XIOMARA GARCIA

ACTIVIDAD DE PRUEBAS 1

1. Teniendo en cuenta la herramienta tecnológica, investigar: **JASMINE JS**
2. ¿Qué es Jasmine JS?

Jasmine es un framework utilizado para pruebas unitrarias de JS. Es código abierto y permite aplicar pruebas al código JS de nuestros proyectos, y no necesita un DOM para ello, y sigue el procedimiento behavior-driven development (BDD) para asegurarse de que cada línea de JS es debidamente probada.

Jasmine proporciona todo lo necesario para probar código JavaScript:

- **Estructura de pruebas**: Permite organizar las pruebas en suites y specs

- **Assertions**: Ofrece un rico conjunto de matchers para verificar resultados esperados

- **Spies**: Herramientas para simular funciones y verificar llamadas

- **Mocks**: Capacidad para crear objetos simulados

- **Ejecución asíncrona**: Soporte nativo para pruebas de código asíncrono

1. Propietario y diseñador:

* **Propietario**: Pivotal Labs (anteriormente parte de VMware, ahora parte de Broadcom).
* **Diseñado por:**

- **Rajan Agaskar**: Desarrollador principal y creador original

- **Gregg Van Hove**: Co-mantenedor principal

- **Davis W. Frank**: Contribuidor principal y mantenedor

El proyecto fue inicialmente desarrollado por Pivotal Labs y es mantenido activamente por una comunidad de desarrolladores de código abierto.

1. Tipo de licencia

Jasmine utiliza la **licencia MIT**, que es una licencia de código abierto muy permisiva la cual permite:

- Uso comercial y no comercial

- Modificación del código fuente

- Distribución

- Uso privado

- Sublicenciamiento

La única condición es mantener el aviso de copyright y la licencia en las copias del software.

1. Ventajas

**a) Sintaxis Clara y Expresiva:**

- Utiliza un lenguaje natural que hace las pruebas fáciles de leer y entender

- Estructura BDD con `describe()`, `it()`, `expect()`

**b) No Requiere DOM:**

- Puede ejecutarse en cualquier entorno JavaScript

- No depende de navegadores para funcionar

**c) Rica Librería de Matchers:**

- Amplio conjunto de funciones de verificación (`toBe()`, `toEqual()`, `toContain()`, etc.)

- Posibilidad de crear matchers personalizados

**d) Soporte Asíncrono Nativo:**

- Manejo elegante de callbacks, promesas y async/await

- Timeouts configurables para pruebas asíncronas

**e) Spies Integrados:**

- Sistema de espionaje robusto para verificar llamadas a funciones

- Capacidad de simular comportamientos de funciones

**f) Configuración Mínima:**

- Funciona "out of the box" con configuración básica

- Fácil de integrar en proyectos existentes

**g) Documentación Excelente:**

- Documentación completa y ejemplos claros

- Comunidad activa y recursos abundantes

1. Desventajas

**a) Limitaciones en Pruebas de UI:**

- No está diseñado específicamente para pruebas de interfaz de usuario

- Requiere herramientas adicionales para testing de DOM

**b) Curva de Aprendizaje:**

- Los conceptos de spies y mocks pueden ser confusos para principiantes

- Requiere entendimiento de conceptos de testing

**c) Performance:**

- Puede ser más lento que otros frameworks de testing para suites grandes

- El overhead de spies puede impactar performance

**d) Limitado para E2E:**

- No es la mejor opción para pruebas end-to-end

- Necesita complementarse con otras herramientas para testing completo

**e) Dependencias:**

- Aunque es independiente, algunos proyectos pueden requerir configuración adicional

- Integración con algunos build tools puede requerir plugins específicos

1. Categoría de herramienta que es (framework, dobles de prueba, e2e, rendimiento, cobertura)

**Jasmine es un FRAMEWORK DE TESTING UNITARIO** que pertenece a múltiples categorías:

- **Framework de Testing Unitario**: Principalmente diseñado para pruebas unitarias

- **Herramienta BDD (Behavior Driven Development)**: Permite escribir pruebas describiendo comportamientos

- **Framework con Dobles de Prueba**: Incluye sistema de spies y mocks integrado

- **Soporte para Pruebas de Integración**: Puede usarse para pruebas de integración de componentes

1. En qué lenguaje se puede utilizar y/o tipo de software (web, escritorio, móvil)

**Lenguajes:**

- **JavaScript** (lenguaje principal y nativo)

- **TypeScript** (con configuración adecuada)

- **Node.js** (para aplicaciones del lado del servidor)

**Tipos de Software donde se puede utilizar:**

**a) Aplicaciones Web:**

- Single Page Applications (SPA)

- Aplicaciones web progresivas (PWA)

- Sitios web con JavaScript interactivo

**b) Aplicaciones de Escritorio:**

- Electron apps

- Aplicaciones NW.js

- Progressive Desktop Apps

**c) Aplicaciones Móviles:**

- Aplicaciones híbridas (Cordova/PhoneGap)

- React Native apps

- Ionic applications

**d) Backend/Servidor:**

- APIs REST en Node.js

- Servicios web

- Microservicios

**e) Librerías y Frameworks:**

- Angular (Jasmine es el framework por defecto)

- React applications

- Vue.js applications

- Bibliotecas JavaScript independientes

1. ¿Para qué tipos de prueba sirve?

**a) Pruebas Unitarias:**

- Testing de funciones individuales

- Verificación de métodos de clases

- Validación de lógica de negocio aislada

**b) Pruebas de Componentes:**

- Testing de componentes de UI de forma aislada

- Verificación de props y estados en frameworks como Angular/React

**c) Pruebas de Integración:**

- Testing de interacciones entre múltiples componentes

- Verificación de flujos de datos entre módulos

**d) Pruebas de Servicios:**

- Testing de servicios HTTP (con mocking)

- Verificación de APIs internas

- Testing de utilidades y helpers

**e) Pruebas de Comportamiento (BDD):**

- Verificación de comportamientos específicos

- Testing basado en escenarios de usuario

- Pruebas de aceptación automatizadas

**f) Pruebas Asíncronas:**

- Testing de callbacks

- Verificación de promesas

- Testing de async/await

- Pruebas con timeouts

1. ¿En qué sistema operativo sirve?

Jasmine es **multiplataforma** y funciona en Windows, macOS, y Linux.

1. Requisitos de funcionamiento, instalación, y/o configuración.

**REQUISITOS PREVIOS:**

**a) Node.js y npm:**

Verificar instalación

node --version   v14.0.0 o superior recomendado

npm --version   v6.0.0 o superior

**INSTALACIÓN:**

**Opción 1: Global** 🡪 npm install -g jasmine

**Opción 2: Por proyecto** 🡪 npm install --save-dev jasmine

**CONFIGURACIÓN BÁSICA:**

**1. Inicializar Jasmine 🡪** npx jasmine init

**2. Estructura generada:**

proyecto/

├── spec/

│   └── support/

│       └── jasmine.json

└── package.json

**3. Configuración jasmine.json:**

{

  "spec\_dir": "spec",

  "spec\_files": [

    "/\*[sS]pec.js"

  ],

  "helpers": [

    "helpers//\*.js"

  ],

  "stopSpecOnExpectationFailure": false,

  "random": true

}

1. Crear un ejemplo funcional paso a paso el cual debe ser verificado antes de clase y que sirva en Linux.

**PASO 1: Preparación del entorno en Linux**

Actualizar sistema (Ubuntu/Debian)

sudo apt update

sudo apt upgrade -y

Instalar Node.js y npm

curl -fsSL https://deb.nodesource.com/setup\_lts.x | sudo -E bash -

sudo apt-get install -y nodejs

Verificar instalación

node --version

npm --version

**PASO 2: Crear proyecto**

* Crear directorio del proyecto

mkdir jasmine-ejemplo

cd jasmine-ejemplo

* Inicializar proyecto npm: npm init -y
* Instalar Jasmine: npm install --save-dev jasmine
* Inicializar configuración Jasmine: npx jasmine init

**PASO 3: Crear aplicación de ejemplo (src/calculadora.js)**

// Instancia de la calculadora

const calc = new Calculadora();

// Función para mostrar resultados

function mostrarResultado(elementId, resultado, esError = false) {

    const elemento = document.getElementById(elementId);

    elemento.textContent = resultado;

    elemento.className = `result ${esError ? 'error' : 'success'}`;

}

// Operaciones básicas

function calcular(operacion) {

    try {

        const a = parseFloat(document.getElementById('numA').value);

        const b = parseFloat(document.getElementById('numB').value);

        if (isNaN(a) || isNaN(b)) {

            throw new Error('Por favor ingresa números válidos');

        }

        let resultado;

        let simbolo;

        switch(operacion) {

            case 'sumar':

                resultado = calc.sumar(a, b);

                simbolo = '+';

                break;

            case 'restar':

                resultado = calc.restar(a, b);

                simbolo = '-';

                break;

            case 'multiplicar':

                resultado = calc.multiplicar(a, b);

                simbolo = '×';

                break;

            case 'dividir':

                resultado = calc.dividir(a, b);

                simbolo = '÷';

                break;

            default:

                throw new Error('Operación no válida');

        }

        mostrarResultado('resultado-basicas', `${a} ${simbolo} ${b} = ${resultado}`);

    } catch (error) {

        mostrarResultado('resultado-basicas', `❌ Error: ${error.message}`, true);

    }

}

// Verificar par/impar

function verificarParImpar() {

    try {

        const num = parseFloat(document.getElementById('numEspecial').value);

        if (isNaN(num)) {

            throw new Error('Por favor ingresa un número válido');

        }

        const resultado = calc.esParImpar(num);

        const emoji = resultado === 'par' ? '✅' : '🔴';

        mostrarResultado('resultado-especiales', `${emoji} El número ${num} es ${resultado.toUpperCase()}`);

    } catch (error) {

        mostrarResultado('resultado-especiales', `❌ Error: ${error.message}`, true);

    }

}

// Calcular factorial

function calcularFactorial() {

    try {

        const num = parseFloat(document.getElementById('numEspecial').value);

        if (isNaN(num)) {

            throw new Error('Por favor ingresa un número válido');

        }

        const resultado = calc.factorial(num);

        mostrarResultado('resultado-especiales', `${num}! = ${resultado}`);

    } catch (error) {

        mostrarResultado('resultado-especiales', `❌ Error: ${error.message}`, true);

    }

}

// Calcular potencia

function calcularPotencia() {

    try {

        const base = parseFloat(document.getElementById('base').value);

        const exp = parseFloat(document.getElementById('exponente').value);

        if (isNaN(base) || isNaN(exp)) {

            throw new Error('Por favor ingresa números válidos');

        }

        const resultado = calc.potencia(base, exp);

        mostrarResultado('resultado-potencia', `${base}^${exp} = ${resultado}`);

    } catch (error) {

        mostrarResultado('resultado-potencia', `❌ Error: ${error.message}`, true);

    }

}

// Limpiar todos los resultados

function limpiarResultados() {

    const resultados = document.querySelectorAll('.result');

    resultados.forEach(resultado => {

        resultado.textContent = 'Resultado aparecerá aquí...';

        resultado.className = 'result neutral';

    });

}

// Función para limpiar todos los campos

function limpiarCampos() {

    document.getElementById('numA').value = '0';

    document.getElementById('numB').value = '0';

    document.getElementById('numEspecial').value = '5';

    document.getElementById('base').value = '2';

    document.getElementById('exponente').value = '3';

    limpiarResultados();

}

// Eventos para Enter en los inputs

document.addEventListener('DOMContentLoaded', function() {

    const inputs = document.querySelectorAll('input');

    inputs.forEach(input => {

        input.addEventListener('keypress', function(e) {

            if (e.key === 'Enter') {

                // Determinar qué botón presionar basado en el input activo

                if (input.id === 'numA' || input.id === 'numB') {

                    calcular('sumar');

                } else if (input.id === 'numEspecial') {

                    verificarParImpar();

                } else if (input.id === 'base' || input.id === 'exponente') {

                    calcularPotencia();

                }

            }

        });

    });

    // Mensaje de bienvenida en consola

    console.log('Calculadora cargada exitosamente!');

    console.log('Todas las funciones disponibles y probadas');

});

**PASO 4: Crear pruebas (spec/calculadoraSpec.js)**

const Calculadora = require('../src/calculadora');

describe('Calculadora - Pruebas Completas', function() {

    let calculadora;

    beforeEach(function() {

        calculadora = new Calculadora();

    });

    describe('Operaciones Básicas', function() {

        describe('Suma', function() {

            it('debería sumar dos números correctamente', function() {

                expect(calculadora.sumar(2, 3)).toBe(5);

                expect(calculadora.sumar(-10, 15)).toBe(5);

            });

            it('debería sumar números negativos correctamente', function() {

                expect(calculadora.sumar(-1, -1)).toBe(-2);

                expect(calculadora.sumar(-5, 3)).toBe(-2);

            });

            it('debería sumar con cero correctamente', function() {

                expect(calculadora.sumar(0, 0)).toBe(0);

                expect(calculadora.sumar(5, 0)).toBe(5);

                expect(calculadora.sumar(0, -3)).toBe(-3);

            });

            it('debería manejar números decimales correctamente', function() {

                expect(calculadora.sumar(0.1, 0.2)).toBeCloseTo(0.3);

                expect(calculadora.sumar(1.5, 2.5)).toBe(4);

            });

        });

        describe('Resta', function() {

            it('debería restar dos números positivos correctamente', function() {

                expect(calculadora.restar(5, 3)).toBe(2);

                expect(calculadora.restar(10, 10)).toBe(0);

            });

            it('debería manejar restas que resultan en negativos', function() {

                expect(calculadora.restar(3, 5)).toBe(-2);

                expect(calculadora.restar(0, 3)).toBe(-3);

            });

        });

        describe('Multiplicación', function() {

            it('debería multiplicar números positivos correctamente', function() {

                expect(calculadora.multiplicar(3, 4)).toBe(12);

                expect(calculadora.multiplicar(7, 6)).toBe(42);

            });

            it('debería manejar multiplicación por cero', function() {

                expect(calculadora.multiplicar(5, 0)).toBe(0);

                expect(calculadora.multiplicar(0, 8)).toBe(0);

            });

            it('debería manejar números negativos', function() {

                expect(calculadora.multiplicar(-2, 3)).toBe(-6);

                expect(calculadora.multiplicar(-4, -5)).toBe(20);

            });

        });

        describe('División', function() {

            it('debería dividir números correctamente', function() {

                expect(calculadora.dividir(10, 2)).toBe(5);

                expect(calculadora.dividir(15, 3)).toBe(5);

            });

            it('debería manejar divisiones con decimales', function() {

                expect(calculadora.dividir(7, 2)).toBe(3.5);

                expect(calculadora.dividir(1, 3)).toBeCloseTo(0.333, 2);

            });

            it('debería lanzar error al dividir entre cero', function() {

                expect(function() {

                    calculadora.dividir(5, 0);

                }).toThrowError('No se puede dividir entre cero');

            });

        });

    });

    describe('Validaciones de Entrada', function() {

        it('debería lanzar error con parámetros no numéricos en suma', function() {

            expect(function() {

                calculadora.sumar('a', 2);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.sumar(null, 2);

            }).toThrowError('Los parámetros deben ser números');

        });

        it('debería lanzar error con parámetros no numéricos en resta', function() {

            expect(function() {

                calculadora.restar(undefined, 2);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.restar('texto', 5);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.restar(true, false);

            }).toThrowError('Los parámetros deben ser números');

        });

        it('debería lanzar error con parámetros no numéricos en multiplicación', function() {

            expect(function() {

                calculadora.multiplicar([], 2);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.multiplicar('5', 3);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.multiplicar({valor: 10}, 2);

            }).toThrowError('Los parámetros deben ser números');

        });

        it('debería lanzar error con parámetros no numéricos en división', function() {

            expect(function() {

                calculadora.dividir({}, 2);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.dividir(new Date(), 5);

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.dividir(10, 'cero');

            }).toThrowError('Los parámetros deben ser números');

        });

        // Pruebas adicionales de validación más exhaustivas

        it('debería manejar valores extremos correctamente', function() {

            // Números muy grandes en suma

            expect(calculadora.sumar(Number.MAX\_VALUE, 1)).toBe(Number.MAX\_VALUE);

            // Números muy pequeños en suma

            expect(calculadora.sumar(Number.MIN\_VALUE, 0)).toBe(Number.MIN\_VALUE);

            // Infinito en operaciones básicas

            expect(calculadora.sumar(Infinity, 1)).toBe(Infinity);

            expect(calculadora.sumar(-Infinity, 1)).toBe(-Infinity);

            expect(calculadora.restar(Infinity, 1)).toBe(Infinity);

            expect(calculadora.restar(-Infinity, 1)).toBe(-Infinity);

            expect(calculadora.multiplicar(Infinity, 2)).toBe(Infinity);

            expect(calculadora.dividir(Infinity, 2)).toBe(Infinity);

        });

        it('debería manejar NaN correctamente en todas las operaciones', function() {

            // NaN debería producir NaN en operaciones matemáticas

            expect(calculadora.sumar(NaN, 5)).toBeNaN();

            expect(calculadora.restar(NaN, 3)).toBeNaN();

            expect(calculadora.multiplicar(NaN, 3)).toBeNaN();

            expect(calculadora.dividir(NaN, 2)).toBeNaN();

        });

        it('debería validar parámetros faltantes en todas las operaciones', function() {

            // Suma

            expect(function() {

                calculadora.sumar();

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.sumar(5);

            }).toThrowError('Los parámetros deben ser números');

            // Resta

            expect(function() {

                calculadora.restar();

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.restar(5);

            }).toThrowError('Los parámetros deben ser números');

            // Multiplicación

            expect(function() {

                calculadora.multiplicar();

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.multiplicar(5);

            }).toThrowError('Los parámetros deben ser números');

            // División

            expect(function() {

                calculadora.dividir();

            }).toThrowError('Los parámetros deben ser números');

            expect(function() {

                calculadora.dividir(5);

            }).toThrowError('Los parámetros deben ser números');

        });

        it('debería validar tipos de datos específicos', function() {

            // Strings que parecen números

            expect(function() {

                calculadora.sumar('5', 3);

            }).toThrowError('Los parámetros deben ser números');

            // Booleanos

            expect(function() {

                calculadora.sumar(true, 5);

            }).toThrowError('Los parámetros deben ser números');

            // Fechas

            expect(function() {

                calculadora.sumar(new Date(), 5);

            }).toThrowError('Los parámetros deben ser números');

        });

    });

    describe('Función Par/Impar', function() {

        it('debería identificar números pares correctamente', function() {

            expect(calculadora.esParImpar(2)).toBe('par');

            expect(calculadora.esParImpar(0)).toBe('par');

            expect(calculadora.esParImpar(-4)).toBe('par');

            expect(calculadora.esParImpar(100)).toBe('par');

        });

        it('debería identificar números impares correctamente', function() {

            expect(calculadora.esParImpar(1)).toBe('impar');

            expect(calculadora.esParImpar(3)).toBe('impar');

            expect(calculadora.esParImpar(-5)).toBe('impar');

            expect(calculadora.esParImpar(99)).toBe('impar');

        });

        it('debería lanzar error con parámetros no numéricos', function() {

            expect(function() {

                calculadora.esParImpar('texto');

            }).toThrowError('El parámetro debe ser un número');

            expect(function() {

                calculadora.esParImpar(null);

            }).toThrowError('El parámetro debe ser un número');

            expect(function() {

                calculadora.esParImpar(undefined);

            }).toThrowError('El parámetro debe ser un número');

            expect(function() {

                calculadora.esParImpar([]);

            }).toThrowError('El parámetro debe ser un número');

            expect(function() {

                calculadora.esParImpar({});

            }).toThrowError('El parámetro debe ser un número');

        });

        it('debería manejar casos extremos en par/impar', function() {

            expect(calculadora.esParImpar(0)).toBe('par');

            expect(calculadora.esParImpar(Number.MAX\_SAFE\_INTEGER)).toBe('impar'); // 9007199254740991 es impar

            expect(calculadora.esParImpar(Number.MIN\_SAFE\_INTEGER)).toBe('impar'); // -9007199254740991 es impar

        });

    });

    describe('Función Factorial', function() {

        it('debería calcular factoriales correctamente', function() {

            expect(calculadora.factorial(0)).toBe(1);

            expect(calculadora.factorial(1)).toBe(1);

            expect(calculadora.factorial(3)).toBe(6);

            expect(calculadora.factorial(5)).toBe(120);

        });

        it('debería lanzar error con números negativos', function() {

            expect(function() {

                calculadora.factorial(-1);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

        });

        it('debería lanzar error con números decimales', function() {

            expect(function() {

                calculadora.factorial(3.5);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

            expect(function() {

                calculadora.factorial(2.1);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

            expect(function() {

                calculadora.factorial(-1.5);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

        });

        it('debería lanzar error con parámetros no numéricos en factorial', function() {

            expect(function() {

                calculadora.factorial('5');

            }).toThrowError('El parámetro debe ser un número entero no negativo');

            expect(function() {

                calculadora.factorial(null);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

            expect(function() {

                calculadora.factorial(undefined);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

            expect(function() {

                calculadora.factorial([]);

            }).toThrowError('El parámetro debe ser un número entero no negativo');

        });

        it('debería manejar casos extremos en factorial', function() {

            expect(calculadora.factorial(0)).toBe(1);

            expect(calculadora.factorial(1)).toBe(1);

            // Factorial de números grandes puede causar overflow, pero la función debería manejarlo

            expect(calculadora.factorial(10)).toBe(3628800);

        });

    });

    describe('Función Potencia', function() {

        it('debería calcular potencias correctamente', function() {

            expect(calculadora.potencia(2, 3)).toBe(8);

            expect(calculadora.potencia(5, 2)).toBe(25);

            expect(calculadora.potencia(10, 0)).toBe(1);

        });

        it('debería manejar exponentes negativos', function() {

            expect(calculadora.potencia(2, -2)).toBe(0.25);

            expect(calculadora.potencia(4, -1)).toBe(0.25);

        });

        it('debería lanzar error con parámetros no numéricos', function() {

            expect(function() {

                calculadora.potencia('base', 2);

            }).toThrowError('Ambos parámetros deben ser números');

            expect(function() {

                calculadora.potencia(2, 'exponente');

            }).toThrowError('Ambos parámetros deben ser números');

            expect(function() {

                calculadora.potencia(null, 2);

            }).toThrowError('Ambos parámetros deben ser números');

            expect(function() {

                calculadora.potencia(2, undefined);

            }).toThrowError('Ambos parámetros deben ser números');

            expect(function() {

                calculadora.potencia([], {});

            }).toThrowError('Ambos parámetros deben ser números');

        });

        it('debería validar parámetros faltantes en potencia', function() {

            expect(function() {

                calculadora.potencia();

            }).toThrowError('Ambos parámetros deben ser números');

            expect(function() {

                calculadora.potencia(5);

            }).toThrowError('Ambos parámetros deben ser números');

        });

        it('debería manejar casos extremos en potencia', function() {

            expect(calculadora.potencia(0, 0)).toBe(1); // 0^0 = 1 por convención matemática

            expect(calculadora.potencia(1, 1000)).toBe(1); // 1 elevado a cualquier potencia es 1

            expect(calculadora.potencia(2, 0)).toBe(1); // Cualquier número elevado a 0 es 1

            expect(calculadora.potencia(-2, 2)).toBe(4); // Número negativo elevado a potencia par

            expect(calculadora.potencia(-2, 3)).toBe(-8); // Número negativo elevado a potencia impar

        });

    });

});

**PASO 5: Configurar package.json**

{

  "name": "jasmine-ejemplo",

  "version": "1.0.0",

  "description": "Ejemplo práctico de Jasmine JS para Actividad Pruebas",

  "main": "src/calculadora.js",

  "scripts": {

    "test": "jasmine",

    "test:verbose": "jasmine --verbose",

    "test:calculadora": "jasmine spec/calculadoraSpec.js",

    "test:tareas": "jasmine spec/gestorTareasSpec.js",

    "test:watch": "nodemon --exec 'npm test' --watch src --watch spec"

  },

  "license": "MIT",

  "devDependencies": {

    "jasmine": "^4.5.0",

    "nodemon": "^3.0.1"

  },

  "engines": {

    "node": ">=14.0.0"

  }

}

**PASO 6: Ejecutar pruebas**

* Ejecutar todas las pruebas: npm test
* Ejecutar pruebas específicas: npx jasmine spec/calculadoraSpec.js

**PASO 7: Salida esperada en Linux**

Started

........

48 specs, 0 failures

Finished in 0.012 seconds

Randomized with seed 12345

**VERIFICACIÓN EN LINUX:** **Ubuntu/Debian:**

bash

Comando adicional para verificar

lsb\_release -a

node --version && npm --version && npx jasmine –version

**COMANDOS DE VERIFICACIÓN COMPLETA:**

```bash

Verificar estructura del proyecto

find . -name "\*.js" -type f

tree -I node\_modules

Verificar sintaxis

node -c src/calculadora.js

node -c spec/calculadoraSpec.js

1. Para tener una noción de pruebas es necesario aclarar algunos conceptos:
2. ¿Qué son pruebas de software y por qué son importantes en el desarrollo?

Las pruebas de software son un proceso sistemático de evaluación y verificación de que una aplicación o sistema cumple con los requisitos especificados y funciona correctamente. Es una actividad que busca encontrar errores, defectos o comportamientos inesperados antes de que el software llegue a los usuarios finales.

**Importancia en el desarrollo:**

**a) Calidad:** Garantizan funcionamiento correcto, confiabilidad y estabilidad

**b) Detección Temprana de Errores:** Encontrar y corregir errores durante el desarrollo es significativamente más económico y rápido

**c) Costos:** Evitan gastos altos y pérdidas por errores en producción.

**d) Confianza del Usuario:** Un software bien probado genera confianza en los usuarios finales, reduce la cantidad de reportes de bugs y quejas de usuarios y mejora la reputación de la empresa y del producto

**e) Cumplimiento de Requisitos:**- Valida que el software cumple con todos los requisitos funcionales y no funcionales y asegura que las especificaciones del cliente sean satisfechas

**f) Seguridad:**- Las pruebas de seguridad protegen contra vulnerabilidades e identifican posibles puntos de entrada para ataques maliciosos asegurando la protección de datos sensibles

1. ¿Qué diferencia hay entre encontrar un error durante las pruebas y que lo encuentre un usuario después de usar el software?

Encontrar un error durante las pruebas permite corregirlo de forma controlada, rápida y económica, mientras que si lo detecta un usuario en producción genera mayores costos, afecta la confianza y puede dañar la reputación del software o de la empresa.

1. Explica con tus palabras qué es una prueba manual y qué es una prueba automatizada.

Una **prueba manual** es cuando una persona ejecuta casos de prueba paso a paso sin usar herramientas automáticas, validando de forma directa que el sistema funcione como se espera.

**Cuándo es útil:**

- Pruebas de usabilidad (¿es fácil de usar?)

- Pruebas exploratorias (encontrar problemas no documentados)

- Verificación visual (¿se ve bien la interfaz?)

- Pruebas de nuevas funcionalidades (primera vez que se prueban)

En cambio, una **prueba automatizada** utiliza scripts o programas que ejecutan pruebas de manera repetitiva y rápida, reduciendo tiempo y esfuerzo humano, ideal para tareas repetitivas o de gran volumen.

**Cuándo es útil:**

- Pruebas de regresión (verificar que cambios no rompan funcionalidades existentes)

- Pruebas repetitivas (que debo hacer frecuentemente)

- Pruebas de carga (simular muchos usuarios simultáneamente)

- Integración continua (ejecutar automáticamente con cada cambio de código)

1. ¿Qué diferencia existe entre una prueba funcional y una no funcional? Dar un ejemplo sencillo de cada una.

Una **prueba funcional** valida que el sistema haga lo que debe según los requisitos (el *qué* hace).

***Ejemplo****:* verificar que al ingresar usuario y contraseña correctos se pueda iniciar sesión.

Una **prueba no funcional** evalúa características de calidad del sistema, como rendimiento, seguridad o usabilidad (el *cómo* lo hace).

***Ejemplo****:* medir cuánto tarda el sistema en cargar después de iniciar sesión.

1. ¿Qué es una prueba unitaria? Dar un ejemplo sencillo con una función de tu aplicativo

Una prueba unitaria es un tipo de prueba que verifica el comportamiento de una unidad individual de código (generalmente una función o método) de forma aislada. Es la prueba más granular que existe, enfocándose en probar la lógica interna de pequeñas porciones de código sin depender de componentes externos.

***Ejemplo****:* - Probar que la función que calcula la comisión del 5% sobre un pedido retorne el valor correcto al recibir un monto.

- Validar que una función que **filtra productos por categoría** devuelva solo los que corresponden.

1. ¿Qué es una prueba de integración? Dar un ejemplo con dos módulos de tu aplicativo.

Una prueba de integración verifica que dos o más **componentes** o **módulos** del sistema trabajen correctamente cuando se combinan. A diferencia de las pruebas unitarias que aíslan componentes, las pruebas de integración verifican la interacción entre diferentes partes del sistema, asegurando que los datos fluyan correctamente y que los módulos se comuniquen como se espera.

***Ejemplo:***comprobar que cuando un **usuario** realiza un **pedido**, se registre correctamente en el módulo de **pedidos** y además genere la comisión en el módulo de **comisiones**.

1. Escribe los casos de prueba para comprobar el registro de un usuario nuevo funcione en el aplicativo.

* Ingresar datos válidos (nombre, correo, contraseña) → debe crearse el usuario.
* Correo ya registrado → debe mostrar error de duplicado.
* Contraseña demasiado corta o débil → debe mostrar mensaje de validación.
* Campos obligatorios vacíos → debe impedir el registro.
* Confirmar correo válido → debe enviar notificación o correo de confirmación.
* Contraseñas no coinciden → Contraseñas diferentes → Error de validación
* Email inválido → Email sin @ o sin .com → Error de formato
* Términos no aceptados → Checkbox no marcado → Error de términos
* Límites de longitud → Error de longitud
* Mostrar contraseña → Funcionalidad correcta

1. Nombra una herramienta de pruebas muy usada en JS/nodejs y una en Python

* En **JavaScript/Node.js**: **Jest**.

Jest es actualmente la herramienta de testing más popular en el ecosistema JavaScript/Node.js, desarrollada por Meta (Facebook). Es especialmente dominante en proyectos de React, pero se usa ampliamente en todo tipo de aplicaciones JavaScript.

* En **Python**: **Pytest**.

Pytest es el framework de testing más popular en Python, superando incluso al unittest nativo de Python. Es conocido por su simplicidad, poder y flexibilidad.

1. ¿Qué significa cobertura de pruebas y por qué no basta con tener el 100% de cobertura?

La cobertura de pruebas (test coverage o code coverage) es una métrica que mide qué porcentaje del código fuente es ejecutado durante la ejecución de las pruebas automatizadas. Es una forma cuantitativa de evaluar qué tan "completas" son nuestras pruebas en términos de líneas de código alcanzadas.

* La cobertura mide cantidad, no calidad
* No garantiza que se prueben todos los casos Edge
* No detecta lógica incorrecta
* No garantiza integración correcta
* No considera la complejidad ciclomática