<!-- Taller: Ingeniería de Software -->

Testing de Software{

<Tipo="Pruebas Unitarias"/>

}

UNIVERSIDAD ORT Uruguay

Facultad de **Ingeniería**

<!--Contenido:-->

¿Qué son las Pruebas Unitarias?

La base de un software confiable y mantenible

Rápidas

Se ejecutan en milisegundos

Confiables

Detectan errores temprano

Automáticas

No requieren intervención manual

Definición

Una prueba unitaria es un pequeño programa que verifica si una parte específica de tu código funciona correctamente.

Es como tener un asistente que constantemente revisa que cada función de tu programa haga exactamente lo que debe hacer, ni más ni menos.

Diferencias:

Sin pruebas:

```
// calculadora.js
function dividir(a, b) {
  return a / b;
}

// ¿Qué pasa si b es 0?
// ¿Funciona con números negativos?
// ¿Y con strings?

console.log(dividir(10, 2)); // 5
console.log(dividir(10, 0)); // Infinity 😱
```

❌ Problemas

⚠ No sabemos si funciona en todos los casos

⚠ Los errores aparecen en producción

⚠ Difícil de mantener y modificar

⚠ No hay documentación del comportamiento esperado

Con pruebas:

```
// calculadora.js
function dividir(a, b) {
  if (b === 0) {
    throw new Error('No se puede dividir por cero')
  }
  if (typeof a !== 'number' || typeof b !== 'number
    throw new Error('Los argumentos deben ser númer
  }
  return a / b;
}

// calculadora.test.js
test('debe dividir números correctamente', () => {
  expect(dividir(10, 2)).toBe(5);
});

test('debe lanzar error al dividir por cero', () =>
  expect(() => dividir(10, 0)).toThrow('No se puede
});
```

✅ **Beneficios**

- Sabemos exactamente cómo debe comportarse
- Los errores se detectan antes de producción
- Fácil de refactorizar con confianza
- Las pruebas documentan el comportamiento

Analogía: Construir una Casa

Sin Pruebas

Construimos toda la casa y al final descubrimos que la electricidad no funciona, las tuberías gotean y las puertas no cierran bien

Con Pruebas

Verificamos cada sistema (electricidad, plomería, puertas) mientras construimos. Al final, sabemos que todo funciona perfectamente.

Resultado

Una casa sólida, confiable y fácil de mantener. Si necesitamos hacer cambios, sabemos que no romperemos nada.

Historias Reales (Sin Pruebas)

📖 El Bug de Medianoche

Es viernes a las 11:47 PM. Tu aplicación está en producción con miles de usuarios. De repente, empiezan a llegar reportes: "No puedo hacer login", "La página se crashea", "Perdí todos mis datos". Tu fin de semana acaba de desaparecer.

⚠ El Problema

Sin pruebas, los errores llegan a producción

Con Pruebas

Con pruebas, este bug se habría detectado antes del deploy

📖 El Refactor Peligroso

Tu jefe te pide optimizar una función crítica. Haces los cambios, parece funcionar bien. Pero tres días después descubres que rompiste una funcionalidad en otra parte del sistema que dependía de un comportamiento específico de esa función.

⚠ El Problema

Cambiar código sin pruebas es como caminar con los ojos vendados

Con Pruebas

Las pruebas te avisan inmediatamente si rompes algo

Los Superpoderes de las Pruebas

Ahorro de Tiempo

Detectas errores en segundos, no en semanas

> Ejemplo:
> Una prueba que toma 50ms vs. un bug que toma 3 horas encontrar en producción

Ahorro de Dinero

Arreglar un bug en desarrollo cuesta 100x menos que en producción

> Ejemplo:
> Bug en desarrollo: 30 min. Bug en producción: 50 horas + usuarios perdidos

Menos Estrés

Duermes tranquilo sabiendo que tu código funciona

> Ejemplo:
> No más llamadas de emergencia los fines de semana

Mejor Código

Las pruebas te obligan a escribir código más limpio y modular

> Ejemplo:
> Funciones pequeñas, responsabilidades claras, menos acoplamiento

🚫 Mitos Sobre las Pruebas

⚠ "Las pruebas toman mucho tiempo"

Realidad: Escribir pruebas toma 20% más tiempo inicialmente, pero ahorras 300% del tiempo en debugging y mantenimiento.

⚠ "Mi código es simple, no necesita pruebas"

Realidad: El código "simple" es donde más bugs se esconden. Las funciones básicas son las que más se usan y más impacto tienen.

⚠ "Las pruebas son solo para proyectos grandes"

Realidad: Los proyectos pequeños se vuelven grandes. Es mejor empezar con buenas prácticas desde el día uno.

Conoce Jest

Tu herramienta para escribir pruebas unitarias

Jest es un framework de testing desarrollado por Meta (Facebook) que hace que escribir pruebas sea simple, rápido y divertido.

¿Por Qué Elegir Jest?

Cero Configuración

Funciona inmediatamente sin configuración compleja

```
Solo instala y ejecuta npm test
```

Watch Mode

Re-ejecuta pruebas automáticamente cuando cambias código

```
npm test -- --watch
```

Matchers Intuitivos

Sintaxis natural para escribir expectativas

```
expect(resultado).toBe(esperado)
```

Configuración básica de JEST

Desde la terminal, dentro del proyecto:

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

Los archivos de test deben seguir este formato:

- `archivo.test.js` (siendo archivo el nombre del archivo al que voy a escribirle los test)

✅ Deben estar en la misma carpeta que el archivo original o en una carpeta `/__tests__/`

Configuración básica de JEST

Configurar `package.json`
Agregar este script en la sección `"scripts"`:

```
"scripts": {
  "test": "jest"
}
```

Ejecutar los tests
Desde la terminal:

```
npm test
```

Documentacion oficial por cualquier duda: https://jestjs.io/

Jest se usa con tres elementos principales:

Parte	Significado
`test('nombre', función)`	• Es la **estructura básica del test**. Le damos un **nombre descriptivo** (una frase entre comillas) que diga **qué estamos probando**. Dentro de la función, escribimos el código del test.
`expect(valor)`	• Significa “Mirá este número que obtuve..”. • Lo usamos para decirle a Jest cuál es el **resultado real** que queremos verificar.
`.toBe(...), .toEqual(...)` etc.	• Estas son funciones especiales llamadas **matchers**. Se usan **después de `expect()`**. Le dicen a Jest **cómo comparar** el resultado real con lo que nosotros **esperamos**. Si coinciden, el test pasa ✅. Si no coinciden, el test falla ❌.

¿Como se ve una prueba con Jest?

Todas las pruebas siguen una estructura muy parecida:

```
test('descripción de la prueba', function() {
  // Código que ejecuta algo
    const resultado = funcionAEvaluar();

  // Comprobar resultado
   expect(resultado).matcher(valorEsperado);
});
```

- **`test('descripción', function() {})`** : Define la prueba con un nombre que explique qué estamos verificando.
- **`const resultado = funcionAEvaluar();`** : Ejecuta la función o acción que queremos probar.
- **`expect(resultado)`** **:** Le decimos a Jest: "quiero verificar este resultado".
- **`matcher(valorEsperado)`** : Elegimos cómo comparar el resultado (con toBe, toEqual, etc.).

`expect` y *matchers*: ¿qué son y cómo se usan?

`expect(valor)` - *Es* como decirle a Jest:

➡️ “Mirá este resultado que obtuve, y quiero asegurarme que sea el correcto.”

Luego usamos un **matcher** para decirle a Jest **cómo comparar** ese valor con lo que esperamos.

Por ejemplo:

- `expect(2 + 2).toBe(4);` // Espero que 2 + 2 sea exactamente 4
- `expect("hola").toEqual("hola");` // Espero que el texto sea igual a "hola"
- `expect(true).toBeTruthy();` // Espero que sea algo verdadero
- `expect(0).toBeFalsy();` // Espero que sea algo falso

¿Como se ve una prueba con Jest?

Ejemplo 1

```
test('la suma de 2+2 debe ser 4', function() {
  const  resultado = 2 + 2;
  expect(resultado).toBe(4);
});
```

¿Qué significa esto?

- `expect(2 + 2)` es:
  “Voy a comprobar el resultado de 2 + 2,”
- `.toBe(4)` dice:
  “…y espero que sea exactamente 4.”
- Si se cumple, el test **pasa**.
- Si no se cumple, el test **falla.**

¿Por qué están todas con la misma sintaxis?

Porque Jest **estandariza** cómo se escriben las pruebas para que todas sigan el mismo formato.

Eso hace que:

- Sea más fácil leerlas y mantenerlas.
- El sistema pueda ejecutarlas todas automáticamente.
- No importe si las escribió una sola persona o un equipo: todas siguen la misma lógica

EJEMPLO PRÁCTICO

Vamos a trabajar con un sistema de biblioteca.

- Guarda libros en una lista.
- Cada libro tiene: id, título, autor.
- Podemos **agregar**, **eliminar** y **buscar** libros.

Funcionalidades del sistema de bilbioteca

Vamos a hacer preguntas como:

- ¿Se agregó bien el libro?
- ¿El autor está correcto?
- ¿El ID es más grande que el anterior?

Con Jest vamos a poder responder todo eso automáticamente con `test()`, `expect()` y **matchers** como `toBe()`, `toEqual()`, etc.."

🧪 Matchers básicos en Jest

🔹 1. toBe() → Igualdad estricta (===)

Agregar un libro a la lista vacía

```js
test('agrega un libro a la lista', function() {
  biblioteca.agregarLibro("El Principito", "Antoine");
  expect(biblioteca.libros.length).toBe(1);
});
```

Se espera que, después de agregar un libro, haya exactamente **1 libro** en la lista.

- expect(...) le dice a Jest:
  ➡️ *"Comprobá este valor"*
- biblioteca.libros.length es el **número de libros en la lista**.
  Después de agregar un libro, debería ser **1**.
- .toBe(1) dice:
  ➡️ *"Espero que sea exactamente 1"*
  (compara usando ===, es decir, igualdad exacta).

✔️ Compara valores primitivos (números, strings, booleanos) con igualdad exacta.
❌ No se debe usar con objetos o arrays.

🧪 Matchers básicos en Jest

🔹 2. toEqual() → Igualdad de contenido

Verificar que el autor del libro sea el correcto

```js
test('el objeto libro tiene el autor correcto', function() {
  var libro = biblioteca.agregarLibro("1984", "George Orwell");
  expect(libro.autor).toEqual("George Orwell");
});
```

Revisa que al agregar el libro, el campo `autor` esté bien guardado.

✔ Compara objetos o arrays verificando que tengan el mismo contenido, no solo que sean iguales en referencia.

`toEqual(...)` se usa para comparar objetos o arrays.
Si estamos comparando propiedades dentro de **un objeto**, se suele usar `toEqual()` por claridad.

🧪 Matchers básicos en Jest

🔹 3. toBeTruthy() → Evalúa si algo es “verdadero”

Verifica que la lista no esté vacía

```
js


test('después de agregar un libro, la lista no está vacía', function() {
  biblioteca.agregarLibro("Don Quijote", "Cervantes");
  expect(biblioteca.libros.length > 0).toBeTruthy();
});
```

Confirma que hay al menos un libro.

✔️ Pasa si el valor es evaluado como verdadero en JavaScript (No debe ser: false, 0, cadena vacía, null undefined)

`toBeTruthy()` significa “esto es verdadero”

🧪 Matchers básicos en Jest

🔹 4. toBeFalsy() → Evalúa si algo es “falso”

Eliminar un ID que no existe devuelve null

```
js


test('al eliminar un ID inexistente, devuelve null', function() {
  var resultado = biblioteca.eliminarLibroPorId(999);
  expect(resultado).toBeFalsy();
});
```

Como no hay libro con ID 999, la función devuelve null.

✔ Pasa si el valor es considerado falso en JavaScript (como "", 0, false, null, etc.).

toBeFalsy() significa “esto es falso o null o undefined o 0”. En este caso, es null.

Matchers básicos en Jest (Numéricos)

5. toBeGreaterThan() → Mayor que

Verifica que los IDs van aumentando

```js
test('cada nuevo libro tiene un ID más alto', function() {
  var libro1 = biblioteca.agregarLibro("Libro 1", "Autor 1");
  var libro2 = biblioteca.agregarLibro("Libro 2", "Autor 2");
  expect(libro2.id).toBeGreaterThan(libro1.id);
});
```

Se espera que el segundo libro tenga un **ID más alto** que el primero.
`toBeGreaterThan(...)` compara si un valor es mayor que otro

✔ Pasa si el primer número es mayor que el segundo.
🔎 Se usa para comparar rangos, límites, máximos, etc.

Matchers básicos en Jest (Numéricos)

6. toBeLessThan() → Menor que

Verifica que no haya más de 3 libros.

```js
test('la biblioteca no tiene más de 3 libros', function() {
  biblioteca.agregarLibro("A", "1");
  biblioteca.agregarLibro("B", "2");
  expect(biblioteca.libros.length).toBeLessThan(3);
});
```

Solo se agregan 2 libros. Se verifica que haya **menos de 3**.
`toBeLessThan(...)` verifica que un número sea menor que otro.

✔ Pasa si el primer número es menor que el segundo.

🧪 Matchers básicos en Jest (Numéricos)

🔹 7. toBeCloseTo() → Cercanía con decimales

```js
test('la suma de 0.1 + 0.2 es aproximadamente 0.3', function() {
  const resultado = 0.1 + 0.2;
  expect(resultado).toBeCloseTo(0.3);
});
```

`toBeCloseTo(...)` se usa con números decimales o con pequeñas diferencia

```
const resultado = 0.1 + 0.2;
expect(resultado).toBeCloseTo(0.3); // ✅ correcto
expect(resultado).toBe(0.3);        // ❌ falla por precisión decimal
```

✔️ Pasa si el valor es aproximadamente igual (muy útil con números decimales).

🔍 Evita errores por imprecisiones típicas de JavaScript con floats.

Matcher	¿Qué verifica?	Usos comunes
toBe()	Igualdad exacta (===)	Comparar valores primitivos: números, strings, booleanos
toEqual()	Igualdad de contenido en objetos/arrays	Verificar si dos objetos o arrays tienen el mismo contenido
toBeTruthy()	Que el valor sea evaluado como verdadero	Comprobar que algo no esté vacío, nulo o sea false
toBeFalsy()	Que el valor sea evaluado como falso	Verificar que un valor sea considerado "falso" en JavaScript
toBeGreaterThan()	Mayor que	Compara si un valor es superior al otro
toBeLessThan()	Menor que	Compara si un valor es inferior al otro
toBeCloseTo()	Aproximación (decimales)	Comparar valores decimales con precisión (ej. 0.1 + 0.2 ≈ 0.3)

Comandos Jest más útiles

Comandos básicos

- `npm test`

  Ejecuta todas las pruebas una vez

- `npm test -- --watch`

  Modo watch: re-ejecuta automáticamente

- `npm test -- --coverage`

  Sirve para ver qué partes del código fueron usadas (ejecutadas) por los tests.

- npm test -- -t "nomnbreDelTest"

  Corre solo una prueba específica con Jest

Ejercicios para practicar

```
describe("PracticarConEjercicios", function() {

  test("hay un link para practicar", function() {
    var link = "https://ejerciciospracticos.netlify.app/";
    expect(link).toBeTruthy(); // ✅ Link encontrado
  });

  test("el estudiante hizo clic", function() {
    var confio = true;
    expect(confio).toBe(true); // ✅ Confiamos en vos :)
  });

});
```