ENGINEERING

COMPSCI 732

SOFTENG 750

Testing JavaScript & React apps

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

JEST & basic unit testing

Unit testing JavaScript applications

- Like any other widely-used language, we can use various libraries to write *unit tests* – which enable us to test small pieces of code in isolation
- Several popular frameworks:
  - **JEST**
  - MochaJS
  - Jasmine
  - Karma
  - Puppeteer

JEST

- JEST is maintained by Facebook
- Built-in support for mocking, snapshot testing
- Integrates easily with React & other frameworks
  - For Vite apps, **vitest** is functionally equivalent to Jest, with much easier integration
- Extensive documentation

JEST setup (backend / non-Vite projects only)

- To install JEST in a project:

```
npm install --save-dev jest babel-jest @babel/core @babel/preset-env @types/jest
```

Dev dependencies won't be made available in production environments

JEST itself

Integration with Babel, to allow advanced JS features during testing

Allows proper code completion in test files in VS Code

- In package.json:

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

JEST setup (backend / non-Vite projects only)

- babel.config.js
  - Add the file with the following contents to the project root:

```
module.exports = {
    presets: [
        [
            '@babel/preset-env',
            {
                targets: {
                    node: 'current',
                },
            },
        ],
    ],
};
```

Old pre-ES6 syntax, before `import` and `export` were added to JavaScript. We can't use the more modern syntax here, as this file is read *before* Babel loads and therefore can't be transpiled.

(for interest: the pre-ES6 version of `import` is `require()`).

Running JEST tests

- Add our tests to files named `*.test.js`, optionally in folders named `__tests__`, anywhere in our source tree
  - JEST will be able to discover and run any so-named files
- Now everything is set up, we can run tests in several ways:

Goal	Syntax	Examples
Run all tests	`npm test`	
Run tests matching a pattern / filename	`npm test -- <pattern>`	`npm test -- my-file` `npm test -- /path/to/my-file.js`
Run tests matching test descriptions	`npm test -- -t <desc>`	`npm test -- -t "render test"`
Watch mode	`npm test -- --watch` *or* `npm test -- --watchAll`	← Watches uncommitted files ← Watches all files

Watch mode

- JESTs watch mode will run in the background and watch for changes to files
- When a change is detected, tests will be rerun

Basic tests

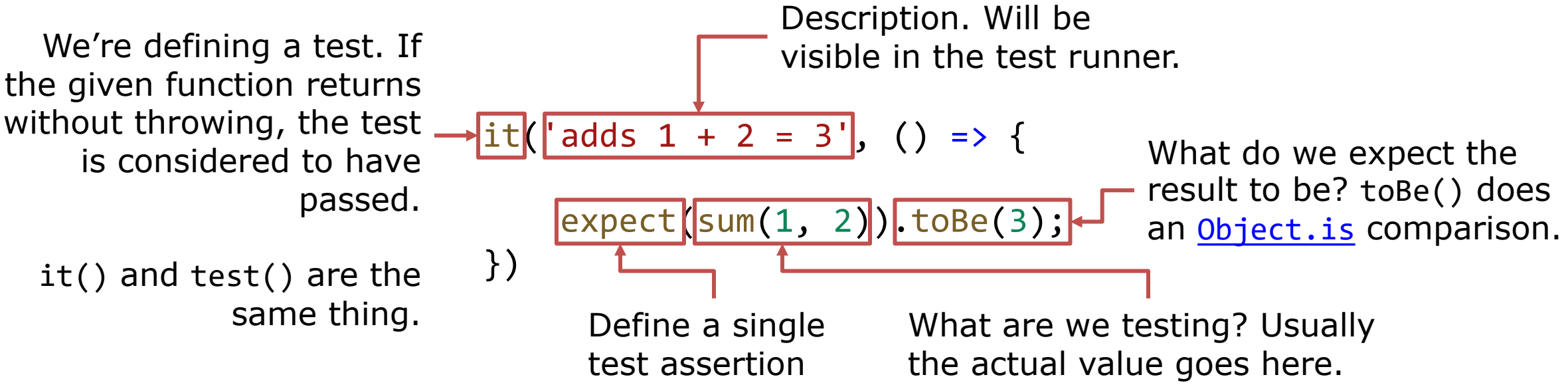
- Suppose we're trying to test a function called `sum()`, which takes any number of arguments and adds them together.
- Here's a first unit test for this function:

```
it('adds 1 + 2 = 3', () => {

    expect(sum(1, 2)).toBe(3);
})
```

Basic tests

- Suppose we're trying to test a function called `sum()`, which takes any number of arguments and adds them together.
- Here's a first unit test for this function:



Commonly used matchers

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
ENGINEERING

Matcher	Description
`toBe()`	Checks if two objects are the same
`toEqual()`	Checks if two objects are equal. Recursively checks properties.
`not`	Negates any other check on this list
`toBeTruthy() / toBeFalsy()`	Checks for "truthyness" or its inverse
`toBeGreaterThan() / toBeLessThan() / toBeGreaterThanOrEqual() / toBeLessThanOrEqual()`	Mathematical >, <, ≥, ≤ operations
`toBeCloseTo()`	For comparing floating point numbers
`toMatch()`	For matching a regular expression
`toContain()`	Checks whether a given array contains a given element
`toThrow()`	Checks whether a given function throws a given error when called

… and many more!

Setup / teardown

- Sometimes we might want to run some code:
  - Once, before *all* tests in a file / group
    - E.g. starting a server, connecting to a database
  - Once, after *all* tests in a file / group
    - E.g. closing connections that were obtained above
  - Before *each* test in a file / group
    - E.g. initializing objects, ensuring each test is working with the same data as a starting point
  - After *each* test in a file / group
    - E.g. restoring state, clearing mocks (see slides on Mocking)

Setup / teardown

- `beforeAll()`: runs a given function once, before all tests in a group
- `afterAll()`: runs a given function once, after all tests in a group
- `beforeEach()`: runs a given function before each test in a group
- `afterEach()`: runs a given function after each test in a group
- Example: This code will ensure that the bank account under test always has the correct starting balance at the beginning of each unit test in the file

```
let account;

beforeEach(() => {
    account = new BankAccount(10000);
})
```

Scoping

- By default, tests are grouped according to the file they're in – each file is considered a separate "test suite".
- We can more finely group tests if desired, using the `describe()` function.
- Tests within a group can have their own setup / teardown functions

```
beforeAll(() => {...})
beforeEach(() => {...})

it('outer test 1', () => {...})

describe('stuff', () => {

    beforeAll(() => {...})
    beforeEach(() => {...})

    it('inner test 1', () => {...})
    it('inner test 2', () => {...})

    afterEach(() => {...})
    afterAll(() => {...})

});

it('outer test 2', () => {...})

afterEach(() => {...})
afterAll(() => {...})
```

Run once at the start of this file

Run before each of the four tests in this file

Run once at the start of this block

Run before each of the two tests in this block

Run after each of the two tests in this block

Run once at the end of this block

Run after each of the four tests in this file

Run once at the end of this file

Scoping

- By default, tests are grouped according to the file they're in – each file is considered a separate "test suite".
- We can more finely group tests if desired, using the `describe()` function.
- Tests within a group can have their own setup / teardown functions

```
beforeAll(() => {...})  1.
beforeEach(() => {...}) 2.  6.  11.  17.

it('outer test 1', () => {...}) 3.

describe('stuff', () => {

    beforeAll(() => {...}) 5.
    beforeEach(() => {...}) 7.  12.

    it('inner test 1', () => {...}) 8.
    it('inner test 2', () => {...}) 13.

    afterEach(() => {...}) 9.  14.
    afterAll(() => {...}) 16.

});

it('outer test 2', () => {...}) 18.

afterEach(() => {...}) 4.  10.  15.  19.
afterAll(() => {...})  20.
```

Testing exceptional cases

- To test that code properly throws an exception, we can use the `expect(…).toThrow(…)` matcher

```
it('depositing not-a-number throws an error', () => {
    expect(() => account.deposit('Hello')).toThrow('not an object or a number');
});
```

- Alternatively, we can use `fail()` along with `try` / `catch` if we want to check for consistency after an error has been thrown

```
it('depositing not-a-number throws an error and doesn\'t change balance', () => {
    try {
        account.deposit('Hello');
        fail('Depositing a string should not succeed');
    }
    catch (err) {
        expect(err).toBe('not an object or a number');
        expectAmount(10000, 100, 0, '$100.00');
    }
})
```

Focusing and disabling tests

- Renaming a particular `test()`, `it()`, or `describe()` function to **`x`**`test()`, **`x`**`it()`, or **`x`**`describe()`, will cause those tests / groups to be ignored
- Renaming a particular `test()`, `it()`, or `describe()` to **`f`**`test()`, **`f`**`it()`, or **`f`**`describe()` will cause those tests / groups to become focused
  - If there is at least one focused test / group in a file, then *only* focused tests / groups in that file will run

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

Testing async code

Testing async code

- Sometimes, we might have *asynchronous code* we wish to test
  - Async code is either declared `async` or returns a `Promise`
- Example: Suppose we have an alternative `sum()` function we're testing, that's been declared `async` because it spends time adding numbers as accurately as possible… 😮

```
async function addReallyAccurately(a, b) {
    …
}
```

Testing async code

- Sometimes, we might have *asynchronous code* we wish to test
  - Async code is either declared `async` or returns a `Promise`
- Example: Suppose we have an alternative `sum()` function we're testing, that's been declared `async` because it spends time adding numbers as accurately as possible… 😮
- The following will **not** work as indented:

```
it('adds 1 + 2 = 3', () => {
    expect(addReallyAccurately(1, 2)).toBe(3);
})
```


Because it is `async`, `addReallyAccurately()` won't return 3 – it will return a Promise. A promise is definitely not equal to 3!

Testing async code

- Sometimes, we might have *asynchronous code* we wish to test
  - Async code is either declared `async` or returns a `Promise`
- Example: Suppose we have an alternative `sum()` function we're testing, that's been declared `async` because it spends time adding numbers as accurately as possible… 😮
- The following will **also not** work as indented:

```
it('adds 1 + 2 = 3', () => {
    addReallyAccurately(1, 2)
      .then(result => expect(result).toBe(3));
})
```


The call to expect() won't happen until after the test has already been completed and registered as "passed". Therefore, we'll never know if it failed.

Testing async code

- If we declare our test function to be `async`, then we can `await` `async` operations inside that function. JEST handles this case perfectly.

```
it('adds 1 + 2 = 3', async () => {
    expect(await addReallyAccurately(1, 2)).toBe(3);
})
```

- Alternatively, we can have our test function return a promise. JEST will wait for that promise to resolve / reject.

```
it('adds one thing = that thing', () => {
    return addReallyAccurately(42).then(result => expect(result).toBe(42));
})
```

Testing async code

- Another option: We can use JEST's `.resolves` / `.rejects` options. This allows us to specifically test whether promises reject when they're supposed to

```
it('ignores non-numeric values when adding', () => {
    return expect(addReallyAccurately(3, null, 4, undefined, [], 2, 'Hello', false))
        .resolves.toBe(9);
})


it('properly rejects this promise', () => {
    return expect(asyncFailsauce()).rejects.toBe('failblog.org');
})
```

Testing async code

- Our final option is to give our test function an argument called **done**. This is a function that we can call when we want the test to complete.
- If we do this, JEST will wait for `done()` to be called
  - If an `expect()` fails before `done()` is called, the test will fail as normal
  - If `done()` is called with no arguments, the test will pass
  - If `done()` is called with an argument, the test will fail and pass that argument on to the test runner
  - If `done()` is not called within a certain timeout, the test will fail
- This allows us:
  - More fine-grained control if required
  - Testing of old-style code which uses callbacks rather than promises or `async` / `await`

Testing async code


```
it('adds nothing = 0', done => {
    addReallyAccurately()
        .then(result => {
            expect(result).toBe(0);
            done();
        })
        .catch(err => {
            done(err);
         });
})
```

- No one method of testing async code is considered best practice
- Use whichever suits your code style / makes your tests look neater / more readable
  - Feel free to mix & match, even within the same file

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

Mocking

Mocking

- Sometimes, our code under tests might call other code. We might want to:
  - Ensure that other code is used as intended
  - Control the return values that other code supplies to our code under test
- We can do this using **mocking**. We *mock* the other code that is used by our code under test
- JEST natively supports mocking both functions and classes

Mocking functions

- Say we have a function we want to test, that's supposed to call a given callback a certain number of times.
- We can use `jest.fn()` to create a function which takes any number of arguments
  - By default it will return `undefined`
- We can investigate:
  - How many times this function has been called
  - What arguments this function was called with

Mocking functions

- **Code under test:**

```
function forEach(array, callback) {
    for (let i = 0; i < array.length; i++) {
        callback(array[i], i);
    }
}
```

Should call the callback once for each element in the array

- **Test case:**

```
it('forEach appropriately calls the callback', () => {

    const mockCallback = jest.fn();

    forEach(['a', 'b', 'c'], mockCallback);

    expect(mockCallback.mock.calls.length).toBe(3);
    expect(mockCallback.mock.calls[0][0]).toBe('a');
    expect(mockCallback.mock.calls[0][1]).toBe(0);
})
```

Create our mock using `jest.fn()`

Invoke the code under test, supplying the mock as the callback

Ensure that our mock was called 3 times, and that the first time it was called the args were ‘a’ and 0.

Mocking functions

- We can supply our own function to `jest.fn()`
  - The mock will "call through" to our supplied function
  - We can use this to specify what value should be returned
- Example: testing a coin flipper which depends on a supplied random source to produce either 'Heads' or 'Tails'

```
function coinFlip(randomSource) {
    if (randomSource() >= 0.5) {
        return 'Heads';
    }
    else {
        return 'Tails';
    }
}
```

```
it('…', () => {
    const mockRandom = jest.fn(() => 0.75);

    expect(coinFlip(mockRandom)).toBe('Heads');
    expect(mockRandom.mock.calls.length).toBe(1);
})
```

This "random" source will always return 0.75, and we can check how many times it has been called.

Mocking object functions

- Sometimes, the functions we want to mock aren't supplied as arguments as in the previous two examples, but are globals.
- Example: Suppose the `coinFlip()` function from the previous slide used `Math.random()` rather than a supplied `randomSource`
- We can spy on calls to such functions – and alter their return values if required – using `jest.spyOn()`

Mocking object functions


```
beforeEach(() => {
    jest.spyOn(Math, 'random');
})

afterEach(() => {
    Math.random.mockRestore();
})

it('coinFlip2 returns heads whenever Math.random() returns > 0.5', () => {

    Math.random.mockReturnValue(0.75);

    expect(coinFlip2()).toBe('Heads');
    expect(Math.random.mock.calls.length).toBe(1);
})
```

Enable spying on `Math.random`. By default this won't change its behaviour, but will allow us to check how many times it has been called / what arguments it has been supplied with

Restore `Math.random` to normal. Good practice to do this in `afterEach()` so it will run even if tests fail.

This will modify our mocked `Math.random` to always return 0.75

Mocking classes

- Other times, we might wish to test that our code creates instances of various classes and uses them as intended
  - E.g. we might want to test that the following function creates and uses a `SoundPlayer` as intended:

```
function beep() {

    const player = new SoundPlayer({ volume: 42, repeat: false });
    player.playSoundFile('./beep.mp3');

}
```

- We can do this using `jest.mock()`
- Example on next slide; for more info see here.

Mocking classes


```
import SoundPlayer from '../sound-player';
jest.mock('../sound-player');

beforeEach(() => {
    SoundPlayer.mockClear();
});

it('beep uses a SoundPlayer properly', () => {

    beep();

    expect(SoundPlayer).toHaveBeenCalledTimes(1);
    expect(SoundPlayer.mock.calls[0][0].volume).toBe(42);
    expect(SoundPlayer.mock.calls[0][0].repeat).toBe(false);

    const mockSoundPlayer = SoundPlayer.mock.instances[0];
    const mockPlayFile = mockSoundPlayer.playSoundFile;
    expect(mockPlayFile).toHaveBeenCalledTimes(1);
    expect(mockPlayFile).toHaveBeenCalledWith('./beep.mp3');
})
```

Enable mocking of the `SoundPlayer` class

Clear the `SoundPlayer` mock before each test. This ensures that any modifications one test makes to this won't bleed into other tests

Check that `SoundPlayer`'s constructor was called once, with the expected arguments

Check that the `SoundPlayer` object's `playSoundFile` method was called once with the expected arguments

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

Mocking axios with axios-mock-adapter

Mocking axios calls

- We might want to mock axios calls:
  - To check whether various REST APIs are being consumed as expected
  - To check our code's reactions to various HTTP responses are as intended (both success and fail cases)
- We could manually set this up using `jest.spyOn()`
  - However there's a lot we'd have to consider when simulating proper axios promises!
- Let's use the **axios-mock-adapter** package instead:

```
npm install --save-dev axios-mock-adapter
```

Mocking axios calls

Function under test:

- Should make a GET request to `https://trex-sandwich.com/ajax/articles?id=:id`, where **`:id`** is the supplied `articleId`
- Should return the article retrieved from the endpoint

```
async function getArticle(articleId) {
    …
}
```

Mocking axios calls

Part 1: Setup

- Takes place outside of the unit test itself

```
import axios from 'axios';
import MockAdapter from 'axios-mock-adapter';

const axiosMock = new MockAdapter(axios);
```

Import `axios` itself, and the `MockAdapter` class. Create an instance of `MockAdapter` which we'll use later in our test file.

```
afterEach(() => {
    axiosMock.reset();
});
```

After each test, reset the `MockAdapter` instance so that we clean up any leftover mess in case tests fail, etc.

Mocking axios calls

Part 2: Mock an axios call

- Do this in the unit test, before the line where axios is expected to be used by the code under test

```
const dummyArticle = {
    id: 2,
    title: 'The title',
    content: 'The content'
}
```

Define dummy data to return from our fake API call

```
axiosMock
    .onGet('https://trex-sandwich.com/ajax/articles?id=2')
    .reply(200, dummyArticle);
```

The next time our axios mock receives a GET request to the given URL…

Respond with status 200 (OK), and the given dummy article.

Mocking axios calls

Part 3: Checking the result

- Do this part after we expect axios to have been used by our code under test

```
const article = await getArticle(2);
```

Invoke the code which should use axios

```
expect(axiosMock.history.get[0].url)
    .toEqual('https://trex-sandwich.com/ajax/articles?id=2');
```

Check that axios.get was called once with the given URL

```
expect(article).toEqual(dummyArticle);
```

Check the article returned was the same one which was returned by axios

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

Testing MongoDB / mongoose

Testing MongoDB / mongoose

- To test code which interacts with MongoDB, we can use the mongodb-memory-server package

```
npm install --save-dev mongodb-memory-server
```

- This allows us to create a fast, in-memory full-featured MongoDB database which we can connect to using mongoose

```
beforeAll(async () => {

    mongod = await MongoMemoryServer.create();

    const connectionString = mongod.getUri();
    await mongoose.connect(connectionString);

});
```

Before any tests run, start the MongoDB in-memory database and connect to it using mongoose

```
afterAll(async () => {
    await mongoose.disconnect();
    await mongod.stop();
});
```

After all tests are complete, disconnect and stop the in-memory database.

Testing MongoDB / mongoose

- Before each test we can populate the database with test data if required…

```
beforeEach(async () => {
    const coll = await mongoose.connection.db.createCollection('breakfasts');

    breakfast1 = …

    await coll.insertMany([breakfast1, breakfast2, breakfast3]);
});
```

- And clear the database after each test..

```
afterEach(async () => {
    await mongoose.connection.db.dropCollection('breakfasts');
});
```

Testing MongoDB / mongoose

- We now have access to a fully functional test database without having to spin up a separate program

```
it('gets a single breakfast', async () => {
    const breakfast = await Breakfast.findById(breakfast2._id);
    expect(breakfast.eggs).toBe(12);
    expect(breakfast.bacon).toBe(2);
    expect(breakfast.drink).toBeUndefined();
});


it('adds a breakfast without crashing', async () => {
    const breakfast = new Breakfast({ eggs: 9, bacon: 2, drink: 'Tea' });
    await breakfast.save();

    const fromDb = await mongoose.connection.db.collection('breakfasts')
        .findOne({ _id: breakfast._id });
    expect(fromDb).toBeTruthy();
    …
});
```

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

Testing Express APIs

Testing Express APIs

- Let's say we have some Express API routes we want to test. There are a couple of ways we could do this:
  - Mock the `express()` / `Router()` etc. functions. Challenging but there is library support
  - Spin up a real Express server during testing, and call it. Works well, but have to appropriately manage starting / stopping the server, and using correct ports.
  - Use the **supertest** package, which handles ports and server startup / shutdown for us.

Supertest

Installation:

```
npm install --save-dev supertest
```

Setup:

- At the beginning of your API tests file, import necessary packages, create an `express()` `app`, add any routes you want to test.
- Make sure you add `express.json()` if you want to test routes that need to receive JSON data in the request body!

```
import routes from …;
import express from 'express';
import request from 'supertest';

const app = express();
app.use(express.json());
app.use('/', routes);
```

Supertest

Usage:

```
it('gets a single breakfast from the server', (done) => {

    request(app)
        .get('/breakfasts/000000000000000000000002')
        .send()
        .expect(200)
        .end((err, res) => {

            if (err) return done(err);

            const breakfast = res.body;
            expect(breakfast.eggs).toBe(12);
            expect(breakfast.bacon).toBe(2);
            expect(breakfast.drink).toBeUndefined();

            return done();
        });
});
```

Often, we use this Jest syntax, which provides us with the `done()` function, which will be called when the test is complete.

Send a `GET` request to the given path. We don't need to supply hostname or port. If we have data to send in the request body, we add it as an argument to the `send()` function.

Expect a 200 response.

If all `expect()`s pass, then this callback will run. We can look at any errors / responses returned from the Express API, and write tests based on them. Make sure to call `done()` when finished.

Supertest

Usage:

- If we don't need any custom testing in the `end()` function, we can supply our `done` callback directly to the final `expect()`.

```
it('returns a 404 response when requesting with an invalid id', (done) => {

    request(app)
        .get('/breakfasts/00000000000000000000000F')
        .send()
        .expect(404, done);

});
```

- Comprehensive documentation on Supertest website.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

Testing React

Testing React

- We can test various React components in isolation
- Different kinds of testing available:
  - React testing library
  - Shallow testing
  - Snapshot testing
- We will focus on the use of React testing library for this course, as it allows for the most comprehensive tests and is widely used in industry.

Testing Vite+React apps

- Packages for testing Vite+React apps
  - **Vitest** – Functionally and syntactically identical to Jest, but much easier integration with Vite apps
  - **Jsdom** – Simulates a browser DOM in memory. Used behind the scenes, we don’t directly interact with this
  - **React testing library** – allows us to “mock” rendering React components, check the rendered HTML, and simulate actions such as button clicks & text input

Testing Vite+React apps

- Install the following *dev* dependencies using npm:
  - `@testing-library/jest-dom`
  - `@testing-library/react`
  - `@testing-library/user-event`
  - `jsdom`
  - `vitest`

Testing Vite+React apps

- Modify vite.config.js as follows:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react-swc'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom'
  }
})
```

Configures Vite to render to jsdom rather than the real browser DOM when running in test mode.

- At the top of every test file, import all the JEST functions you want to use in that file, from **vitest**, e.g:

```
import { describe, expect, it } from 'vitest';
```

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module Five

React testing library

React testing library

- Tests based on DOM nodes rather than React components
- Intended to "write tests that mimic the way *users* will interact with your page" (testing-library.com)
- A simple example: A component which displays a button allowing the user to load a greeting from a URL when it is clicked.
- The library is very comprehensive

React testing library

- Imports:

```
import '@testing-library/jest-dom';
```

Enables useful “expect” syntax such as `expect(…).toBeInTheDocument()`

```
import { render, fireEvent, waitFor } from '@testing-library/react';
```

Allows us to render React components in a test environment, obtain query functions to investigate rendered DOM elements, and simulate user interactions.

React testing library

- Firstly, we import the `render()` function from the package. We use this to render any React components we would like to test
- This returns an object with several functions we can use to query the elements rendered "onscreen". These fit into one of several categories:

Function	Description
`queryBy…`	Returns the matching element, or null if no matches. Usually used for verifying that something does *not* appear onscreen.
`getBy…`	Returns the matching element, or fails the test if no matches. Usually used for verifying that something appears onscreen.
`findBy…`	Returns a promise that can be used to wait until a matching element appears onscreen. Useful for verifying that certain components appear after an async operation (e.g. data appearing onscreen after being retrieved from an API call).

React testing library

- For each kind of function introduced on the previous slide, there are several options for how we search for elements onscreen.
- Some of the most commonly used are:

Function	Description
`…ByText()`	Finds components whose text content matches the given string or Regex.
`…ByLabelText()`	Finds components (usually form components) whose `<label>` text matches the given string or Regex.
`…ByRole()`	Finds components matching the given role (e.g. `button`, `img`).

React testing library

- Other useful imports:
  - A function called `waitFor()`, which accepts a callback as an argument and will continually poll the callback until it returns a value or a timeout expires (default 1000ms).
  - An object called `fireEvent`, which contains methods to simulate user input such as typing text or clicking elements on a page.
  - An extra package called user-event, which provides a more comprehensive event simulation mechanism if required.

React testing library – examples


```
it('renders correctly when all details are supplied', () => {

    const { getByText } = render(
        <BusinessCard name="Bob" phNum="021 123 4567" email="bob@bob.com" />
    );

    // Ensure the person's name appears correctly.
    expect(getByText(/Name/i)).toBeInTheDocument();
    expect(getByText('Bob')).toBeInTheDocument();

    // Ensure the person's phone number appears correctly.
    expect(getByText(/Phone number/i)).toBeInTheDocument();
    expect(getByText('021 123 4567')).toBeInTheDocument();

    // Ensure the person's email appears correctly.
    expect(getByText(/Email/i)).toBeInTheDocument();
    expect(getByText('bob@bob.com')).toBeInTheDocument();
});
```

Render a React component

Ensure components with the given text appear in the document

React testing library – examples


```
import { MyContext, … } from "../component-with-context";


it('renders myGreeting from context correctly', () => {

    const context = {
        myGreeting: 'React is cool!'
    }

    const { getByText } = render(
        <MyContext.Provider value={context}>
            <MyComponentWithContext />
        </MyContext.Provider>
    )

    expect(getByText('React is cool!')).toBeInTheDocument();
})
```

To test a component which uses values from context, we can surround it with our own Provider, supplying whatever test data we like.

React testing library – examples


```
it('renders homepage correctly', () => {
    const { getByText, queryByText } = render(
        <MemoryRouter initialEntries={['/home']}>
            <ComponentWithRoutes />
        </MemoryRouter>
    )

    expect(getByText('App title')).toBeInTheDocument();

    expect(getByText('Homepage')).toBeInTheDocument();

    expect(queryByText('About me!')).not.toBeInTheDocument();
})
```

Testing anything using React Router component (e.g. `Link`, `Routes`, …), we need to surround the code under test with a Router. `MemoryRouter` works well and allows us to specify the URL and history "back stack" using the `initialEntries` prop.

React testing library – examples


```
it('loads a greeting correctly', async () => {

    const { queryByText, findByText, getByRole } = render(<GreetingLoader url="http://test.com" />);
    expect(queryByText('Hello, world!')).not.toBeInTheDocument();
    expect(queryByText('Loading...')).not.toBeInTheDocument();

    const data = {
        greeting: 'Hello, world!'
    }
    axiosMock.onGet('http://test.com').reply(200, data);
```

If our React components are expected to call APIs using `axios`, we can easily combine the react testing library with `axios-mock-adapter`.

```
    const button = getByRole('button');
    fireEvent.click(button);
```

Simulating a button click using `fireEvent`.

```
    expect(queryByText('Loading...')).toBeInTheDocument();

    expect(axiosMock.history.get[0].url).toEqual('http://test.com');

    const helloWorld = await findByText('Hello, world!');
```

Waits for up to 1000ms for the matching component to appear within the document.

```
    expect(helloWorld).toBeInTheDocument();
    expect(queryByText('Loading...')).not.toBeInTheDocument();

});
```

References

- All frameworks & libraries covered in these slides are far more comprehensive than we can cover in one lecture:
- All have excellent documentation, which is a great starting point for further reading:
  - JEST
  - axios-mock-adapter
  - In-memory MongoDB
  - supertest
  - React testing library
  - Vitest