Testing - Utest

Esteban Barrera Sanabria

Maria Camila Castro Porras

Esteban Garcia Gaitan

Tomas Saldana Leguizamo

Facultad de Ingeniería

Universidad Nacional de Colombia

Ingeniería de Software I

Oscar Eduardo Alvarez Rodriguez

Noviembre 2025

Test Unitarios:

En el presente documento se exhibe la creación de test unitarios orientados a probar la funcionalidad esencial de la aplicación en desarrollo, PocketVet, con el fin que el usuario pueda notar la robustez de unidades específicas de código, ya sea función, método o servicio, y que pueda funcionar correctamente por sí misma.

Dado que el backend se desarrolla mediante Node.js, JavaScript y ocasionalmente TypeScript, se opta a usar **Jest**, el framework más usado usualmente en situaciones de testeo por su amplio soporte y compatibilidad con TypeScript.

Cada uno de los 12 test contiene una breve explicación de cuál funcionalidad se prueba (descrita y mostrada en código), porque es importante y su resultado esperado (mostrado en el mismo entorno del proyecto), así como sus casos límite.

1. Validar email del usuario en el registro

La función **validateEmail()** verifica si un correo electrónico ingresado por un usuario cumple con los estándares mínimos de formato definidos por PocketVet.

Esta función utiliza expresiones regulares para identificar si el email es válido y sigue reglas comúnmente exigidas.

Requisitos:

Un email válido debe: Contener un solo símbolo @, tener una parte local no vacía (antes de @), incluir un dominio válido (ej: gmail, outlook, unal, etc.), incluir un TLD (ej: .com, .co, .edu), no contener espacios, no contener caracteres no permitidos como " , ; ( ) *, no iniciar ni terminar con un punto.

Se construye una función que la valida con estas condiciones (email no puede estar vacio, debe tener los caracteres incluidos siempre y ni el dominio ni el local deben estar vacios**:**

```javascript
if (/\s/.test(email)) return false;

const regex = /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$/;

const [local, domain] = email.split("@");
if (!local || !domain) return false;
```

Casos límite (mostrados en el código respectivo):

```
import { validateEmail } from "../Backend/src/utils/validateEmail";

describe("Email validation", () => {
  test("Valid emails", () => {
    const validEmails = [
      "usuario@gmail.com",
      "persona@mail.uni.edu.co",
      "juan.perez2024@mail.com",
    ];

    validEmails.forEach(email => {
      expect(validateEmail(email)).toBe(true);
    });
  });

  test("Invalid emails", () => {
    const invalidEmails = [
      "juanmail.com",                    // Sin @
      "juan@@gmail.com",                 // Dos @
      "usuario@",                        // Sin dominio
      "user@mail",                       // Sin TLD
      "@gmail.com",                      // Local part vacía
      "user name@gmail.com",             // Con espacios
      "user()@gmail.com",
    ];

    invalidEmails.forEach(email => {
      expect(validateEmail(email)).toBe(false);
    });
  });
})
```

Resultados obtenidos:

Después de instalar Jest en el proyecto, se procede a escribir `npm test` en la consola y se obtiene:

```
PASS tests/validateEmail.test.ts
  Email validation
    ✓ Valid emails (1 ms)
    ✓ Invalid emails (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.618 s
Ran all test suites.
```

Los test pasaron, pues lo que se esperaba que fuera *true* salió de esa manera, al igual que con *false*.

2. Validar contraseña del usuario en el registro

La función **validatePassword()** verifica si la contraseña ingresada por un usuario cumple con los requisitos mínimos de seguridad definidos por PocketVet.

Esta función se apoya en expresiones regulares y operaciones básicas sobre cadenas para asegurar que las contraseñas sean lo suficientemente robustas y difíciles de adivinar.

Requisitos:

Una contraseña válida debe: Tener un mínimo de 8 caracteres, incluir al menos una letra mayúscula, incluir al menos una letra minúscula, incluir al menos un número, incluir al menos un símbolo especial (por ejemplo: ! @ # $ % ^ & * - _), no contener espacios y no estar vacía

La función usada en el proyecto aplica las siguientes validaciones:

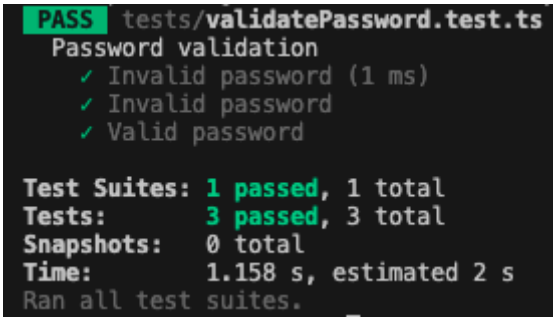
```javascript
if (!password || password.trim() === "") return false;

if (/\s/.test(password)) return false;

const regex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[!@#$%^&*._-])[A-Za-z\d!@#$%^&*._-]{8,}$/;

return regex.test(password);
```

Casos límite (mostrados en el código respectivo):

```javascript
import { validatePassword } from "../Backend/src/utils/validatePassword";

describe("Password validation", () => {
  test("Invalid password", () => {
    expect(validatePassword("Ab1!")).toBe(false);
  });

  test("Invalid password", () => {
    const cases = [
      "Abcd1234",   // Falta carácter especial
      "Abcd!!!!",   // Falta número
      "1234@@@@",   // Falta letra
      "abcd1234!",  // Falta mayúscula (si es requerida)
    ];

    cases.forEach(pwd => {
      expect(validatePassword(pwd)).toBe(false);
    });
  });

  test("Valid password", () => {
    expect(validatePassword("Abcd1234!")).toBe(true);
    expect(validatePassword("Test2024#")).toBe(true);
  });
});
```

Resultados obtenidos:

Se procede a escribir `npm test` en la consola y se obtiene:



Los test pasaron, pues lo que se esperaba que fuera *true* salió de esa manera, al igual que con *false*.

3. Calcular edad de la mascota con base en la fecha de nacimiento

En PocketVet, cada mascota registrada debe mostrar su edad en años a partir de su fecha de nacimiento. Para esta función se impleme ta **calculatePetAge()**, la cual toma una fecha (string o Date) y retorna la edad entera en años.

Requisitos:

La función debe:

1. Asegurar que la fecha sea válida.
2. Asegurar que la fecha no esté en el futuro.
3. Calcular correctamente los años teniendo en cuenta año, mes y dia
4. Retornar la edad como entero.
5. Retornar `null` o `-1` en caso de fecha inválida.

La función usada en el proyecto aplica las siguientes validaciones:

```javascript
const date = new Date(birthdate);
if (isNaN(date.getTime())) return -1; // Formato invalido

const now = new Date();
if (date > now) return -1; // Futura
let age = now.getFullYear() - date.getFullYear();
const monthDiff = now.getMonth() - date.getMonth();
const dayDiff = now.getDate() - date.getDate();

if (monthDiff < 0 || (monthDiff === 0 && dayDiff < 0)) age--;

return age;
```

Casos límite (mostrados en el código respectivo):

```
import { calculatePetAge } from "../Backend/src/utils/calculateAge";

describe("calculateAge()", () => {
  test("Debe calcular correctamente la edad cuando el cumpleaños ya pasó este año", () => {
    const birth = "2000-01-10";
    expect(calculatePetAge(birth)).toBe(new Date().getFullYear() - 2000);
  });

  test("Debe calcular correctamente la edad cuando el cumpleaños es hoy", () => {
    const today = new Date();
    const yyyy = today.getFullYear() - 20;
    const mm = String(today.getMonth() + 1).padStart(2, "0");
    const dd = String(today.getDate()).padStart(2, "0");

    const birth = `${yyyy}-${mm}-${dd}`;
    expect(calculatePetAge(birth)).toBe(20);
  });

  test("Debe calcular correctamente la edad cuando el cumpleaños aún no ha pasado este año", () => {
    const today = new Date();
    const yyyy = today.getFullYear() - 20;
    const mm = String(today.getMonth() + 2).padStart(2, "0"); // Próximo mes
    const dd = "01";

    const birth = `${yyyy}-${mm}-${dd}`;

    expect(calculatePetAge(birth)).toBe(19);
  });
});
```

Resultados obtenidos:

Se procede a escribir npm test en la consola y se obtiene:

```
PASS tests/calculatePetAge.test.ts
  calculatePetAge()
    ✓ Debe calcular correctamente la edad cuando el cumpleaños ya pasó este año (1 ms)
    ✓ Debe calcular correctamente la edad cuando el cumpleaños es hoy
    ✓ Debe calcular correctamente la edad cuando el cumpleaños aún no ha pasado este año

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.823 s
Ran all test suites.
```

Los test pasaron, pues lo que se esperaba que fuera *true* salió de esa manera, al igual que con *false*.

4. Validación de Estado de Evento

Descripción
En PocketVet, cada evento del calendario (vacunas, consultas, etc.) debe tener un estado coherente con su fecha programada. La función validarEstadoEvento() verifica que el estado del evento sea lógico según la fecha actual.

Requisitos:

1. Aceptar solo estados válidos: 'Pendiente', 'Completo', 'Cancelado'
2. Impedir que un evento futuro tenga estado 'Completo'
3. Permitir eventos pasados con estado 'Pendiente' (con advertencia)
4. Retornar objeto con validación, error y advertencias

Lógica implementada:

```javascript
const estadosValidos = ['Pendiente', 'Completo', 'Cancelado'];
const ahora = new Date();

if (!estadosValidos.includes(estado)) {
  return { valido: false, error: 'Estado de evento no válido' };
}

if (estado === 'Completo' && fechaEvento > ahora) {
  return { valido: false, error: 'No se puede completar un evento futuro' };
}

if (estado === 'Pendiente' && fechaEvento < ahora) {
  return { valido: true, advertencia: 'Evento pendiente en fecha pasada' };
}
```

Casos límite (mostrados en el respectivo código):

```javascript
const { validarEstadoEvento } = require('../Backend/src/utils/eventStatusValidation');

describe('Validación de Estado de Evento', () => {
  test('Evento pendiente en fecha futura - VÁLIDO', () => {
    const fechaFutura = new Date();
    fechaFutura.setDate(fechaFutura.getDate() + 1);

    const resultado = validarEstadoEvento('Pendiente', fechaFutura);

    expect(resultado.valido).toBe(true);
    expect(resultado.advertencia).toBeUndefined();
  });

  test('Evento completado en fecha pasada - VÁLIDO', () => {
    const fechaPasada = new Date();
    fechaPasada.setDate(fechaPasada.getDate() - 1);

    const resultado = validarEstadoEvento('Completo', fechaPasada);

    expect(resultado.valido).toBe(true);
  });

  test('Evento completado en fecha futura - INVÁLIDO', () => {
    const fechaFutura = new Date();
    fechaFutura.setDate(fechaFutura.getDate() + 1);

    const resultado = validarEstadoEvento('Completo', fechaFutura);

    expect(resultado.valido).toBe(false);
    expect(resultado.error).toBe('No se puede completar un evento futuro');
  });
});
```

Resultados obtenidos:

```
camilacastro@Camilas-MacBook-Air Proyecto % npm test -- tests/eventEstado.test.ts


> pocketvet@1.0.0 test
> jest tests/eventEstado.test.ts

 PASS  tests/eventEstado.test.ts
  Validación de Estado de Evento
    ✓ Evento pendiente en fecha futura - VÁLIDO (2 ms)
    ✓ Evento completado en fecha pasada - VÁLIDO (1 ms)
    ✓ Evento completado en fecha futura - INVÁLIDO

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.749 s, estimated 2 s
Ran all test suites matching tests/eventEstado.test.ts.
```

5. Validación de Fecha de Evento

Descripción

En PocketVet, los eventos no pueden ser programados en fechas pasadas (excepto eventos históricos). La función validarFechaEvento() asegura que la fecha del evento sea actual o futura.

Requisitos:

1. Asegurar que la fecha no sea en el pasado
2. Considerar válida la fecha actual
3. Considerar válidas todas las fechas futuras
4. Comparar solo día/mes/año (ignorando horas)
5. Retornar objeto con validación y mensaje de error

Lógica implementada:

```javascript
const ahora = new Date();
const hoy = new Date(ahora.getFullYear(), ahora.getMonth(), ahora.getDate());
const fechaEventoSinHora = new Date(
  fechaEvento.getFullYear(),
  fechaEvento.getMonth(),
  fechaEvento.getDate()
);

if (fechaEventoSinHora < hoy) {
  return { valido: false, error: 'La fecha del evento no puede ser en el pasado' };
}
```

Casos límite (mostrados en el respectivo código):

```javascript
const { validarFechaEvento } = require('../Backend/src/utils/eventDateValidation');

describe('Validación de Fecha de Evento', () => {
  test('Fecha futura - VÁLIDO', () => {
    const fechaFutura = new Date();
    fechaFutura.setDate(fechaFutura.getDate() + 5);

    const resultado = validarFechaEvento(fechaFutura);

    expect(resultado.valido).toBe(true);
  });

  test('Fecha de hoy - VÁLIDO', () => {
    const hoy = new Date();

    const resultado = validarFechaEvento(hoy);

    expect(resultado.valido).toBe(true);
  });

  test('Fecha pasada - INVÁLIDO', () => {
    const fechaPasada = new Date();
    fechaPasada.setDate(fechaPasada.getDate() - 1);

    const resultado = validarFechaEvento(fechaPasada);

    expect(resultado.valido).toBe(false);
    expect(resultado.error).toBe('La fecha del evento no puede ser en el pasado');
  });
});
```

Resultados obtenidos:

```
camilacastro@Camilas-MacBook-Air Proyecto % npm test -- tests/eventFecha.test.ts

> pocketvet@1.0.0 test
> jest tests/eventFecha.test.ts

 PASS  tests/eventFecha.test.ts
  Validación de Fecha de Evento
    ✓ Fecha futura - VÁLIDO (2 ms)
    ✓ Fecha de hoy - VÁLIDO
    ✓ Fecha pasada - INVÁLIDO

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.85 s, estimated 2 s
Ran all test suites matching tests/eventFecha.test.ts.
```

6. Validación de Fechas de Vacuna

Descripción
En PocketVet, el carné digital de vacunación requiere validación estricta de fechas para garantizar que las vacunas sean válidas y no estén vencidas. La función validarFechasVacuna() verifica la coherencia temporal entre fecha de aplicación y vencimiento.

Requisitos:

1. Asegurar que la fecha de vencimiento sea posterior a la de aplicación
2. Impedir fechas de vencimiento iguales a la aplicación
3. Verificar que la vacuna no esté vencida al momento de aplicación
4. Retornar objeto con validación y mensaje de error específico

Lógica implementada:

JavaScript

```
if (fechaVencimiento <= fechaAplicacion) {
  return { valido: false, error: 'La fecha de vencimiento no puede ser menor
o igual a la fecha de aplicación' };
}

const ahora = new Date();
if (fechaVencimiento < ahora) {
  return { valido: false, error: 'La vacuna está vencida' };
}
```

Casos límite (mostrados en el respectivo código):

```
const { validarFechasVacuna } = require('../Backend/src/utils/vacunaDateValidation');

describe('Validación de Fechas de Vacuna', () => {
  test('Vencimiento después de aplicación - VÁLIDO', () => {
    const fechaAplicacion = new Date('2025-01-01');
    const fechaVencimiento = new Date('2026-01-01');

    const resultado = validarFechasVacuna(fechaAplicacion, fechaVencimiento);

    expect(resultado.valido).toBe(true);
  });

  test('Vencimiento igual a aplicación - INVÁLIDO', () => {
    const mismaFecha = new Date('2024-06-01');

    const resultado = validarFechasVacuna(mismaFecha, mismaFecha);

    expect(resultado.valido).toBe(false);
    expect(resultado.error).toBe('La fecha de vencimiento no puede ser menor o igual a la fecha de aplicación');
  });

  test('Vencimiento antes de aplicación - INVÁLIDO', () => {
    const fechaAplicacion = new Date('2024-06-01');
    const fechaVencimiento = new Date('2023-06-01');

    const resultado = validarFechasVacuna(fechaAplicacion, fechaVencimiento);

    expect(resultado.valido).toBe(false);
  });
});
```

Resultados obtenidos:

```
camilacastro@Camilas-MacBook-Air Proyecto % npm test -- tests/vacunaFechas.test.ts

> pocketvet@1.0.0 test
> jest tests/vacunaFechas.test.ts

 PASS  tests/vacunaFechas.test.ts
  Validación de Fechas de Vacuna
    ✓ Vencimiento después de aplicación - VÁLIDO (1 ms)
    ✓ Vencimiento igual a aplicación - INVÁLIDO
    ✓ Vencimiento antes de aplicación - INVÁLIDO (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.748 s, estimated 2 s
Ran all test suites matching tests/vacunaFechas.test.ts.
```

7. Validación de Formato de los Documentos Ingresados

Descripción

En PocketVet, ciertos formularios requieren que el usuario cargue documentos en formato PDF como comprobantes válidos (historial médico, certificados, etc.).
La función validatePDF() permite asegurar que los archivos suministrados cumplen con el formato requerido, evitando extensiones incorrectas que puedan generar errores o riesgos en el sistema.

Esta validación verifica que el archivo tenga extensión .pdf, sin importar mayúsculas o minúsculas.

Requisitos:

1. Confirmar que el archivo proporcionado termine en la extensión .pdf.

2. Prevenir la carga de formatos no permitidos como .png, .jpg, .docx
3. Validar nombres con múltiples puntos (ej. *"certificado.vacunacion.v1.pdf"*).
4. Retornar un objeto indicando si el archivo es válido y, en caso contrario, un mensaje de error claro.

Lógica implementada:

```
JavaScript

export function validatePDF(filename: string) {
  const lower = filename.toLowerCase();

  if (!lower.endsWith(".pdf")) {
    return { valido: false, error: "El archivo debe estar en formato PDF
(.pdf)" };
  }

  return { valido: true, error: null };
}
```

Casos límite (mostrados en el respectivo código):

```
import { validatePDF } from "../Backend/src/utils/validatePDF";

Complexity is 4 Everything is cool!
describe("validatePDF()", () => {
  test("Debe aceptar archivos PDF válidos", () => {
    expect(validatePDF("documento.pdf")).toBe(true);
  });

  test("Debe rechazar archivos no PDF", () => {
    expect(validatePDF("imagen.png")).toBe(false);
  });

  test("Debe rechazar archivos sin extensión", () => {
    expect(validatePDF("archivo")).toBe(false);
  });
});
```

Resultados obtenidos:

```
PASS Proyecto/tests/validatePDF.test.ts
  validatePDF()
    √ Debe aceptar archivos PDF válidos (4 ms)
    √ Debe rechazar archivos no PDF (1 ms)
    √ Debe rechazar archivos sin extensión

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.583 s
Ran all test suites matching tests/validatePDF.test.ts.
```

8. Validación del Tamaño de los Archivos

Descripción

En PocketVet, los usuarios pueden subir archivos como certificados médicos, fotos de mascotas o documentos PDF.

Para garantizar un desempeño adecuado del sistema y evitar archivos excesivamente grandes, se implementa la función validateFileSize(), la cual verifica que el archivo no supere un límite definido (por defecto 5 MB).

Requisitos:

1. Aceptar un tamaño en bytes.
2. Rechazar archivos que tengan tamaño 0 o negativo.
3. Validar que el archivo no exceda el límite permitido (5 MB si no se indica otro).
4. Retornar un objeto indicando si el archivo es válido y, en caso contrario, el mensaje de error.

Lógica implementada:

JavaScript

```
export function validateFileSize(bytes: number, maxMB: number = 5) {
  const maxBytes = maxMB * 1024 * 1024;

  if (bytes <= 0) {
    return { valido: false, error: "El tamaño del archivo no es válido" };
  }

  if (bytes > maxBytes) {
    return { valido: false, error: `El archivo supera el límite de ${maxMB}
MB` };
  }
```

```
  return { valido: true, error: null };
}
```

Casos límite (mostrados en el respectivo código):

```
import { validateFileSize } from "../Backend/src/utils/validateFileSize";

Complexity is 4 Everything is cool!
describe("validateFileSize()", () => {

  test("Debe aceptar un archivo pequeño (1MB)", () => {
    const size = 1 * 1024 * 1024; // 1 MB
    expect(validateFileSize(size)).toEqual({ valido: true, error: null });
  });

  test("Debe aceptar un archivo exactamente en el límite (5MB)", () => {
    const size = 5 * 1024 * 1024; // 5 MB
    expect(validateFileSize(size)).toEqual({ valido: true, error: null });
  });

  test("Debe rechazar un archivo que supera el límite (6MB)", () => {
    const size = 6 * 1024 * 1024;
    expect(validateFileSize(size)).toEqual({
      valido: false,
      error: "El archivo supera el límite de 5 MB",
    });
  });

});
```

Resultados obtenidos:

```
PASS Proyecto/tests/validateFileSize.test.ts
  validateFileSize()
    √ Debe aceptar un archivo pequeño (1MB) (3 ms)
    √ Debe aceptar un archivo exactamente en el límite (5MB) (1 ms)
    √ Debe rechazar un archivo que supera el límite (6MB) (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.527 s
Ran all test suites matching tests/validateFileSize.test.ts.
```

9. Validación de Rol del Usuario

Descripción

En PocketVet, cada usuario registrado pertenece a un tipo de rol que determina sus permisos dentro del sistema.

Para garantizar que solamente existan roles reconocidos por la plataforma, se implementa la función validateUserRole(), la cual valida que el rol recibido corresponda a uno de los permitidos por el sistema.

Requisitos:

1. Verificar que el rol pertenezca a la lista de roles válidos.
2. Rechazar cualquier string diferente.
3. Retornar un objeto indicando si el rol es válido.
4. Retornar un mensaje de error claro cuando sea inválido.

Lógica implementada:

```javascript
export function validateUserRole(role: string) {
  const validRoles = ["admin", "cliente", "veterinario"];

  if (!validRoles.includes(role)) {
    return { valido: false, error: "Rol de usuario inválido" };
  }

  return { valido: true, error: null };
}
```

Casos límite (mostrados en el respectivo código):

```
import { validateUserRole } from "../Backend/src/utils/validateUserRole";

Complexity is 4 Everything is cool!
describe("validateUserRole()", () => {

  test("Debe aceptar rol 'admin'", () => {
    expect(validateUserRole("admin")).toEqual({
      valido: true,
      error: null
    });
  });

  test("Debe aceptar rol 'cliente'", () => {
    expect(validateUserRole("cliente")).toEqual({
      valido: true,
      error: null
    });
  });

  test("Debe rechazar un rol inválido", () => {
    expect(validateUserRole("superusuario")).toEqual({
      valido: false,
      error: "Rol de usuario inválido"
    });
  });

});
```

Resultados obtenidos:

```
PASS Proyecto/tests/validateUserRole.test.ts
  validateUserRole()
    √ Debe aceptar rol 'admin' (4 ms)
    √ Debe aceptar rol 'cliente' (1 ms)
    √ Debe rechazar un rol inválido (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.751 s
Ran all test suites matching tests/validateUserRole.test.ts.
```

10. Validación de Campos del Registro de Mascota

Descripción

En el registro de mascotas de PocketVet, es obligatorio que todos los datos fundamentales sean llenados para poder crear una mascota correctamente en la base de datos.
Para asegurar esto se utiliza la función **validatePetFields()**, la cual verifica que ninguno de los campos esenciales esté vacío.

Requisitos

1. Validar que los campos obligatorios contengan valores.
2. Rechazar cualquier mascota con uno o más campos vacíos.
3. Retornar false si falta algún dato.
4. Retornar true si todos los campos están completos.
5. Mantener la función independiente del componente (usable para validación y testing).

Lógica implementada:

JavaScript

```
export function validatePetFields(pet: any) {
  const { petImage, name, species, breed, weight, age } = pet;

  if (!petImage || !name || !species || !breed || !weight || !age) {
    return false;
  }

  return true;
}
```

Casos límite (mostrados en el respectivo código):

```
import { validatePetFields } from "../Frontend/src/utils/validatePet";

describe( name: "validatePetFields()", () :void => { new *
  test( name: "Debe retornar false si algún campo está vacío", () :void => {
    const petData = {
      petImage: "",
      name: "",
      species: "Perro",
      breed: "Labrador",
      weight: "10",
      age: "2",
    };

    expect(validatePetFields(petData)).toBe( expected: false);
  });

  test( name: "Debe retornar true si todos los campos están llenos", () :void => {
    const petData = {
      petImage: "imagen.png",
      name: "Firulais",
      species: "Perro",
      breed: "Labrador",
      weight: "20",
      age: "3",
    };

    expect(validatePetFields(petData)).toBe( expected: true);
  });

  test( name: "Debe retornar false si falta **cualquier** campo", () :void => {
```

```
    const petData = {
      petImage: "imagen.png",
      name: "Firulais",
      species: "Perro",
      breed: "Labrador",
      // weight falta
      age: "3",
    };

    expect(validatePetFields(petData)).toBe( expected: false);
  });
});
```

Resultados obtenidos:

```
PASS tests/validarCampos.test.ts
  validatePetFields()
    √ Debe retornar false si algún campo está vacío (7 ms)
    √ Debe retornar true si todos los campos están llenos
    √ Debe retornar false si falta **cualquier** campo
```

11. Validación de Campos del Registro de usuario

Descripción

En el registro de usuarios de PocketVet, es obligatorio que todos los datos fundamentales sean llenados para poder crear una usuario correctamente en la base de datos.
Para asegurar esto se utiliza la función **validateRegisterFields()**, la cual verifica que ninguno de los campos esenciales esté vacío.

Requisitos

1. Validar que los campos obligatorios contengan valores.
2. Rechazar cualquier usuario con uno o más campos vacíos.
3. Retornar false si falta algún dato.
4. Retornar true si todos los campos están completos.
5. Mantener la función independiente del componente (usable para validación y testing).

Lógica implementada:

JavaScript

```
export function validateRegisterFields(data: any) {
   const { email, password, confirmPassword, fullName } = data;

   if (!email || !password || !confirmPassword || !fullName) {
       return {
       valido: false,
       error: "Por favor completa todos los campos obligatorios",
       };
   }
```

Casos límite (mostrados en el respectivo código):

```ts
interface Usuario {  Show usages  new *
  nombre?: any;
  email?: any;
  password?: any;
  edad?: any;
}

function validarUsuario(usuario: Usuario): string[] {  Show usages  new *
  const faltantes: string[] = [];

  if (!usuario.nombre) faltantes.push( ...items: "nombre");
  if (!usuario.email) faltantes.push( ...items: "email");
  if (!usuario.password) faltantes.push( ...items: "password");

  if (usuario.edad === undefined || usuario.edad === null) {
    faltantes.push( ...items: "edad");
  }

  return faltantes;
}

describe( name: "Validación de campos faltantes en usuario", () :void => {  new *

  test( name: "Caso 1: Usuario totalmente vacío", () :void => {
    const usuario = {};
    const faltantes :string[] = validarUsuario(usuario);
    console.log( message: "Campos faltantes detectados:", faltantes);

    expect(faltantes).toEqual(["nombre", "email", "password", "edad"]);
  });
```

```ts
  test( name: "Caso 2: Campos presentes pero vacíos ('')", () :void => {
    const usuario = { nombre: "", email: "", password: "", edad: "" };
    const faltantes :string[] = validarUsuario(usuario);
    console.log( message: "Campos faltantes detectados:", faltantes);
    expect(faltantes).toEqual(["nombre", "email", "password"]);
  });

  test( name: "Caso 3: Edad = 0 (caso límite válido)", () :void => {
    console.log( message: "→ Probando edad 0 (válida)");
    const usuario = { nombre: "Ana", email: "a@b.com", password: "123", edad: 0 };
    const faltantes :string[] = validarUsuario(usuario);
    console.log( message: "Campos faltantes detectados:", faltantes);

    expect(faltantes).toEqual([]);
  });

  test( name: "Caso 4: Falta solo 1 campo", () :void => {
    console.log( message: "→ Falta únicamente el campo email");
    const usuario = { nombre: "Ana", email: null, password: "123", edad: 20 };
    const faltantes :string[] = validarUsuario(usuario);
    console.log( message: "Campos faltantes detectados:", faltantes);

    expect(faltantes).toEqual(["email"]);
  });

  test( name: "Caso 5: Usuario completamente válido", () :void => {
    console.log( message: "→ Probando usuario válido");
    const usuario = { nombre: "Luis", email: "l@l.com", password: "abc", edad: 25 };
    const faltantes :string[] = validarUsuario(usuario);
```

```
59        const usuario = { nombre: "Luis", email: "l@l.com", password: "abc", edad: 25 };
60        const faltantes : string[] = validarUsuario(usuario);
61
62        expect(faltantes).toEqual([]);
63      });
64
65    });
```

Resultados obtenidos:

```
PASS tests/registrarCamposUsuario.test.ts (11.497 s)
  ● Console

    console.log
      Campos faltantes detectados: [ 'nombre', 'email', 'password', 'edad' ]

      at Object.<anonymous> (tests/registrarCamposUsuario.test.ts:27:13)

    console.log
      Campos faltantes detectados: [ 'nombre', 'email', 'password' ]

      at Object.<anonymous> (tests/registrarCamposUsuario.test.ts:35:13)

    console.log
      → Probando edad 0 (válida)

      at Object.<anonymous> (tests/registrarCamposUsuario.test.ts:40:13)

    console.log
      Campos faltantes detectados: []

      at Object.<anonymous> (tests/registrarCamposUsuario.test.ts:43:13)

    console.log
      → Falta únicamente el campo email

      at Object.<anonymous> (tests/registrarCamposUsuario.test.ts:49:13)

    console.log
      Campos faltantes detectados: [ 'email' ]

      at Object.<anonymous> (tests/registrarCamposUsuario.test.ts:52:13)

    console.log
      → Probando usuario válido
```

12. Validación de filtro de eventos por dia

En PocketVet, el frontend necesita mostrar únicamente los eventos que ocurren en el día seleccionado por el usuario.
Para esto se implementa la función **getEventsForDate()**, la cual recibe una fecha y una lista de eventos, y devuelve únicamente aquellos cuya fecha de inicio coincide exactamente con la indicada.

Esta función es especialmente útil para mostrar en pantalla los eventos programados para un día específico en el calendario, evitando procesamientos innecesarios y reduciendo errores al comparar fechas.

Requisitos

La función **getEventsForDate()** debe cumplir los siguientes criterios:

1. **Filtrar correctamente** los eventos cuya fecha (`fecha_inicio`) coincida con la fecha dada.
2. Comparar fechas **usando solo la parte de día/mes/año**, ignorando horas.
3. Retornar un **arreglo de eventos**, vacío si no coincide ninguno.
4. Permitir recibir un arreglo externo (mock o real) para facilitar pruebas y evitar dependencias internas.
5. Mantener compatibilidad con fechas en formato **string ISO**.

Lógica implementada:

```javascript
export const getEventsForDate = (date: Date, events: Event[]) => {
  return events.filter(event => {
    const eventDate = new Date(event.fecha_inicio);
    return eventDate.toDateString() === date.toDateString();
  });
};
```

Casos límite (mostrados en el respectivo código):

```ts
import { getEventsForDate, Event } from "../Frontend/src/utils/getEventsForDate";

describe( name: "getEventsForDate()", () :void => { new *

  const mockEvents: Event[] = [
    { nombre: "Evento A", fecha_inicio: "2024-10-10" },
    { nombre: "Evento B", fecha_inicio: "2024-10-10" },
    { nombre: "Evento C", fecha_inicio: "2024-11-01" }
  ];

  test( name: "Filtra eventos correctamente por fecha", () :void => {
    console.log( message: "→ Probando filtrado por fecha 2024-10-10...");

    const date = new Date( value: "2024-10-10");
    const result :Event[] = getEventsForDate(date, mockEvents);

    console.log( message: "Eventos encontrados:", result.map(e :Event => e.nombre));

    expect(result.map((e: Event) :string => e.nombre)).toEqual(["Evento A", "Evento B"]);

    console.log( message: "✓ Test finalizado correctamente.");
  });

});
```

Resultados obtenidos:

```
PASS tests/filtroEventosPorDia.test.ts
  ● Console

    console.log
      → Probando filtrado por fecha 2024-10-10...

      at Object.<anonymous> (tests/filtroEventosPorDia.test.ts:12:13)

    console.log
      Eventos encontrados: [ 'Evento A', 'Evento B' ]

      at Object.<anonymous> (tests/filtroEventosPorDia.test.ts:17:13)

    console.log
      ✓ Test finalizado correctamente.

      at Object.<anonymous> (tests/filtroEventosPorDia.test.ts:21:13)


Test Suites: 10 passed, 10 total
Tests:       30 passed, 30 total
Snapshots:   0 total
Time:        2.772 s
Ran all test suites.
```