El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

CICLO IV:

Desarrollo de Aplicaciones Web

MisionTIC2022

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

Sesión 15:

Desarrollo de Aplicaciones Web

Testing

Mision TIC 2022

Objetivos de la sesión

Al finalizar esta sesión estarás en capacidad de:

1. Crear pruebas unitarias de una aplicación web.
2. Crear pruebas de integración de una aplicación web.

Pruebas de Software

- Es posible que cuando estemos trabajando en nuestro proyecto y hagamos cambios perdamos alguna funcionalidad o introducimos algún bug en las mismas.
- Por este motivo, se introducen el concepto de pruebas de software.
- Principalmente hay varios tipos de pruebas entre los que destacan:
  - Pruebas unitarias.
  - Pruebas de integración.
  - Entre otros.

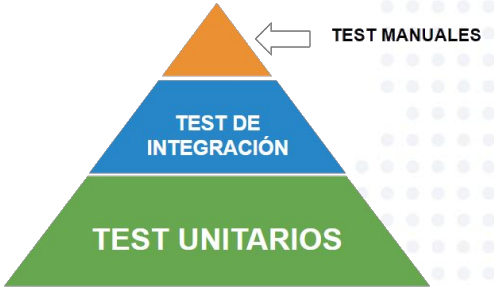


Imagen tomada de [netmentor]

Pruebas Unitarias

- Este tipo de pruebas constan en dividir funcionalidades en unidades pequeñas..
- Para cada una de estas unidades definimos pruebas de software que nuestro programa de forma interna verificará.
- Generalmente son implementadas por desarrolladores de software.
- Nos permite verificar el correcto funcionamiento de cada una de nuestras funcionalidades.
- Para un proyecto definimos pruebas unitarias que garanticen su correcto funcionamiento y nos el riesgo de introducir bugs durante el desarrollo.

Pruebas Unitarias - Node

- Para hablar sobre pruebas unitarias en Node.js, primero consideremos el siguiente proyecto.
- Una calculadora que contenga las 4 operaciones básicas:
  - Suma
  - Resta
  - Multiplicación
  - División
- Este proyecto será un servicio REST, hecho en Express.

Pruebas Unitarias - Node

- Para iniciar clonamos nuestra plantilla de node:
- Instalamos los siguientes paquetes:
  - express.
  - compression.
  - body-parser.
- De igual forma instalamos los siguientes paquetes a modo de desarrollador:
  - jest.
  - supertest.
  - eslint-plugin-jest.

Pruebas Unitarias - Node

- Configuramos nuestro archivo de configuración para eslint:
  - habilitamos “jest/globals”: true en “env”:

```
"env": {
  "node": true,
  "es2021": true,
  "jest/globals": true
},
```

  - agregamos “jest” a nuestro listado de plugins:

```
"plugins": ["prettier", "jest"],
```

Pruebas Unitarias - Node

- Configuramos nuestro archivo de configuración para eslint:
  - Agregamos la regla “node/no-unpublished-import” para habilitar la importación de supertest:

```
"rules": {
  // ...
  "node/no-unpublished-import": [
    "error",
    {
      "allowModules": ["supertest"]
    }
  ]
}
```

- Por último agregamos el script test en nuestro “package.json”:

```
"test": "jest",
```

Pruebas Unitarias - Node

- Definimos el controlador para nuestra calculadora en “src/controllers/calculator.controller.js”:

```
import { Router } from 'express';
const router = Router();
```

- Definimos el método o recurso /add:

```
router.post('/add', async (req, res) => {
  const { a, b } = req.body;
  res.json({ operation: 'Suma', result: a + b });
});
```

- Definimos el método o recurso /sub:

```
router.post('/sub', async (req, res) => {
  const { a, b } = req.body;
  res.json({ operation: 'Resta', result: a - b });
});
```

Pruebas Unitarias - Node

- Definimos el método o recurso /mul:

```
router.post('/mul', async (req, res) => {
  const { a, b } = req.body;
  res.json({ operation: 'Multiplicación', result: a * b });
});
```

- Definimos el método o recurso /div:

```
router.post('/div', async (req, res) => {
  const { a, b } = req.body;
  const operation = 'División'
  if (b === 0) res.status(400).json(
          { operation, error: 'No se permite división por cero' }
      );
  else res.json({ operation, result: a / b });
});
```

Pruebas Unitarias - Node

- Definimos nuestro metodo de configuracion de controladores en “src/controllers/index.js”:

```
import CalculatorRouter from './calculator/calculator.controller';

export const setUpControllers = (app) => {
  app.use('/calc', CalculatorRouter);
};
```

Pruebas Unitarias - Node

- Definimos nuestra aplicación por aparte en **“src/app.js”**:

```
import express from 'express';
import compression from 'compression';
import { json, urlencoded } from 'body-parser';
import { setUpControllers } from './controllers';

const app = express();

app.use(compression());
app.use(urlencoded({ extended: false }));
app.use(json());

setUpControllers(app);

export default app;
```

- Esto con el fin de poder acceder a ella mediante supertest.

Pruebas Unitarias - Node

- Instanciamos nuestra aplicación en **“src/index.js”**:

```
import app from './app';

const main = async () => {
  try {
    const PORT = process.env.PORT || 3000;
    app.listen(PORT, () => console.log(`Servidor esperando por peticiones
en localhost:${PORT}`));
  } catch (error) {
    console.error(error);
  }
};

main();
```

Pruebas Unitarias - Node

- Hasta ahora no hemos definido ninguna clase de unit testing.
- Para definir nuestro primer unit testing crearemos el archivo **“src/controllers/calculator.test.js”**.
- En este podremos acceder de forma libre a todos los métodos de paquete jest ya que este paquete es el que se encargará de ejecutar las pruebas por nosotros.
- El identificara nuestras pruebas con aquellos archivos que terminen en **“.test.js”** o en **“.spec.js”**.
- Dentro de los métodos de jest los más importantes son:
  - describe.
  - test.
  - expect.

Pruebas Unitarias - Node

- Asi mismo describiremos sub pruebas dentro de **`/calc`**:

```
describe('/calc', () => {
  const controller = 'calc';
  describe('/add', () => {
    const resource = 'add';
    const endpoint = `/${controller}/${resource}`;
  });
  describe('/sub', () => {
    const resource = 'sub';
    const endpoint = `/${controller}/${resource}`;
  });
  describe('/mul', () => {
    const resource = 'mul';
    const endpoint = `/${controller}/${resource}`;
  });
  describe('/div', () => {
    const resource = 'div';
    const endpoint = `/${controller}/${resource}`;
  });
});
```

Pruebas Unitarias - Node

- De igual forma antes de nuestra prueba principal **`/calc`**, definiremos las siguientes variables:

```
const OPERATIONS = {
  ADD: 'Suma',
  SUB: 'Resta',
  MUL: 'Multiplicación',
  DIV: 'División',
};

const TEST = {
  ADD: { a: 1, b: 2, result: 3 },
  SUB: { a: 2, b: 1, result: 1 },
  MUL: { a: 2, b: 5, result: 10},
  DIV: { a: 6, b: 2, result: 3 },
  DIV_ZERO: { a: 10, b: 0 }
};

const ACCEPT_JSON = ['Accept', 'application/json'];
```

- Esto para poder facilitar nuestras pruebas.

Pruebas Unitarias - Node

- Importamos nuestra aplicación y a supertest de esta forma al principio de nuestro archivo:

```
import request from 'supertest';
import app from '../../app';
```

- Ahora empezaremos con la primera prueba en la subprueba /add:

```
  describe('/add', () => {
    const resource = 'add';
    const endpoint = `/${controller}/${resource}`;
    test('Código de respuesta 200', async () => {
      const { status } = await
request(app).post(endpoint).set(...ACCEPT_JSON).send(TEST.ADD);
      expect(status).toBe(200);
    });
  });
```

- Definimos una evaluación con test, le pasamos un nombre y un callback.
- Con supertest consumimos endpoints de nuestra app.

Pruebas Unitarias - Node

- Así mismo, podemos definir pruebas más complejas que solo analizar el status code de nuestra respuesta:

```
test(`'operation' debería de ser '${OPERATIONS.ADD}'`, async () => {
  const { status, body } = await request(app).post(endpoint).set(...ACCEPT_JSON).send(TEST.ADD);
  expect(status).toBe(200);
  expect(typeof body.operation).toBe('string');
  expect(body.operation).toBe(OPERATIONS.ADD);
});

test(`'result' debería de ser = ${TEST.ADD.a} + ${TEST.ADD.b} = ${TEST.ADD.result}`, async () => {
  const { status, body } = await request(app).post(endpoint).set(...ACCEPT_JSON) .send(TEST.ADD);
  expect(status).toBe(200);
  expect(typeof body.operation).toBe('string');
  expect(body.operation).toBe(OPERATIONS.ADD);
  expect(typeof body.result).toBe('number');
  expect(body.result).toBe(TEST.ADD.result);
});
```

- En estos casos evaluamos el cuerpo de la respuesta.

Pruebas Unitarias - Node

- Replicaremos la prueba donde evaluamos el resultado en el resto de métodos de la siguiente forma:
  - /sub:

```
test(`'result' debería de ser = ${TEST.SUB.a} - ${TEST.SUB.b} =
${TEST.SUB.result}`, async () => {
  const { status, body } = await
request(app).post(endpoint).set(...ACCEPT_JSON).send(TEST.SUB);
  expect(status).toBe(200);
  expect(typeof body.operation).toBe('string');
  expect(body.operation).toBe(OPERATIONS.SUB);
  expect(typeof body.result).toBe('number');
  expect(body.result).toBe(TEST.SUB.result);
});
```

Pruebas Unitarias - Node

- Replicaremos la prueba donde evaluamos el resultado en el resto de métodos de la siguiente forma:
  - /mul:

```
test(`'result' debería de ser = ${TEST.MUL.a} * ${TEST.MUL.b} =
${TEST.MUL.result}`, async () => {
  const { status, body } = await
request(app).post(endpoint).set(...ACCEPT_JSON).send(TEST.MUL);
  expect(status).toBe(200);
  expect(typeof body.operation).toBe('string');
  expect(body.operation).toBe(OPERATIONS.MUL);
  expect(typeof body.result).toBe('number');
  expect(body.result).toBe(TEST.MUL.result);
});
```

Pruebas Unitarias - Node

- Replicaremos la prueba donde evaluamos el resultado en el resto de métodos de la siguiente forma:
  - /div:

```
test(`'result' debería de ser = ${TEST.DIV.a} / ${TEST.DIV.b} =
${TEST.DIV.result}`, async () => {
  const { status, body } = await
request(app).post(endpoint).set(...ACCEPT_JSON).send(TEST.DIV);
  expect(status).toBe(200);
  expect(typeof body.operation).toBe('string');
  expect(body.operation).toBe(OPERATIONS.DIV);
  expect(typeof body.result).toBe('number');
  expect(body.result).toBe(TEST.DIV.result);
});
```

Pruebas Unitarias - Node

- Finalmente ejecutamos npm test para realizar todas nuestras pruebas:

```
PASS src/controllers/calculator/calculator.test.js
  /calc
    /add
      √ Código de respuesta 200 (50 ms)
      √ 'operation' debería de ser 'Suma' (6 ms)
      √ 'result' debería de ser = 1 + 2 = 3 (4 ms)
    /sub
      √ Código de respuesta 200 (4 ms)
      √ 'operation' debería de ser 'Resta' (3 ms)
      √ 'result' debería de ser = 2 - 1 = 1 (4 ms)
    /mul
      √ Código de respuesta 200 (4 ms)
      √ 'operation' debería de ser 'Multiplicación' (4 ms)
      √ 'result' debería de ser = 2 * 5 = 10 (4 ms)
    /div
      √ Código de respuesta 200 (5 ms)
      √ 'operation' debería de ser 'División' (4 ms)
      √ 'result' debería de ser = 6 / 2 = 3 (4 ms)
      √ No debería permitir divion por 0 (5 ms)

Test Suites: 1 passed, 1 total
Tests:       13 passed, 13 total
Snapshots:   0 total
Time:        1.426 s, estimated 13 s
Ran all test suites.
```

- Como podemos observar se nos indican las pruebas, las subpruebas y los exámenes ejecutados.
- La duración de estos.
- Y las pruebas realizadas.

Pruebas Unitarias - Node

- Así mismo, si le agregamos el flag --coverage a jest, este nos genera un reporte adicional:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
src	100	100	100	100	
app.js	100	100	100	100	
src/controllers	100	100	100	100	
index.js	100	100	100	100	
src/controllers/calculator	100	100	100	100	
calculator.controller.js	100	100	100	100	

- Esto nos es de utilidad para medir el % de código que ha sido probado por nuestras pruebas.
- Para más información sobre Jest ver este enlace.
- Esta template se encuentra en su repositorio de GitHub.

Pruebas - React

- Para implementar pruebas en React seguimos prácticamente el mismo flujo de trabajo que con node.
- A diferencia de node nosotros trabajamos con la librería **`@testing-library/react`**.
- De igual forma nombraremos nuestras pruebas unitarias con .test.js como extensión.
- Y modificaremos nuestro script de test de la siguiente forma:

```
"test": "react-scripts test --verbose",
```

Pruebas - React

- A continuación desarrollaremos una aplicación de prueba que consuma nuestra API de calculadora.
- Primero instalaremos bootstrap de la siguiente forma:

```
npm i --save bootstrap @popperjs/core
```

- Luego en **src/index.js** agregaremos los siguientes imports:

```
import 'bootstrap';
import 'bootstrap/dist/css/bootstrap.min.css';
```

Pruebas - React

- Luego modificaremos **src/App.js** de la siguiente forma:

```
import React from 'react';
import Calculator from './components/Calculator';

function App() {
  return (
    <div className='container p-4'>
      <div className='row'>
        <div className='col-12'>
          <div className='row'>  <h1>Misi&oacute;n Tic - Calculadora</h1>
</div>
          <Calculator />
        </div>
      </div>
    </div>
  );
}
```

Pruebas - React

- Luego declararemos el componente Calculator en **src/components/Calculator**, lógica:

```
const [a, setA] = useState('0');
const [b, setB] = useState('0');
const [operation, setOperation] = useState(OPERATION.LIST[0].id);
const { loading, error, payload, fetch: fetchOperation } = useOperation();

const isSubmitDisbaled = operation === OPERATION.LIST[0].id;
const resultExists = typeof payload?.result === 'number';
const hasError = Boolean(error);

const onAChangeHandler = event => setA(event.target.value);
const onBChangeHandler = event => setB(event.target.value);
const formSubmitHandler = event => {
  event.preventDefault();
  fetchOperation(+a, +b, OPERATION.RESOURCE[operation]);
};
```

Pruebas - React

- Luego declararemos el componente Calculator en **src/components/Calculator**, formulario:

```
const renderForm = () => (
  <form className='input-group mb-3' onSubmit={formSubmitHandler}>
    <Dropdown options={OPERATION.LIST} onSelect={setOperation}> {OPERATION.MAP[operation]} </Dropdown>
    <input type='number' className='form-control text-center' aria-label='Text input with dropdown button'
      placeholder='Inserte el valor de a' value={a} onChange={onAChangeHandler} />
    <span data-testid='calculator-operation' className='input-group-text d-inline-flex
justify-content-center'style={{ width: 40 }}>
      {OPERATION.SYMBOL[operation]}
    </span>
    <input type='number' className='form-control text-center' aria-label='Text input with dropdown button'
      placeholder='Inserte el valor de b' value={b} onChange={onBChangeHandler} />
    <Button data-testid='calculator-submit' type='submit' disabled={isSubmitDisbaled || loading}>
      Calcular
    </Button>
  </form>
);
```

Pruebas - React

- Luego declararemos el componente Calculator en **src/components/Calculator**, render:

```
return (
  <>
    <div className='row py-3'>{renderForm()}</div>
    {resultExists && (
      <p className='text-success' data-testid='calculator-result'>
        Resultado: {payload.result}
      </p>
    )}
    {hasError && (
      <p className='text-danger' data-testid='calculator-result'>
        {error.message}
      </p>
    )}
  </>
);
```

Pruebas - React

- Luego declararemos el componente Dropdown en **src/components/Dropdown**:

```
import React from 'react';
import Button from '../Button';
import DropdownMenu from './DropdownMenu';

const Dropdown = ({ options, onSelect, children }) => (
  <>
    <Button isDropdown>{children}</Button>
    <DropdownMenu options={options} onSelect={onSelect} />
  </>
);

export default Dropdown;
```

Pruebas - React

- Luego declararemos el componente Dropdown en **src/components/Dropdown**:

```
import React from 'react';
import Button from '../Button';
import DropdownMenu from './DropdownMenu';

const Dropdown = ({ options, onSelect, children }) => (
  <>
    <Button isDropdown>{children}</Button>
    <DropdownMenu options={options} onSelect={onSelect} />
  </>
);

export default Dropdown;
```

Pruebas - React

- Luego declararemos el componente DropdownMenu en **src/components/DropdownMenu**:

```
import React from 'react';
import DropdownItem from './DropdownItem';

const DropdownMenu = ({ options, onSelect }) => {
  const renderOptions = () =>
    options.map(({ id, value }) => (
      <DropdownItem key={`option-${id}`} onClick={() => onSelect(id)}>
        {value}
      </DropdownItem>
    ));

  return <ul className='dropdown-menu'>{renderOptions()}</ul>;
};

export default DropdownMenu;
```

Pruebas - React

- Luego declararemos el componente DropdownItem en **src/components/DropdownItem**:

```
import React from 'react';

const DropdownItem = ({ onClick, children }) => (
  <li onClick={onClick}>
    <div className='dropdown-item'>{children}</div>
  </li>
);

export default DropdownItem;
```

Pruebas - React

- Luego declararemos el componente Button en **src/components/Button**:

```
import React from 'react';

const Button = ({ onClick = () => {}, isDropdown = false, children, ...rest }) => {
  const extraClasses = isDropdown && 'dropdown-toggle';
  const extraProps = isDropdown && {'data-bs-toggle': 'dropdown', 'aria-expanded':
'false' };
  const dataTestId = isDropdown ? 'dropdown-button' : 'button';

  return (
    <button className={`btn btn-outline-secondary ${extraClasses}`} type='button'
onClick={onClick}
      data-testid={dataTestId} {...extraProps} {...rest}>
      {children}
    </button>
  );
};

export default Button;
```

Pruebas - React

- Luego declararemos el hook personalizado useOperation en **src/hooks/useOperation**:

```
import { useCallback, useState } from 'react';
import { CalculatorController } from '../api';

export const useOperation = () => {
  const [payload, setPayload] = useState({});
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const fetch = useCallback(async (a, b, operation) => {
    try {
      setLoading(true); setError(null);
      const res = await CalculatorController.fetchOperation({ a, b, operation });
      setLoading(false); setPayload(res);
    } catch (error) {
      setError(error); setPayload(null);
    } finally { setLoading(false); }
  }, []);
  return { loading, error, payload, fetch };
};
```

Pruebas - React

- De igual forma consultamos la api de la siguiente forma en **src/api/calculator.controller.js**:

```
import CONFIGURATION from './config.json';
const HEADERS = { Accept: 'application/json', 'Content-Type': 'application/json' };
const getURL = (operation = null) => {
  if (!Boolean(operation)) throw new Error('resource not valid');
  return new URL(`${CONFIGURATION.PROTOCOL}://${CONFIGURATION.HOST}/calc/${operation}`);
};
export const fetchOperation = async ({ a, b, operation }) => {
  try {
    const resource = getURL(operation);
    const body = JSON.stringify({ a, b });
    const response = await fetch(resource, { method: 'POST', headers: HEADERS, body });
    const asJson = await response.json();
    if (!response.ok) throw new Error(asJson.error);
    return asJson;
  } catch (error) {
    throw error;
  }
};
```

Pruebas - React

- Por último, tenemos dos archivos de configuración:
  - `src/api/config.json`:

```
{
  "PROTOCOL": "http",
  "HOST": "localhost:8000"
}
```

Pruebas - React

- Por último, tenemos dos archivos de configuración:
  - **src/utilities/constants.js**:

```
const OPERATION_LIST = [
  { id: 'NO_OP', value: 'Seleccione su opción...' },
  { id: 'ADD', value: 'Sumar' },
  { id: 'SUB', value: 'Restar' },
  { id: 'MUL', value: 'Multiplicar' },
  { id: 'DIV', value: 'Dividir' },
];

const OPERATION_MAP = {
  NO_OP: 'Seleccione su opción...',
  ADD: 'Sumar',
  SUB: 'Restar',
  MUL: 'Multiplicar',
  DIV: 'Dividir',
};
```

Pruebas - React

- Por último, tenemos dos archivos de configuración:
  - `src/utilities/constants.js`:

```
const OPERATION_SYMBOLS = {
  NO_OP: '',
  ADD: ' + ',
  SUB: ' - ',
  MUL: ' • ',
  DIV: ' / ',
};

const OPERATION_RESOURCE = {
  NO_OP: '',
  ADD: 'add',
  SUB: 'sub',
  MUL: 'mul',
  DIV: 'div',
};
```

Pruebas - React

- Por último, tenemos dos archivos de configuración:
  - `src/utilities/constants.js`:

```
export const DIVISION_BY_ZERO_NOT_ALLOWED = 'No se permite división por cero';

export const OPERATION = {
  LIST: OPERATION_LIST,
  MAP: OPERATION_MAP,
  SYMBOL: OPERATION_SYMBOLS,
  RESOURCE: OPERATION_RESOURCE,
};
```

Pruebas Unitarias - React

- Corregimos la prueba por defecto en **src/App.test.js**:

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('Tiene título', () => {
  render(<App />);
  const title = screen.getByText(/Misión Tic - Calculadora/i);
  expect(title).toBeInTheDocument();
});
```

- Como podemos ver describimos una prueba unitaria para verificar que el componente app efectivamente renderize el título como esperamos que lo haga.

Pruebas Unitarias - React

- Definimos nuestras pruebas en **`src/components/Calculator/Calculator.test.js`**:
  - La estructura inicial se vera asi:

```
describe('Pruebas de calculadora', () => {
  describe('Pruebas unitarias', () => {
    /* ... */
  });

  describe('Pruebas de integración', () => {
    /* ... */
  });
});
```

Pruebas Unitarias - React

- Definimos nuestras pruebas en **`src/components/Calculator/Calculator.test.js`**:
  - Pruebas unitarias:

```
describe('Pruebas unitarias', () => {
  test('Verificar estado inicial', () => {
    /* ... */
  });

  test('Verificar cambio de operación', async () => {
    /* ... */
  });
});
```

Pruebas Unitarias - React

- Definimos nuestras pruebas en **src/components/Calculator/Calculator.test.js**:
  - Prueba #01:

```
test('Verificar estado inicial', () => {
  const { getByTestId, getByPlaceholderText } = render( <Calculator /> );

  const span = getByTestId('calculator-operation');
  const button = getByTestId('calculator-submit');
  const aInput = getByPlaceholderText('Inserte el valor de a');
  const bInput = getByPlaceholderText('Inserte el valor de b');
  const dropdown = getByTestId('dropdown-button');

  expect(span.textContent).toBe(OPERATION.SYMBOL.NO_OP);
  expect(aInput).toHaveValue(0);
  expect(bInput).toHaveValue(0);
  expect(button).toBeDisabled();
  expect(dropdown.textContent).toBe(OPERATION.MAP.NO_OP);
});
```

Pruebas Unitarias - React

- Definimos nuestras pruebas en **src/components/Calculator/Calculator.test.js**:
  - Prueba #02:

```
test('Verificar cambio de operación', async () => {
  const { getByTestId, getByText } = render(<Calculator />);

  const dropdown = getByTestId('dropdown-button');
  fireEvent.click(dropdown);

  const sumOption = await waitFor(() => getByText(OPERATION.MAP.ADD));
  fireEvent.click(sumOption);

  const span = getByTestId('calculator-operation');
  const button = getByTestId('calculator-submit');

  expect(span.textContent).toBe(OPERATION.SYMBOL.ADD);
  expect(button).not.toBeDisabled();
});
```

Pruebas Unitarias - React

- Como podemos ver usamos el método render para ‘renderizar’ nuestro componente de React.
- Luego seleccionamos elementos de nuestro componente con métodos que obtenemos desde el render como:
  - **`getByTestId`**, que obtiene el elemento que contenga el data-testid correspondiente.
  - **`getByPlaceholderText`**, que obtiene el elemento que contenga el placeholder correspondiente.
  - **`getByText`**, que obtiene el elemento cuyo innerText sea el correspondiente.
- Adicionalmente, tenemos **`fireEvent`** que nos permite disparar eventos programaticamente.

Pruebas de Integración

- Definimos nuestras pruebas en **`src/components/Calculator/Calculator.test.js`**:
  - Pruebas de integración:

```
describe('Pruebas de integración', () => {
  for (const operation of Object.values(OPERATION.MAP).slice(1)) {
    const operationListElement = OPERATION.LIST.find(el => el.value === operation);
    const symbol = OPERATION.SYMBOL[operationListElement.id].trim();
    test(`Verificar envío de '${operation}' 0 ${symbol} 0`, () => {
      /* ... */
    });
  }

  test(`Verificar envío de 'Dividir' 2/1`, async () => {
    /* ... */
  });
});
```

- Hacemos un ciclo para probar las 4 operaciones posibles.

Pruebas de Integración

- Definimos nuestras pruebas en **src/components/Calculator/Calculator.test.js**:
  - Prueba #01:

```
test(`Verificar envío de '${operation}' 0 ${symbol} 0`, async () => {
  const { getByTestId, getByText } = render(<Calculator />);
  const dropdown = getByTestId('dropdown-button');
  fireEvent.click(dropdown);
  const sumOption = await waitFor(() => getByText(operation));
  fireEvent.click(sumOption);
  const button = getByTestId('calculator-submit');
  fireEvent.click(button);
  const result = await waitFor(() => getByTestId('calculator-result'));
  const toBeContained = operation === OPERATION.MAP.DIV
      ? DIVISION_BY_ZERO_NOT_ALLOWED
      : 'Resultado: 0';
  expect(result.textContent).toContain(toBeContained);
});
```

Pruebas de Integración

- Definimos nuestras pruebas en **src/components/Calculator/Calculator.test.js**:
  - Prueba #02:

```
test(`Verificar envío de 'Dividir' 2/1`, async () => {
  const { getByTestId, getByText, getByPlaceholderText } = render( <Calculator />
);
  const dropdown = getByTestId('dropdown-button');
  fireEvent.click(dropdown);
  const sumOption = await waitFor(() => getByText(OPERATION.MAP.DIV));
  fireEvent.click(sumOption);
  const aInput = getByPlaceholderText('Inserte el valor de a');
  const bInput = getByPlaceholderText('Inserte el valor de b');
  fireEvent.change(aInput, { target: { value: 2 } });
  fireEvent.change(bInput, { target: { value: 1 } });
  const button = getByTestId('calculator-submit');
  fireEvent.click(button);
  const result = await waitFor(() => getByTestId('calculator-result'));
  expect(result.textContent).toContain('Resultado: 2');
});
```

Pruebas de Integración

- Finalmente, hacemos la prueba asíncrona con el keyword **`async`** para poder utilizar el método **`waitFor`**.
- Esto nos permite esperar cambios en el documento HTML ya sea por eventos de nuestros componentes o resultados de peticiones HTTP que se hayan enviado.
- Por lo que nos permite definir nuestras pruebas de forma secuencial.
- Para ver el repositorio acceder al siguiente enlace.
- Para más información ver testing-library.

Pruebas - React

- Por último, ejecutamos el script **`npm run test`**:

```
PASS src/App.test.js
  √ Tiene título (48 ms)

PASS src/components/Calculator/Calculator.test.js
  Pruebas de calculadora
    Pruebas unitarias
      √ Verificar estado inicial (41 ms)
      √ Verificar cambio de operación (22 ms)
    Pruebas de integración
      √ Verificar envío de 'Sumar' 0 + 0 (59 ms)
      √ Verificar envío de 'Restar' 0 - 0 (40 ms)
      √ Verificar envío de 'Multiplicar' 0 • 0 (29 ms)
      √ Verificar envío de 'Dividir' 0 / 0 (31 ms)
      √ Verificar envío de 'Dividir' 2/1 (30 ms)

Test Suites: 2 passed, 2 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        3.586 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

Ejercicios de práctica

Mision TIC 2022

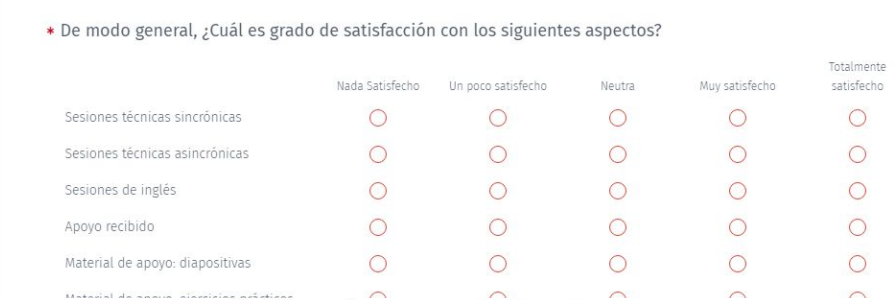
Referencias

- https://medium.com/@testautomation/software-testing-types-the-most-common-types-77789e77c1ad
- https://medium.com/swlh/levels-of-software-testing-b943ce41a2c7
- https://techclub.tajamar.es/unit-testing-en-react/
- https://github.com/benawad/apollo-mocked-provider-example/blob/1_test/src/App.test.tsx
- https://testing-library.com/docs/react-testing-library/intro/

El futuro digital es de todos | MinTIC

UNIVERSIDAD DEL NORTE
Vigilada Mineducación

Seguimiento Habilidades Digitales en Programación



* De modo general, ¿Cuál es grado de satisfacción con los siguientes aspectos?

	Nada Satisfecho	Un poco satisfecho	Neutra	Muy satisfecho	Totalmente satisfecho
Sesiones técnicas sincrónicas	○	○	○	○	○
Sesiones técnicas asincrónicas	○	○	○	○	○
Sesiones de inglés	○	○	○	○	○
Apoyo recibido	○	○	○	○	○
Material de apoyo: diapositivas	○	○	○	○	○

Completa la siguiente encuesta para darnos retroalimentación sobre esta semana ▼▼▼

https://www.questionpro.com/t/ALw8TZlxOJ

MisionTIC2022

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

¡GRACIAS

POR SER PARTE DE ESTA EXPERIENCIA DE APRENDIZAJE!

Mision TIC 2022