting of `describe`, `it` and `test` blocks
- Fill in some happy tests in `maths-utils`
- Fill in some unhappy tests in `maths-utils`
- Write some basic unit tests

Why we test

Testing software increases your confidence and your client's confidence in the product.

Why we test

Testing software increases your confidence and your client's confidence in the product.
Better tests enable you to develop new features, experiment and improve your codebase more easily and with lower risk.

Why we test

Testing software increases your confidence and your client's confidence in the product.
Better tests enable you to develop new features, experiment and improve your codebase more easily and with lower risk.
A good, repeatable testing process is critical to ensuring sustainable, continuous delivery.

Why we test

Testing software increases your confidence and your client's confidence in the product.
Better tests enable you to develop new features, experiment and improve your codebase more easily and with lower risk.
A good, repeatable testing process is critical to ensuring sustainable, continuous delivery.
Well tested software means the team goes home on time.

Why we test

Testing software increases your confidence and your client's confidence in the product.
Better tests enable you to develop new features, experiment and improve your codebase more easily and with lower risk.
A good, repeatable testing process is critical to ensuring sustainable, continuous delivery.
Well tested software means the team goes home on time.
Reliable software means no call out or overtime required!

Testing

Testing

Which parties care that your code works as expected?

Testing

Which parties care that your code works as expected?
- You

Testing

Which parties care that your code works as expected?
- You
- Your team

Testing

Which parties care that your code works as expected?
- You
- Your team
- Your client

Testing

Which parties care that your code works as expected?
- You
- Your team
- Your client
- Your users

What is Unit Testing?

Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

What is Unit Testing?

What is Unit Testing?

Here's a "unit":

```
function add(a, b) {
  return a + b
}
```

What is Unit Testing?

Here's a "unit":

```
function add(a, b) {
  return a + b
}
```

A form of testing that aims to validate "units" of code.

What is Unit Testing?

Here's a "unit":

```
function add(a, b) {
  return a + b
}
```

A form of testing that aims to validate "units" of code.
A `Unit` is the smallest amount of code that can be executed in isolation from other code.

What is Unit Testing?

Here's a "unit":

```
function add(a, b) {
  return a + b
}
```

A form of testing that aims to validate "units" of code.
A `Unit` is the smallest amount of code that can be executed in isolation from other code.
In essence it relies on nothing else in order to function.

Why do we do Unit testing?

Why do we do Unit testing?

It helps us understand the design of the code.

Why do we do Unit testing?

It helps us understand the design of the code.
Unit tests are efficient and require minimal effort to re-run.

Why do we do Unit testing?

It helps us understand the design of the code.
Unit tests are efficient and require minimal effort to re-run.
Having them eliminates questions about the code.

Why do we do Unit testing?

It helps us understand the design of the code.
Unit tests are efficient and require minimal effort to re-run.
Having them eliminates questions about the code.
They are "Living Documentation".

What do we need to test?

Let us consider three main aspects to think about when approaching testing:

1. Write the test around how you expect a unit to be used.
2. Write multiple tests covering the different code paths (explained in the next slide).
3. Write tests that will test different inputs.

What do we test? - Code Paths

- A single "happy path" test probably won't give much coverage if the code has significant logic
- When testing we want all the code "paths" to be covered
- If every `if/else` is a path, then we need to cover all the "branches" of those statements
- If every `loop` is a path, we need to test each loop for "zero", "one" and "many" iterations

What do we test? - Different inputs

- It's essential to use different test inputs to ensure different code paths are being hit
  - i.e. different parameter values
- Ideally each test should verify a different case
- Ideally each test should verify a specific case
- Ideally tests shouldn't be repeating each other / overlapping very much.

Emoji Check:

Do you understand why and what we test with unit tests?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

The general structure

Once we have our test cases, use the Three "A"s:

The general structure

Once we have our test cases, use the Three "A"s:
> Arrange ...some data

The general structure

Once we have our test cases, use the Three "A"s:
> Arrange ...some data
> Act ...on the target function (or unit)

The general structure

Once we have our test cases, use the Three "A"s:
> Arrange ...some data
> Act ...on the target function (or unit)
> Assert ...the answer was correct

Structure of a Jest test

This is illustrated in `examples/skeleton.text.ts`:

```
test('the thing being tested should do xyz', () => {
  // Arrange
  ...
  // Act
  ...
  // Assert
  ...
});
```

Arrange

Coming back to our unit of code, what do we need to arrange?

```
function add(a, b) {
  return a + b
}
```

If we look at that, there are two parameters required as input:

```
// Arrange
const firstParam = 10
const secondParam = 5
```

Act

What is it we are acting on? It's our function:

```
function add(a, b) {
  return a + b
}
```

So we act like so:

```
// Arrange
const firstParam = 10
const secondParam = 5

// Act
const result = add(firstParam, secondParam)
```

Asserting

What should we be asserting against, after we Act?

```
// Act
const result = add(firstParam, secondParam)
```

Asserting

When thinking about the assertion, you want to think about what you want that output to be, what SHOULD this function return?
Once you know that you can create a variable to represent it.

Asserting in Jest

Here is our complete set of the three As with our expected result:

```
// Arrange
const firstParam = 10
const secondParam = 5
const expectedResult = 15

// Act
const result = add(firstParam, secondParam)

// Assert
expect(result).toBe(expectedResult)
```

Never mind the theory, lets do some!

On the following slides we will bring that theory together by:

- Making sure jest is in our dependencies in `package.json`
- Running a skeleton `maths-utils` we have set up already
- Filling in some of the tests in `maths-utils`

Unit testing with Jest

Jest is a Javascript / Typescript unit test framework
It provides a mechanism for test automation
The docs are a great place for reference

Adding the Jest dependency

> Our examples folder has this done for you already

First we need the jest framework installed in a project; We can install the test runner `jest`. In order to work nicely with TypeScript we need a couple of supporting packages, `@types/jest` and `ts-jest`:

```
npm install --save-dev jest @types/jest ts-jest
```

It must also be listed in the scripts block, like so:

```
"scripts": {
  "test": "jest"
}
```

Configuring Jest

Create a file named `jest.config.js` with the following contents:

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node'
}
```

Running tests with Jest

Then you can run all the supplied tests in the project directory using:

```
npm test
```

You can run a single module of tests using:

```
npm test my-file
```

Note only the name of the module is used, so only `my-file`, without the `.ts` extension. Else you must specify `my-file.test.js`, to make sure we run the test file!

Task: Running tests with Jest - 2 mins

Run the blank `maths-utils.test.ts` in the `./exercises` folder using:

```
cd ./exercises
npm install
npm test maths-utils
```

Note only the name of the module is used, so only `maths-utils`, without the extension. Else you must specify `maths-utils.test.ts`.
This should print out some nice default logs.

Emoji Check:

Do you understand how to structure a unit test with Jest?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Task - Adding maths tests - 10 mins

We will split into small groups. Help each other. Everyone in the group do this yourselves for practice!

- Change directory into the `exercises` folder and `npm install`
- Run `maths-utils.test.ts` test using `npm test maths-utils`
- Work on the `add` tests only
- Fill in the missing Arrange, Act and Assert using the code blocks from the previous slides
- Re-run your passing tests

Emoji Check:

How did you find writing your first unit tests?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Comparing values

Jest has a lot more matchers than `toBe()`. Jest has many matchers
Some few of note are:

- `.toEqual()`, which is slightly different to `.toBe()`
- `.toBeTruthy()`, `.toBeFalsy()`, `.toBeUndefined()`, and `.toBeNull()`
- `.not` to negate, i.e. `not.toBeUndefined()`
- `.toBeStrictEqual()`, `expect.arrayContaining()` and `expect.objectContaining()`

As already seen in javascript, the handling and matching of arrays and objects needs special consideration. Jest has us covered.

Notes on matchers

Some of those are custom matchers, and we use them like so;

```
const animal = { mammal: true, flippers: 2}

expect(animal).toEqual(
    expect.objectContaining({
        flippers: 2
    })
)
```

This would pass as the object does indeed contain the `key:value` pair `flippers: 2`

Task - Object and Array tests - 15 mins

We will split into small groups. Everyone in the group do this yourselves for practice!

- Run the supplied `zoo-animals.test.ts` test with `npm test zoo-animals`
- Use the array and object friendly matchers to validate the zoo animals
  - There are hints in the files
- Re-run your tests

Emoji Check:

How did you find using object and array matchers in your tests?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

It's not just a 'test'

Jest has a few methods so we can semantically arrange our tests:

- `describe()`
- `it()`
- `test()`

`it()` is an alias for `test()`. We can use these interchangeably, and nest them, to make readable code.
Example on next slide where we discuss test names.

Naming a Jest test

In the `add` example, can anyone think of a good name for the test? Is this one good?

```
describe('invoking the add function with parameters', () => {
  it('returns 15 when invoked with 10 and 5', () => {
    // Arrange
    const firstParam = 10
    const secondParam = 5
    // Act
    const result = add(firstParam, secondParam)
    // Assert
    expect(result).toBe(15)
  });
})
```

Naming of tests

A few things to think about when naming a test:

- What is the test actually testing?
- Do we need to refer specifically to the inputs being used?
- Is there anything else that might affect the test we should use?

Naming a test

Here we have named two tests using the `add` function, are these names descriptive enough?

```
describe('When invoking the add function', () => {
  it('adds two numbers together', () => {
    // ... test goes here
  })
  it('adds two strings together', () => {
    //... another test goes here
  })
})
```

Test name examples

In this `place.test.ts` example, are these good test names?

```
test('When place is called the counter will go to'
      'the bottom of the column', () => {
    // ...
})

test('When place is called on the same column twice'
      'then the counter goes to the bottom free slot', () => {
    // ...
})
```

Test name rules

Your test names are the documentation of your system.

> If you have descriptive test names and the variables and functions are also named accurately - you should not need comments in your code.

Emoji Check:

How do you feel about naming tests?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Unhappy tests

All the tests we have done so far have been testing "Happy" cases.
That is - they all cover scenarios that we expect and want to happen.
However code does not always work the way we want, and we need to know we can handle those cases.
We can, and should, also test for the negative or "Unhappy" cases.

Some code might go boom!

When code fails we will often receive an exception (aka an error). We can catch these and inspect them to see what happened.
There are two ways we can do this using jest. Usually we write our `expect` in a way that expects the error, but you will sometimes see a `try/catch` block around the code instead.
Lets take a look at each of these now.

Expect on the function

- In order to catch the exception, we need to wrap up our tested function
- This is because the error will stop us before we finish the test
- We can use `toThrow()` to do this:

```
const myWrapperFn = () => {
  myBadFunction(badValue)
}

expect(myWrapperFn).toThrow('and contain this text')
expect(myWrapperFn).toThrow(/and contain this regex text/)
expect(myWrapperFn).toThrow(new Error('exact text'))
```

Try Catch error block

- We wrap the function up in a try block.
- We then write a catch block and declare the variable name for our exception
- We can then inspect that exception and make some assertions on it

```
try {
  // some code that explodes
  myBadFunction(badValue)
  fail('Expected the thing to go boom') // you need this in ca
  // test from passing!
} catch (exception) {
  expect(exception.message).toContain('some text') // or
  expect(exception.message).toEqual('exact text') // or
  expect(exception).toStrictEqual(new Error('exact text'))
}
```

Task - Unhappy tests - 15 mins

In breakout rooms lets take a look at the `safeMultiply` unhappy tests:

- Run the supplied `maths-utils.test.ts` test with `npm test maths-utils`
- Work on the `safeMultiply` tests
- Verify the negative inputs on the `safeMultiply` function
- Re-run your tests

Emoji Check:

Do you understand how we go about writing tests for 'unhappy' cases?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

More tests

Now we've seen a few things we can do with tests, we'll add some more test code to one of the examples.
Use any of the methods and functions we've practiced earlier.
Task on next slide

Task - Adding tests - 15 mins

- Run the supplied `place.test.ts` with `npm test place`
- Check the test passes
- Write additional happy case unit tests covering:
  - When a column has a second / third / fourth piece inserted?
- Write an additional unhappy case unit tests covering:
  - When a column is full and we attempt to put a fifth piece in it?

Emoji Check:

How did you feel about adding extra cases to the tests?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😄 Yes, enough to start working on it collaboratively

Common test requirements

We've seen our `place` tests do several things over and over now.
We have also seen that we have had to create the same variables for each test.
Lets take a look at how we can prevent repeating ourselves and introduce some setup and teardown.

Setup and Teardown

All test frameworks support the notion of a `setup` to do before each (`beforeEach`) test and a `teardown` to do after (`afterEach`).
They also support a `beforeAll` to initialise the whole system and `afterAll` to clean up.
Jest has a good section on this in Jest Setup & Teardown

Setup and teardown with Jest

- Here's an example where we want to add an item to an array, but we want the array to start with 'Early Bird' each time
- We can do this with beforeEach()

```
let arrayOfAnimals = []

beforeEach(() => {
  //Every test gets a clean board
  arrayOfAnimals = ['Early Bird']
});

test('Remove item from array', () => { /*...*/ });

test('Add item to array', () => { /*...*/ });
```

Can anyone explain why we can't just create the `arrayOfAnimals` with `'Early Bird'` in it?

Scoping with Describe

We can do this at the top level of the test file (so it happens for every `describe` block and every `test`)
But we can also use these for individual `describe` blocks then we can have setup specific to them.
This allows us to be really flexible with test setup.

Scoping example

```
beforeEach(() => { // Applies to all tests in the file
  arrayOfAnimals = ['Early Bird']
});

describe('when making valid moves', () => {

  // Applies only to tests in this describe block
  beforeEach(() => {
     arrayOfAnimals.push('Timely Hippo')
  });

  test('Add item to animalArray', () => {
    //Result will be ['Early Bird', 'Timely Hippo', 'addedItem
  });
})
```

Task - Add Setup and Teardown - 10 mins

- Go to `animal-array.ts` and take a look at what the function is doing
- Open the `animal-array.test.ts` tests file.
- Some tests are set up already, follow the tasks in the file and use the beforeEach() method to get them passing and cleaned up

Emoji Check:

Do you understand what setup and teardown methods are for?

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Before All and After All with Jest

`beforeAll()` and `afterAll()` allow us to set things up for our tests or clean up afterwards.
Some real-life use cases might be:

- start or kill a database connection
- create and destroy a database table...

Before All and After All with Jest

```
let animalArray = []

beforeAll(() => { // this only happens once
  animalArray.push('Early Bird')
});

afterAll(() => { // this only happens once
  animalArray.push('Bird has gone')
});
```

continues on next slide

Before All and After All with Jest continued

```
test('Add item to animal array', () => {
  // Adds an additional item to the array
  const anotherAnimal = 'Worm that got caught'

  animalArray.push(anotherAnimal)

  expect(animalArray)
    .toEqual(['Early Bird', 'Worm that got caught'])
});

console.log(animalArray)
// ['Early Bird', 'Worm that got caught', 'Bird has gone']
```

Offline Task - Tests for Noughts and Crosses

- Add a `package.json` to your folder
  - You can do this with `npm init` or copy one from the examples
  - Use `npm init` inside your noughts and crosses folder, answer each of the questions it asks
    - Most can be blank, but add your name, git repo and `jest` as the tester
- Use `npm install --save-dev jest` to install the Jest testing framework

Offline Task - Tests for Noughts and Crosses

- In your `package.json` file, change the value inside the `scripts` key to be `"test": "jest"`
- Extend your `academy.ts` and `connectors.ts` to have any new functions you made in `module.exports` at the end of the files
- Make a new `academy.test.ts` file and import `academy.ts` with `const academy = require('./academy');`

Offline Task - Tests for Noughts and Crosses

- Start by identifying your units for testing
- Think about what scenarios need testing for each unit
- Create a new git branch called `unit-tests`

Offline Task - Tests for Noughts and Crosses

- Using Jest - add unit tests around your noughts & crosses code
- Include tests for failing scenarios - check out the `toThrow(Error)` matcher in the Jest documentation
- Create a Pull Request on your repo once you have some tests and have someone else review them

Overview - recap

- Why do we test?
- How do we test in TypeScript?
- Using Jest
- Happy tests
- Unhappy tests

Objectives - recap

- We have understood the value of unit testing
- We Ran some pre-written example tests to check code
- We understand the nesting of `describe`, `it` and `test` blocks
- We filled in some happy tests in `maths-utils`
- We filled in some unhappy tests in `maths-utils`
- We have written some basic unit tests

Emoji Check:

On a high level, do you think you understand the main concepts of this session? Say so if not!

1. 😢 Haven't a clue, please help!
2. 😕 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively