BTI425/WEB422 - Web Programming for Apps and Services

Lecture Recap:

Week 10 – Introduction to Testing

Agenda

- Unit Testing
- End to End (e2e) Testing

Unit Testing

- Types of software testing
  - Unit testing, End to End (E2E) testing, performance testing, integration testing, functional testing, acceptance testing
  - Automated vs Manual tests
- Software development practices such as Continuous Integration
  - rely heavily on testing to help ensure that bugs are not introduced when merging / integrating code from multiple developers.
- Unit Tests - testing classes, components or modules, are
  - typically automated tests
  - used to finds problems early in the development cycle
    - including both bugs and flaws/missing parts in the units (properties, methods, UIs)
  - very low level and close to the source of an application

Jest Introduction

- Unit testing our Next.js code using "Jest" testing framework
  - Jest, a delightful JavaScript Testing Framework, works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!
- Getting started
  - Install Jest as a "development dependency":

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

  - Create a new 'test' script in the 'scripts' section of the package.json file:

    ```
    "scripts": {
        …
        "test": "jest --watchAll"
    }
    ```

    - So, to run the test, use the command in terminal: `>npm run test`

  - Create a folder named "tests" in project base folder and create a file called "**practice.test.js**" in the "test" folder

Writing Tests using Jest

- How to create a block of tests (test group)?

  `describe('some tests', () => { … … });` // Optional

  - 1st param: string, name
  - 2nd param: callback function

- How to define a test?

  `test('test name', () => { … … });` // alias: `it('test name', () => { … … });`

  - 1st param: string, description
  - 2nd param: call back function
  - 3rd Param: timeout // optional
  - a test can be created inside or outside a test block

Writing Tests using Jest

- How to create an Expectation?
  - Use an expect() function chained with matcher function:

```
let x = 5;
expect(x).toBe(5);
```

  - expect() function, accepts a param called actual value
  - matcher function, e.g. toBe(), usually accepts a param called expected value
  - a test can contain 1 or more expectations

```
let sum = (num1, num2) => num1 + num2;
describe('Practice Tests', () => {
    test('sum function adds 1 + 2 to equal 3', () => {
        expect(sum(1, 2)).toBe(3);
    });
});
```

- Modifier: .not
  - ".not" lets you test the opposite, e.g. `expect(sum(1, 2)).not.toBe(5);`

Introduction to "Matchers"

- Jest uses matcher functions ("matchers") to define a complete "expectation" for a value.
  - Matcher examples: "toBe()", "toHaveReturned()", "toBeCloseTo()", etc.

A list of the most common matchers:

- Truthiness
  - toBeNull matches only null
  - toBeUndefined matches only undefined
  - toBeDefined is the opposite of toBeUndefined
  - toBeTruthy matches anything that an if statement treats as true
  - toBeFalsy matches anything that an if statement treats as false

Note:

- You should use the matcher that most precisely corresponds to what you want your code to be doing.
- Truthy: the values they aren't falsy which includes values - false, 0, "", null, undefined, NaN.

Introduction to "Matchers"

- Numbers
  - toBeGreaterThan()
  - toBeGreaterThanOrEqual()
  - toBeLessThan()
  - toBeLessThanOrEqual()
  - toBe()
  - toEqual()
  - toBeCloseTo() // For floating point equality
- Strings
  - toMatch()
- Arrays and iterables
  - toContain() // it also available for Strings
- Exceptions
  - toThrow()

Testing Components and Pages

- Install a few additional dependencies, ie:
  - jest-environment-jsdom
  - @testing-library/react
  - @testing-library/jest-dom

```
npm install --save-dev jest-environment-jsdom @testing-library/react @testing-library/jest-dom
```

- Create file jest.config.js in project's base folder
  - to configure the testing environment

Sample tests:

- Test 1: "Vercel" Link in the first child of the "main" element
  - To ensure/test that a link to "https://vercel.com" is rendered within the first child of the "main" element (section).
  - Test file: /tests/**index.test.js**

Note: API container: the single <div> element within the page body for rendering the page

Testing Components and Pages

- Test 2: Component with User Event(s)
  - re-create our familiar "ClickCounter" component
  - write a test to ensure that when the user clicks the button, the counter increases
  - Install external "companion" library for Testing Library: "user-event"

    ```
    npm install --save-dev @testing-library/user-event
    ```
  - Test file: /tests/clickCounter.test.js
- Test 3: API Route with Route Parameter
  - create a new file: pages/vehicles/[id].js and a test file: /tests/vehicles.test.js
  - install a 3rd -party module to help make (mock) requests to web API :

    ```
    npm install --save-dev node-mocks-http
    ```
    - module "get" functions: res._getData(), res._getStatusCode()
  - write tests to ensure
    - when the user makes "get" request with a valid route parameter value, status code 200 returned and return object with id value matching route parameter
    - when the user make a request with a invalid route parameter value, status code 404 as no object is returned

E2E (End to End) Testing

- **End-to-End Testing: What is it?**
  - End-to-end testing is a technique that tests the entire software product from beginning to end to ensure the application flow behaves as expected. It defines the product's system dependencies and ensures all integrated pieces work together as expected.
  - The main purpose of End-to-end (E2E) testing is to test from the end user's experience by simulating the real user scenario and validating the system under test and its components for integration and data integrity.
- The first testing tool recommended in the Next.js documentation is "Cypress"
  - Essentially, we will be using Cypress to test how multiple pieces of the application work together to enable the user to perform a series of tasks (ie: logging in, performing an action with multiple steps, logging out, etc.).

Installing / Configuring Cypress

- We will be writing some tests on the "Iron Session" example code.
- Install Cypress:

```
npm install --save-dev cypress
```

- Add the following entry to **"scripts"** in **package.json**

```
"cypress": "cypress open"
```

- Execute the command to run e2e test:

```
npm run cypress
```

  - It will add the files at the first time:
    - cypress.config.js
    - cypress/fixtures/example.json
    - cypress/support/e2e.js
    - Click "Start E2E…", Select "Create new empty SPEC": cypress/e2e/spec.cy.js file
      - Click the test file in the pop-up browser window to get the test result

Note: Spec is short for "Specification" - refer to the technical details of a given feature or application which must be fulfilled, or simply a test.

Testing the "iron-session" example

- Make an important configuration change for our application:
  - adding a baseUrl to **cypress.config.js,** then you can use the shortened cy.visit('index.html') for cy.visit('http://localhost:3000/index.html')
- Cypress Test Syntax - similar to Jest's syntax
  - **describe(name, fn):** Creates a block that groups several related tests together:
  - **it(name, fn)** - This is the function that defines a test, identified by "name"
  - **cy.visit()** - Visit (navigate to) a remote URL
  - **cy.url()** - Get the current URL of the page that is currently active.
  - **cy.should()** - Create an assertion. Assertions are automatically retried until they pass or time out.
  - **cy.get()** - Get one or more DOM elements by selector or alias
  - **cy.contains()** - Get the DOM element containing the text. DOM elements can contain more than the desired text and still match.
  - **cy.click()** - Click a DOM element.
  - **cy.type()** - Type into a DOM element. Curly braces ({}) may be used to type a key such as "enter", "esc", "backspace", etc.

Testing the "iron-session" example

Creating tests/specs using the cypress/e2e/spec.cy.js file

- Test 1 Protected Route /profile-sg
- Test 2 Rejecting Invalid Github Users
- Test 3 Granting Access to Valid Github Users

Note: You need to run the app "**npm run dev**" in another terminal before starting the e2e test using "**npm run cypress**"

Running in "Headless" Mode

- To run the tests strictly from the command prompt (ie: "Headlessly") rather than using the GUI tool
- Add a "scripts" entry in **package.json**:

```
"cypress:headless": "cypress run"
```

- To start testing, we can run:

```
npm run cypress:headless
```

The End