19. Testing of Full Stack Application

Aim/Objective: The objective of this experiment is to be familiar with different testing tools for React frontend and Express-backend

Description:

The aim of testing a full stack application is to ensure its quality, reliability and Functionality across all layers and components of the application. By testing the full stack application to identify and resolve issues at different layers of the application, deliver a reliable and high-quality user experience, and minimize the risk of failures or performance issues. It helps build confidence in the applications functionality, stability, and security.

Some common approaches and tools for testing React frontend apps are :

Unit Testing, Integration Testing, Snapshot Testing, End-to-End (E2E) Testing, State Management Testing, Mocking and Stubbing, Continuous Integration (CI) etc.

For testing React components **'jest-dom'** is a library extends the functionality of the Jest testing framework. It provides additional matchers and utilities to make it easier to write expressive and readable tests for React components.

cross-env is a popular npm package that allows you to set environment variables in a cross-platform manner, making it useful for testing Express and Node.js backends.

Pre-Requisites:

Node js, Web Browsers, express, Postman, nodemon.

>**npm install --save-dev @testing-library/jest-dom**

>npm install --save-dev cross-env

Pre-Lab:

1. What is the importance of testing a React frontend app?

   Testing a React frontend app is crucial to ensure its reliability, functionality, and user experience, while also reducing maintenance costs and enhancing developer confidence.

2. Explain the difference between unit testing and integration testing in the context of React components.

   Unit testing focuses on isolated testing of individual React components or functions to ensure they work correctly, while integration testing checks how these components work together to validate interactions and data flow within the application.

3. How does snapshot testing work in React, and what are its benefits?

   Snapshot testing in React captures and compares the rendered output of a component with a previously stored "snapshot." It ensures UI consistency, helps detect unintended changes, and simplifies visual regression testing.

4. What are the different types of tests you can perform on an Express backend?
   You can perform unit tests to check individual components, integration tests to validate interactions, end-to-end tests for complete user flows, and security, performance, and regression tests on an Express backend.

5. b. Explain the purpose of unit testing and integration testing in the context of an Express application.
   Unit testing in Express focuses on testing isolated components like middleware or route handlers for correctness. Integration testing validates how these components work together, ensuring the application functions correctly as a whole.

In-Lab:

Exercise 1: Write a react code for testing frontend using jest environment. Observe the test cases.

Exercise 2: How testing is performed for Express-backend?

Procedure/Program:

INLAB

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <p data-testid="count">{count}</p>
      <button data-testid="increment" onClick={increment}>Increment</button>
      <button data-testid="decrement" onClick={decrement}>Decrement</button>
    </div>
  );
}

export default Counter;
```

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

test('renders the counter component', () => {
  const { getByTestId } = render(<Counter />);
  const countElement = getByTestId('count');
  const incrementButton = getByTestId('increment');
  const decrementButton = getByTestId('decrement');

  expect(countElement).toBeInTheDocument();
  expect(countElement).toHaveTextContent('0');

  fireEvent.click(incrementButton);
  expect(countElement).toHaveTextContent('1');

  fireEvent.click(decrementButton);
  expect(countElement).toHaveTextContent('0');
});


import React, { useState } from 'react';

function Button() {
  const [clicked, setClicked] = useState(false);

  const handleClick = () => {
    setClicked(!clicked);
  };

  return (
    <button onClick={handleClick} data-testid="button">
      {clicked ? 'Clicked' : 'Click me'}
    </button>
  );
}

export default Button;

import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import Button from './Button';

test('renders a button with the initial label', () => {
  const { getByTestId } = render(<Button />);
  const buttonElement = getByTestId('button');

  expect(buttonElement).toBeInTheDocument();
  expect(buttonElement).toHaveTextContent('Click me');
});

test('changes the label when the button is clicked', () => {
  const { getByTestId } = render(<Button />);
  const buttonElement = getByTestId('button');

  fireEvent.click(buttonElement);
  expect(buttonElement).toHaveTextContent('Clicked');

  fireEvent.click(buttonElement);
  expect(buttonElement).toHaveTextContent('Click me');
});
```

INLAB2

```
import express from 'express';
```

```
const app = express();

app.get('/users', function(req, res) {
  res.json({ users: 'allUsers' });

  // Real code from my application below
  //  model.User.findAll().then (users => {
  //       res.status(200).json({ users });
  //    }).catch(error=>{
  //       console.log(error)
  //       req.status(500).send(error)
  //  })
});

app.get('/users/3', function(req, res) {
  res.json({ user: 'user3' });

  // Real code from my application below
  // const { id } = req.params;
  //    model.User.findOne({
  //       where: { id: Number(id) }
  //    }).then(user=>{
  //       res.status(200).json({ user });
  //    }).catch(error=>{
  //       console.log(error)
  //       req.status(500).send(error)
  //    })
});

export const server = app;
```

```
// package.json
 "scripts": {
    ...
    "test": "cross-env NODE_ENV=test jest --testTimeout=10000",
    "pretest": "cross-env NODE_ENV=test npm run migrate:reset",
    "migrate:reset": "npx sequelize-cli db:migrate:undo:all && npm run
migrate",
```

```
  "migrate": "npx sequelize-cli db:migrate && npx sequelize-cli
db:seed:all",
 }
```

```
...
 "jest": {
   "testEnvironment": "node",
   "coveragePathIgnorePatterns": [
     "/node_modules/"
   ]
 },
...
```

```
//test.js

const server = require('../index.js');
const supertest = require('supertest');
const requestWithSupertest = supertest(server);
```

```
describe('User Endpoints', () => {

 it('GET /user should show all users', async () => {
   const res = await requestWithSupertest.get('/users');
    expect(res.status).toEqual(200);
    expect(res.type).toEqual(expect.stringContaining('json'));
    expect(res.body).toHaveProperty('users')
 });

});
```

```
it('GET /user/:id should show a user', async () => {
      const res = await requestWithSupertest.get('/users/3')
      expect(res.statusCode).toEqual(200)
      expect(res.body).toHaveProperty('user3')
  });
```

```
npm test
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

- **Data and Results:**

```
FAIL src/utils..test.js
  • utils › formatUserName does not add @ when it is already provided

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

    Expected: "@jc"
    Received: "@@jc"

       7 |
       8 |   test('formatUserName does not add @ when it is already provided', () => {
    >  9 |     expect(formatUserName('@jc')).toBe('@jc');
         |                                   ^
      10 |   });
      11 | });
      12 |

      at Object.<anonymous> (src/utils..test.js:9:35)

Test Suites: 1 failed, 1 skipped, 1 of 2 total
Tests:       1 failed, 1 skipped, 1 passed, 3 total
Snapshots:   0 total
Time:        1.55 s, estimated 2 s
Ran all test suites with tests matching "utils".
```

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (express-backend)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\Syamlal\Techomoro_Sample\express-backend\package.json:

{
  "name": "express-backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes
```

Sample VIVA-VOCE Questions (In-Lab):

1. **What is the essence of javascript in front-end validation of the application**

JavaScript in front-end validation enhances user experience by checking and validating user inputs in real-time, reducing the need for server requests and providing instant feedback to users, ensuring data integrity and a smoother interaction with web applications.

2. **What are the possible anomaly's of database?**

   Data anomalies can include insertion anomalies (incomplete data during record creation), update anomalies (inconsistencies when modifying data), and deletion anomalies (unintended data loss).

3. **How you authenticate the user access?**

   User access can be authenticated using a combination of methods, including username and password authentication, two-factor authentication (2FA), biometric authentication, and OAuth for third-party authorization. Strong authentication mechanisms help secure user access to applications and data.

4. **Why use unit testing ?**

Unit testing ensures that individual components of a software application work correctly in isolation, helping detect and fix issues early in the development process, improving code quality, and making it easier to maintain and refactor code.

5. **What are testing tools available mention any two?**

Two commonly used testing tools are:

Selenium: A widely-used framework for automating web browsers for end-to-end testing.
Jest: A JavaScript testing framework commonly used for unit and integration testing in Node.js and React applications.

Evaluator Remark (if Any):	**Marks Secured:_____out of 50**
	Signature of the Evaluator with Date