DuocUC® INFORMÁTICA Y TELECOMUNICACIONES

Configuración de Jasmine y Karma

- Desarrollo Fullstack II
- DSY1104

FORMAMOS **PERONAS** PARA UNA **SOCIEDAD** MEJOR

01

Configuración de Jasmine y Karma

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Pruebas Unitarias para React en AWS con Karma y Jasmine

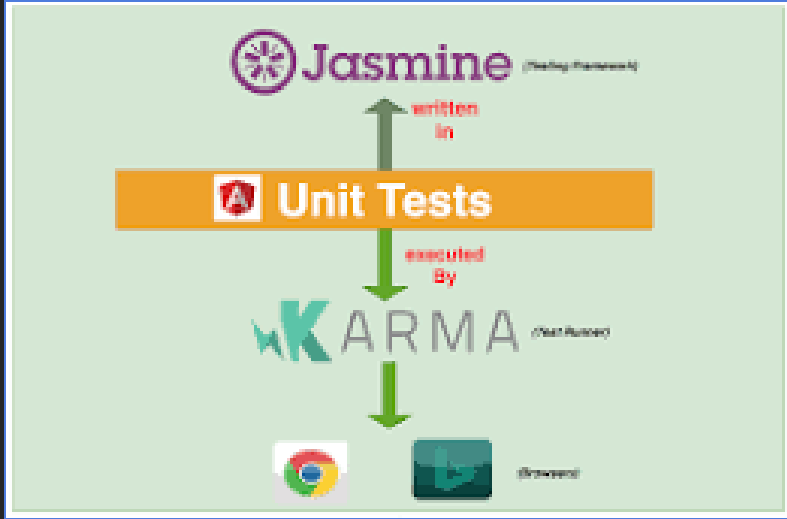
Las pruebas unitarias son fundamentales en el desarrollo de software porque aseguran que cada parte del código funcione correctamente.

Importancia de las Pruebas Unitarias

- Calidad del Código
- Documentación del Código
- Refactorización Segura
- Detección de Regresiones
- Desarrollo Ágil

DuocUC®

CONFIGURACIÓN DE JASMINE Y KARMA



Karma es un corredor de pruebas para JavaScript que permite ejecutar pruebas en múltiples navegadores y dispositivos. Sus características y beneficios incluyen integración con herramientas de desarrollo como Webpack, Gulp y Grunt, ejecución de pruebas en tiempo real, soporte para múltiples navegadores y compatibilidad con sistemas de integración continua como Jenkins, Travis CI y CircleCI.

CONFIGURACIÓN DE JASMINE Y KARMA

Jasmine es un framework de pruebas para JavaScript diseñado para ser fácil de leer y escribir. Sus características incluyen una sintaxis clara y concisa, funciones de espionaje y mocking para pruebas unitarias, la capacidad de ejecutarse en cualquier entorno JavaScript, y la independencia del DOM y de dependencias externas, lo que simplifica su configuración y uso.

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Ejemplo de una prueba unitaria con Jasmin y Karma

```
    // Ejemplo de una prueba unitaria con Jasmine
describe("Calculadora", function() {
  it("debería sumar dos números correctamente", function() {
    var resultado = sumar(2, 3);
    expect(resultado).toBe(5);
  });
});

// Configuración de Karma (karma.conf.js)
module.exports = function(config) {
  config.set({
    frameworks: ['jasmine'],
    files: [
      'src/**/*.js',
      'test/**/*.spec.js'
    ],
    browsers: ['Chrome'],
    singleRun: false,
    preprocessors: {
      'src/**/*.js': ['coverage']
    },
    reporters: ['progress', 'coverage'],
    coverageReporter: {
      type: 'html',
      dir: 'coverage/'
    }
  });
};
```

DuocUC®

CONFIGURACIÓN DE JASMINE Y KARMA

Mocking es una técnica en pruebas unitarias que simula el comportamiento de dependencias externas, permitiendo que las pruebas se concentren en la unidad bajo prueba. Consiste en crear objetos simulados, o "**mocks**", que imitan el comportamiento de objetos reales en un entorno controlado, pudiendo devolver valores específicos o registrar interacciones sin depender de la implementación real de las dependencias.

CONFIGURACIÓN DE JASMINE Y KARMA

Simulación de Dependencias

Es esencial para probar componentes de manera aislada, ya que permite controlar y predecir comportamientos, asegurando pruebas consistentes sin influencias externas. Ejemplos de dependencias que se pueden simular incluyen servicios externos (simulando respuestas de API), bases de datos (simulando consultas y respuestas), y funciones o métodos (simulando métodos de otras clases o módulos para enfocar la prueba en la lógica específica de la unidad).

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Aislamiento de la Unidad Bajo Prueba

El objetivo principal del **mocking** es aislar la unidad bajo prueba de sus dependencias, permitiendo que las pruebas sean más precisas y se centren en la funcionalidad específica que se está verificando.

Beneficios del Aislamiento

Precisión: Las pruebas verifican solo el comportamiento de la unidad, no el de sus dependencias.

Rapidez: Las pruebas son más rápidas al no depender de recursos externos.

Control: Se puede controlar completamente el entorno de prueba, lo que facilita la reproducción de errores y condiciones específicas.

CONFIGURACIÓN DE JASMINE Y KARMA

A continuación, se presenta un ejemplo simple de cómo se puede utilizar mocking en Jasmine

javascript

```javascript
// calculadora.js
function Calculadora(servicioSuma) {
  this.sumar = function(a, b) {
    return servicioSuma.sumar(a, b);
  };
}
```

Código de prueba

javascript

```javascript
// calculadora.spec.js
describe("Calculadora", function() {
  var servicioSumaMock, calculadora;

  beforeEach(function() {
    // Crear un mock del servicio de suma
    servicioSumaMock = {
      sumar: jasmine.createSpy("sumar").and.returnValue(5)
    };
    // Crear una instancia de la calculadora con el mock
    calculadora = new Calculadora(servicioSumaMock);
  });

  it("debería usar el servicio de suma para sumar dos números", function() {
    var resultado = calculadora.sumar(2, 3);
    expect(resultado).toBe(5);
    expect(servicioSumaMock.sumar).toHaveBeenCalledWith(2, 3);
  });
});
```

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Pruebas de Componentes con Estado

Las pruebas de componentes con estado son fundamentales en aplicaciones modernas, especialmente cuando se utilizan frameworks como Angular, React o Vue.js.

Conceptos Básicos

Componentes con Estado: Son componentes que mantienen su propio estado interno y pueden actualizarse dinámicamente en respuesta a eventos o interacciones del usuario.

Pruebas de Componentes: Involucran la verificación de la funcionalidad del componente, incluyendo cómo maneja y modifica su estado.

CONFIGURACIÓN DE JASMINE Y KARMA

Supongamos que tenemos un componente Angular con estado que gestiona una lista de tareas.

```
// tarea.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-tarea',
  template: `
    <h1>Lista de Tareas</h1>
    <ul>
      <li *ngFor="let tarea of tareas">{{ tarea }}</li>
    </ul>
    <input [(ngModel)]="nuevaTarea" placeholder="Agregar tarea"/>
    <button (click)="agregarTarea()">Agregar</button>
  `
})
export class TareaComponent {
  tareas: string[] = [];
  nuevaTarea: string = '';

  agregarTarea() {
    if (this.nuevaTarea.trim()) {
      this.tareas.push(this.nuevaTarea);
      this.nuevaTarea = '';
    }
  }
}
```

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

```typescript
// tarea.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { FormsModule } from '@angular/forms';
import { TareaComponent } from './tarea.component';

describe('TareaComponent', () => {
  let component: TareaComponent;
  let fixture: ComponentFixture<TareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ TareaComponent ],
      imports: [ FormsModule ]
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(TareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('debería crear el componente', () => {
    expect(component).toBeTruthy();
  });

  it('debería agregar una nueva tarea a la lista', () => {
    const tarea = 'Nueva tarea';
    component.nuevaTarea = tarea;
    component.agregarTarea();
    expect(component.tareas.length).toBe(1);
    expect(component.tareas[0]).toBe(tarea);
  });

  it('debería limpiar el campo de entrada después de agregar una tarea', () => {
    component.nuevaTarea = 'Otra tarea';
    component.agregarTarea();
    expect(component.nuevaTarea).toBe('');
  });

  it('no debería agregar una tarea vacía', () => {
    component.nuevaTarea = '';
    component.agregarTarea();
    expect(component.tareas.length).toBe(0);
  });
});
```

Pruebas del Componente

DuocUC®

CONFIGURACIÓN DE JASMINE Y KARMA

Para ejecutar las pruebas con Karma, necesitas una configuración básica en tu proyecto. Aquí hay un ejemplo de archivo de configuración karma.conf.js:

```
module.exports = function(config) {
    config.set({
      basePath: '',
      frameworks: ['jasmine', '@angular-devkit/build-angular'],
      plugins: [
        require('karma-jasmine'),
        require('karma-chrome-launcher'),
        require('karma-jasmine-html-reporter'),
        require('karma-coverage-istanbul-reporter'),
        require('@angular-devkit/build-angular/plugins/karma')
      ],
      client: {
        clearContext: false // leave Jasmine Spec Runner output visible in browser
      },
      coverageIstanbulReporter: {
        dir: require('path').join(__dirname, './coverage/my-app'),
        reports: ['html', 'lcovonly', 'text-summary'],
        fixWebpackSourcePaths: true
      },
      reporters: ['progress', 'kjhtml'],
      port: 9876,
      colors: true,
      logLevel: config.LOG_INFO,
      autoWatch: true,
      browsers: ['Chrome'],
      singleRun: false,
      restartOnFileChange: true
    });
  };
```

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Pruebas de Eventos e Interacciones

Las pruebas de eventos e interacciones son esenciales para verificar que los componentes respondan correctamente a las acciones del usuario, como clics, cambios en formularios y otras interacciones. A continuación, te muestro cómo realizar estas pruebas utilizando Karma y Jasmine en un contexto de Angular, aunque los conceptos pueden aplicarse a otros frameworks de frontend.

CONFIGURACIÓN DE JASMINE Y KARMA

Supongamos que tienes un componente que maneja un evento de clic para aumentar un contador.

```
// Contador.js
import React, { useState } from 'react';

function Contador() {
  const [contador, setContador] = useState(0);

  const aumentar = () => setContador(contador + 1);

  return (
    <div>
      <button onClick={aumentar}>Aumentar</button>
      <p>Contador: {contador}</p>
    </div>
  );
}

export default Contador;
```

CONFIGURACIÓN DE JASMINE Y KARMA

Prueba del componente

```
// Contador.test.js
import { render, screen, fireEvent } from '@testing-library/react';
import Contador from './Contador';

test('debería crear el componente', () => {
  render(<Contador />);
  const boton = screen.getByText(/aumentar/i);
  expect(boton).toBeInTheDocument();
});

test('debería aumentar el contador cuando se hace clic en el botón', () => {
  render(<Contador />);
  const boton = screen.getByText(/aumentar/i);
  fireEvent.click(boton);
  expect(screen.getByText(/contador: 1/i)).toBeInTheDocument();
});

test('debería mantener el estado del contador después de múltiples clics', () => {
  render(<Contador />);
  const boton = screen.getByText(/aumentar/i);
  fireEvent.click(boton);
  fireEvent.click(boton);
  fireEvent.click(boton);
  expect(screen.getByText(/contador: 3/i)).toBeInTheDocument();
});
```

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Pruebas de Hooks Personalizados

Las pruebas de hooks personalizados en React se centran en verificar que los hooks funcionen como se espera en diversas situaciones. Puedes utilizar herramientas como Jasmine y Karma para realizar estas pruebas, aunque en el ecosistema de React es más común usar Jest junto con React Testing Library.

Conceptos claves

Hooks Personalizados: Son funciones que permiten reutilizar lógica de estado y efectos en componentes funcionales de React.

Pruebas de Hooks: Verifican que el hook maneje correctamente el estado, efectos y otros aspectos que se espera que maneje.

CONFIGURACIÓN DE JASMINE Y KARMA

Supongamos que tenemos un hook personalizado que maneja un contador.

```
// useContador.js
import { useState } from 'react';

function useContador(inicial) {
  const [contador, setContador] = useState(inicial);

  const aumentar = () => setContador(contador + 1);
  const reiniciar = () => setContador(inicial);

  return { contador, aumentar, reiniciar };
}

export default useContador;
```

CONFIGURACIÓN DE JASMINE Y KARMA

Componente de Prueba: Para probar el hook, puedes crear un componente simple que lo use y luego renderizar este componente en las pruebas.

```
// TestContador.js
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import useContador from './useContador';

function TestContador() {
  const { contador, aumentar, reiniciar } = useContador(0);

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={aumentar}>Aumentar</button>
      <button onClick={reiniciar}>Reiniciar</button>
    </div>
  );
}

export default TestContador;
```

CONFIGURACIÓN DE JASMINE Y KARMA

Configuración de Karma: Asegúrate de tener Karma configurado para trabajar con React y Jasmine. Ejemplo básico de configuración (***karma.conf.js***):

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', 'browserify'],
    files: [
      'src/**/*.test.js'
    ],
    preprocessors: {
      'src/**/*.test.js': ['browserify']
    },
    browsers: ['Chrome'],
    singleRun: true,
    reporters: ['progress'],
    browserNoActivityTimeout: 60000
  });
};
```

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Pruebas del Hook: Ejemplo para escribir pruebas para el hook ***useContador***.

```
// karma.conf.js
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', 'jest'],
    plugins: [
      require('karma-jasmine'),
      require('karma-jest'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter')
    ],
    files: [
      'src/**/*.test.js'
    ],
    preprocessors: {
      'src/**/*.test.js': ['jest']
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false,
    restartOnFileChange: true
  });
};
```

DuocUC®

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Pruebas de Integración con API

Las pruebas de integración con APIs en aplicaciones React son cruciales para asegurar que los componentes interactúan correctamente con los servicios externos.

Conceptos Clave

Pruebas de Integración: Verifican cómo varios componentes y servicios trabajan juntos en un entorno similar al de producción.

Interacción con API: Incluye la verificación de que las solicitudes y respuestas a las APIs se manejan correctamente y se reflejan en la interfaz de usuario.

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Supongamos que tenemos un componente React que realiza una solicitud a una API para obtener una lista de usuarios.

```jsx
// UsersList.js
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const UsersList = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios.get('https://api.example.com/users')
      .then(response => {
        setUsers(response.data);
        setLoading(false);
      })
      .catch(error => {
        setError(error);
        setLoading(false);
      });
  }, []);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error loading users!</p>;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
};

export default UsersList;
```

DuocUC®

CONFIGURACIÓN DE JASMINE Y KARMA

Para probar el componente que interactúa con una API, puedes usar Jest y React Testing Library. Para simular las respuestas de la API, se puede usar jest.mock para interceptar las llamadas a axios.

```
// UsersList.test.js
import { render, screen, waitFor } from '@testing-library/react';
import axios from 'axios';
import UsersList from './UsersList';

// Mock de la API
jest.mock('axios');

test('debería mostrar una lista de usuarios al recibir datos exitosamente', async () => {
  // Configurar la respuesta simulada de la API
  const usuarios = [{ id: 1, name: 'John Doe' }, { id: 2, name: 'Jane Smith' }];
  axios.get.mockResolvedValue({ data: usuarios });

  render(<UsersList />);

  // Esperar y verificar el contenido
  await waitFor(() => expect(screen.getByText('John Doe')).toBeInTheDocument());
  expect(screen.getByText('Jane Smith')).toBeInTheDocument();
});

test('debería mostrar un mensaje de error si la solicitud falla', async () => {
  // Configurar la respuesta simulada de la API para que falle
  axios.get.mockRejectedValue(new Error('Network Error'));

  render(<UsersList />);

  // Esperar y verificar el mensaje de error
  await waitFor(() => expect(screen.getByText('Error loading users!')).toBeInTheDocument());
});
```

CONFIGURACIÓN DE JASMINE Y KARMA

Configuración de Karma y Jasmine para React- Si prefieres usar Karma y Jasmine para tus pruebas de integración con API, aquí está cómo podrías configurarlo

```
// karma.conf.js
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', 'webpack'],
    files: [
      'src/**/*.test.js'
    ],
    preprocessors: {
      'src/**/*.test.js': ['webpack']
    },
    webpack: {
      // Configuración de Webpack para los tests
      module: {
        rules: [
          {
            test: /\.js$/,
            exclude: /node_modules/,
            use: {
              loader: 'babel-loader',
              options: {
                presets: ['@babel/preset-env', '@babel/preset-react']
              }
            }
          }
        ]
      }
    },
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false,
    restartOnFileChange: true
  });
};
```

Asegúrate de tener webpack y babel-loader instalados para compilar el código de pruebas.

´´´***bash***

npm install --save-dev webpack webpack-cli babel-loader @babel/core @babel/preset-env @babel/preset-react

´´´

DuocUC®

CONFIGURACIÓN DE JASMINE Y KARMA

Las pruebas de rendimiento en aplicaciones React son cruciales para asegurar que la aplicación responde de manera eficiente bajo diversas condiciones. Estas pruebas se centran en medir el tiempo de renderizado, la capacidad de respuesta, el uso de memoria y otros aspectos críticos del rendimiento.

Pruebas de Rendimiento con react-profiler

React Profiler es una API integrada en React que te permite medir el rendimiento de tus componentes.

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Configuración del Componente con Profiler

```jsx
// App.js
import React, { Profiler } from 'react';

const onRenderCallback = (
  id, // ID del Profiler
  phase, // "mount" o "update"
  actualDuration, // Tiempo real en renderizar la actualización
  baseDuration, // Duración base (sin optimizaciones)
  startTime, // Tiempo de inicio del renderizado
  commitTime, // Tiempo de finalización del commit
  interactions // Conjunto de interacciones
) => {
  console.log({
    id,
    phase,
    actualDuration,
    baseDuration,
    startTime,
    commitTime,
    interactions,
  });
};

const App = () => (
  <Profiler id="App" onRenderCallback={onRenderCallback}>
    {/* Tu aplicación */}
    <div>Hola, mundo!</div>
  </Profiler>
);

export default App;
```

CONFIGURACIÓN DE JASMINE Y KARMA

Prueba del Rendimiento

```
// performance.test.js
import React from 'react';
import ReactDOM from 'react-dom';
import { act } from 'react-dom/test-utils';
import App from './App';

describe('Pruebas de rendimiento de React con Profiler', () => {
  let container;

  beforeEach(() => {
    container = document.createElement('div');
    document.body.appendChild(container);
  });

  afterEach(() => {
    document.body.removeChild(container);
    container = null;
  });

  it('mide el tiempo de renderizado', () => {
    act(() => {
      ReactDOM.render(<App />, container);
    });

    // Las mediciones del Profiler se logran automáticamente a través de onRenderCallback
  });
});
```

CONFIGURACIÓN DE JASMINE Y KARMA

Las pruebas de accesibilidad son fundamentales para asegurar que la aplicación React es usable por todas las personas, incluyendo aquellas con discapacidades. Las herramientas y librerías disponibles te permiten automatizar la detección de problemas de accesibilidad y asegurar que tu aplicación cumple con los estándares establecidos.

Herramientas para Pruebas de Accesibilidad

- React Testing Library
- axe-core
- jest-axe
- eslint-plugin-jsx-a11y

CONFIGURACIÓN DE JASMINE Y KARMA

Otras Herramientas de Accesibilidad

Lighthouse: Herramienta de Google para auditar el rendimiento y la accesibilidad de tus aplicaciones web.

Accessibility Insights: Herramienta de Microsoft que ayuda a encontrar y solucionar problemas de accesibilidad.

Ejemplo: Componente React

```jsx
// App.js
import React from 'react';

const App = () => (
  <div>
    <h1>Bienvenido a mi aplicación</h1>
    <button>Haz clic aquí</button>
  </div>
);

export default App;
```

CONFIGURACIÓN DE JASMINE Y KARMA

Prueba de Accesibilidad

```
// accessibility.test.js
import React from 'react';
import { render } from '@testing-library/react';
import { axe, toHaveNoViolations } from 'jest-axe';
import App from './App';

expect.extend(toHaveNoViolations);

test('El componente App no tiene violaciones de accesibilidad', async () => {
  const { container } = render(<App />);
  const results = await axe(container);
  expect(results).toHaveNoViolations();
});
```

CONFIGURACIÓN DE JASMINE Y KARMA

El manejo de errores y los casos límite (edge cases) son fundamentales para asegurar que una aplicación React sea robusta y resistente a fallos.

Manejo de Errores en React

Componentes de Error Boundary: Los "Error Boundaries" son componentes React que atrapan errores en sus componentes hijos durante el renderizado, en métodos del ciclo de vida, y en constructores de todo el árbol por debajo de ellos.

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Ejemplo de Componente Error Boundary

```jsx
import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Actualiza el estado para mostrar la interfaz de usuario alternativa
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // Puedes registrar el error en un servicio de reporte de errores
    console.log("Error:", error, "Info:", errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // Puedes renderizar cualquier interfaz de usuario alternativa
      return <h1>Algo salió mal.</h1>;
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

CONFIGURACIÓN DE JASMINE Y KARMA

Uso del Error Boundary

```javascript
// App.js
import React from 'react';
import ErrorBoundary from './ErrorBoundary';
import MyComponent from './MyComponent';

const App = () => (
  <ErrorBoundary>
    <MyComponent />
  </ErrorBoundary>
);

export default App;
```

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Manejo de Errores en Funciones Asíncronas

Para manejar errores en funciones asíncronas, como solicitudes de red, se utilizan bloques try...catch.

Ejemplo de Manejo de Errores Asíncronos

```jsx
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const DataFetchingComponent = () => {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://api.example.com/data');
        setData(response.data);
      } catch (error) {
        setError(error);
      }
    };

    fetchData();
  }, []);

  if (error) return <p>Error al cargar los datos: {error.message}</p>;
  if (!data) return <p>Cargando...</p>;

  return <div>{/* Renderiza los datos */}</div>;
};

export default DataFetchingComponent;
```

DuocUC®

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Ejemplo de Manejo de Casos Límite

Entrada de Usuario

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

const InputComponent = () => {
  const [input, setInput] = useState('');
  const [error, setError] = useState(null);

  const handleSubmit = () => {
    if (input.trim() === '') {
      setError('El campo no puede estar vacío');
      return;
    }

    // Procesar la entrada
    console.log('Entrada válida:', input);
    setError(null);
  };

  return (
    <div>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
      />
      <button onClick={handleSubmit}>Enviar</button>
      {error && <p style={{ color: 'red' }}>{error}</p>}
    </div>
  );
};

export default InputComponent;
```

DuocUC®

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Datos Nulos o Indefinidos

```jsx
import React from 'react';

const UserProfile = ({ user }) => {
  if (!user) {
    return <p>Usuario no encontrado</p>;
  }

  return (
    <div>
      <h1>{user.name}</h1>
      <p>Email: {user.email}</p>
      {/* Otros campos del usuario */}
    </div>
  );
};

export default UserProfile;
```

CONFIGURACIÓN DE JASMINE Y KARMA

La documentación de pruebas y el uso de comentarios para explicar lógica compleja son esenciales para mantener un código comprensible y mantenible.

Buenas Prácticas de Documentación para Pruebas

Descripciones Claras de las Pruebas: Cada prueba debe tener una descripción clara y concisa que explique qué está probando. Esto ayuda a otros desarrolladores a entender el propósito de la prueba rápidamente.

A continuación, un ejemplo de descripción de Prueba

CONFIGURACIÓN DE JASMINE Y KARMA

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

describe('App Component', () => {
  test('muestra el saludo correcto al hacer clic en el botón', () => {
    render(<App />);
    const button = screen.getByText('Mostrar saludo');
    userEvent.click(button);
    expect(screen.getByText('Hola, mundo!')).toBeInTheDocument();
  });
});
```

Estructura y Organización de las Pruebas: Organiza las pruebas de manera lógica y estructurada utilizando bloques describe y test o it. Agrupa pruebas relacionadas para mejorar la legibilidad.

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

CONFIGURACIÓN DE JASMINE Y KARMA

Ejemplo de Estructura de Pruebas

```javascript
describe('User Login Flow', () => {
  describe('when the login is successful', () => {
    test('redirects to the dashboard', () => {
      // Lógica de la prueba
    });
  });

  describe('when the login fails', () => {
    test('shows an error message', () => {
      // Lógica de la prueba
    });
  });
});
```

CONFIGURACIÓN DE JASMINE Y KARMA

Uso de Datos de Prueba Claros: Utiliza nombres descriptivos para los datos de prueba y asegúrate de que sean fáciles de entender.

Ejemplo de Datos de Prueba

```javascript
const validUser = {
  username: 'user123',
  password: 'password123'
};

const invalidUser = {
  username: 'user123',
  password: 'wrongpassword'
};
```

CONFIGURACIÓN DE JASMINE Y KARMA

Uso de Comentarios para Explicar Lógica Compleja

Comentarios para Explicar Por Qué, No Qué: Los comentarios deben explicar por qué se hace algo en lugar de qué se está haciendo. El código en sí mismo debe ser lo suficientemente claro para explicar el qué.

Ejemplo de Comentarios Explicativos

```
// Inicia un nuevo juego cuando el usuario hace clic en el botón "Nuevo Juego"
const handleNewGameClick = () => {
  // Restablece el estado del juego a su configuración inicial
  resetGameState();
  // Genera un nuevo tablero de juego
  generateNewBoard();
};
```

CONFIGURACIÓN DE JASMINE Y KARMA

Comentarios en Funciones Complejas

En funciones complejas, usa comentarios para dividir la lógica en secciones más manejables y explicativas.

Ejemplo de Comentarios en Funciones Complejas

```
const processUserData = (users) => {
  // Filtra los usuarios inactivos
  const activeUsers = users.filter(user => user.isActive);

  // Ordena los usuarios activos por nombre
  const sortedUsers = activeUsers.sort((a, b) => a.name.localeCompare(b.name));

  // Mapea los usuarios ordenados a un nuevo formato
  const formattedUsers = sortedUsers.map(user => ({
    id: user.id,
    displayName: `${user.firstName} ${user.lastName}`
  }));

  return formattedUsers;
};
```

CONFIGURACIÓN DE JASMINE Y KARMA

Documentación de Funciones Públicas

Para funciones públicas o API, utiliza comentarios JSDoc para proporcionar una documentación más formal y detallada.

Ejemplo de Comentarios JSDoc

javascript

```javascript
/**
 * Agrega dos números.
 *
 * @param {number} a - El primer número.
 * @param {number} b - El segundo número.
 * @returns {number} La suma de los dos números.
 */
function add(a, b) {
  return a + b;
}
```

• Bibliografía

Libros Digitales Biblioteca Duoc. (Con tu cuenta de Duoc puedes consultar)

- Larsson, M. (2023). Microservices with Spring Boot 3 and Spring Cloud: Build resilient and scalable microservices using Spring Cloud, Istio, and Kubernetes (3a ed.). Packt Publishing.

Recursos de información.

- Testing en React.js: Guía Práctica y Herramientas Esenciales (Jest, Testing Library, Cypress): https://www.youtube.com/watch?reload=9&v=bTGil8qPmXo
- Recetas sobre pruebas: https://es.legacy.reactjs.org/docs/testing-recipes.html
- React Testing Library – Tutorial with JavaScript Code Examples: https://www.freecodecamp.org/news/react-testing-library-tutorial-javascript-example-code/

Duoc UC®

CERCANÍA. LIDERAZGO. FUTURO.

duoc.cl

7 AÑOS ACREDITADO

Comisión Nacional de Acreditación CNA-Chile

DESDE AGOSTO 2017 HASTA AGOSTO 2024.
DOCENCIA DE PREGRADO. GESTIÓN INSTITUCIONAL. VINCULACIÓN CON EL MEDIO.