Sprint 2 (Quality) (Pepe)

Planeación de sistemas de software(Gpo 102)

Team 24:

Julián Enrique Espinoza Valenzuela | A01254679

Santiago Gutiérrez González | A00572499

Alejandro Moncada Espinosa | A01638343

Ana Camila Jimenez Mendoza | A01174422

Jorge Ivan Sanchez Gonzalez | A01761414

Guadalajara, Jalisco

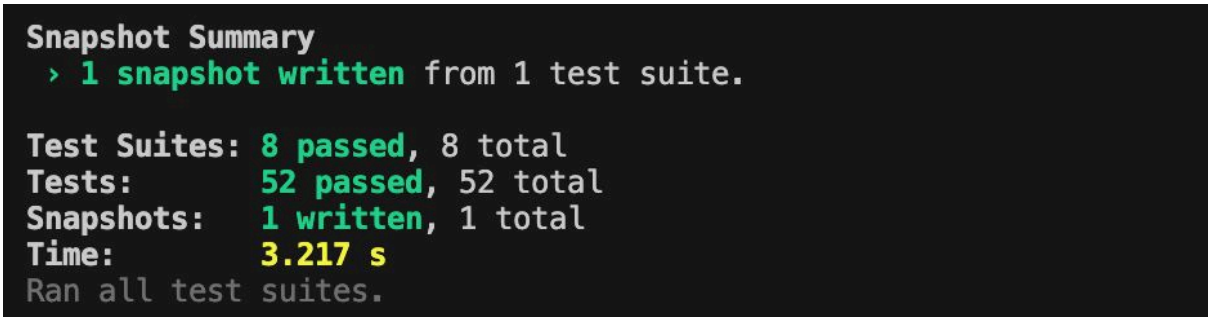
April, 2025

Introducción

Este documento describe las pruebas unitarias implementadas para los componentes principales de nuestra aplicación de gestión de tareas. Las pruebas fueron desarrolladas utilizando Jest y React Testing Library para asegurar el correcto funcionamiento de la interfaz de usuario y la lógica de negocio.

Tema	Descripción (La descripción dada en el test	Archivo
Mock functions (fn or spy)	DashboardInput Component	DashboardInput.Test
Mock modules	Presente en el archivo con la función jest.mock	Report.test
Snapshots	Dashboard component matches snapshot	Dashboard.test
React Testing Library and User Event Testing Library	Report Component	Report.test
Form Testing (user click or type events)	does not submit form when isInserting is true	DashboardInput.Test
Avoid implementation details	Práctica seguida	Práctica seguida
Mocking HTTP Requests with Mock Server Worker	Entre otras cosas se usa msw/node y server.use()	Todos
Testing With Contexts	Estos tests no se implementaron puesto que no hemos creado contextos para nuestro proyecto ni está en los planes	-
Testing Custom Hooks	No se han creado custom hooks de momento por lo que no existe nada que probar en este aspecto	-
Dynamic Test Data	Inicio del archivo	DashboardContent.test

Resultado de Pruebas



Ejemplos de test válidos implementados por cada tema solicitado:

Mock functions (fn or spy)

JavaScript

```
// Test rendering of the component
 test("renders the component with correct elements", async () => {
   // Mock the addItem function
   const mockAddItem = jest.fn();

   // Render the component
   render(<DashboardInput addItem={mockAddItem} isInserting={false} />);

   // Check if all form elements are rendered
   expect(screen.getByPlaceholderText("Title")).toBeInTheDocument();
   expect(screen.getByPlaceholderText("Description")).toBeInTheDocument();
   expect(screen.getByText("Responsible")).toBeInTheDocument();
   expect(screen.getByPlaceholderText("Story Points")).toBeInTheDocument();
   expect(screen.getByPlaceholderText("Hours")).toBeInTheDocument();
   expect(screen.getByText("Sprint")).toBeInTheDocument();
   expect(screen.getByText("Add")).toBeInTheDocument();

   // Wait for modules to be loaded
   await waitFor(() => {
     expect(screen.getByText("1 - Sprint 1")).toBeInTheDocument();
   });
 });
```

Mock modules

JavaScript

```
// Mocking example.
jest.mock(
```

```
  "./ReportContent/ReportKPI/ReportKPIHours/ReportKPIHours",
  () =>
    ({ KPIHoursData }) =>
      (
        <div data-testid="kpi-hours">
          <div>Estimated Hours: {KPIHoursData.stimated_hours}</div>
          <div>Worked Hours: {KPIHoursData.worked_hours}</div>
        </div>
      )
);
```

Snapshots

JavaScript

```
test("Dashboard component matches snapshot", async () => {
   const { container } = render(<Dashboard />);

   // Wait for loading to complete
   await waitFor(() => {
     expect(screen.queryByRole("progressbar")).not.toBeInTheDocument();
   });

   // Take a snapshot of the rendered component
   expect(container).toMatchSnapshot();
 });
```

React Testing Library and User Event Testing Library

JavaScript

```
test("filters tasks by status (To Do and Completed)", async () => {
   render(<Dashboard />);

   await waitFor(() => {
     expect(screen.queryByRole("progressbar")).not.toBeInTheDocument();
   });

   // Check if both tables are displayed
   expect(screen.getByText("To Do")).toBeInTheDocument();
```

```
    expect(screen.getByText("Completed")).toBeInTheDocument();

    // Check if tasks are in the correct tables
    expect(screen.getByText("Task 1")).toBeInTheDocument(); // To Do
    expect(screen.getByText("Task 2")).toBeInTheDocument(); // To Do
    expect(screen.getByText("Task 3")).toBeInTheDocument(); // Completed
  });
```

Avoid Implementation Details

JavaScript

```
//Examples where we avoid implementation details

const mockAddItem = jest.fn();
const mockToggleDone = jest.fn();
const mockDeleteItem = jest.fn();
```

Form Testing (user click or type events)

JavaScript

```
// Test form submission when isInserting is true
  test("does not submit form when isInserting is true", async () => {
    // Mock the addItem function
    const mockAddItem = jest.fn();

    const user = userEvent.setup();

    // Render the component with isInserting set to true
    render(
      <DashboardInput
        addItem={mockAddItem}
        isInserting={true}
        employeesList={mockEmployees}
      />
    );

    // Wait for modules to be loaded
    await waitFor(() => {
      expect(screen.getByText("1 - Sprint 1")).toBeInTheDocument();
```

```
    });

    // Fill in the form
    await user.type(screen.getByPlaceholderText("Title"), "New Task");
    await user.type(screen.getByPlaceholderText("Description"), "Description
for New Task");
    await
user.selectOptions(screen.getByText("Responsible").closest("select"), "1");
    await user.type(screen.getByPlaceholderText("Story Points"), "5");
    await user.type(screen.getByPlaceholderText("Hours"), "10");
    await user.selectOptions(screen.getByText("Sprint").closest("select"),
"1");

    // Submit the form
    await user.click(screen.getByText("Add"));

    // Check if addItem was not called
    expect(mockAddItem).not.toHaveBeenCalled();
  });
```

Dynamic Test Data

JavaScript

```
function generateMockTasks(count = 10, options = {}) {
  // Set up seed for deterministic results if needed
  if (options.deterministic) {
    faker.seed(options.seed || 456);
  }

  const doneRatio = options.doneRatio || 0.3;
  const modules = options.modules || [1, 2];
  const responsibles = options.responsibles || [1, 2, 3, 4, 5];

  return Array(count)
    .fill()
    .map((_, index) => {
      const id = index + 1;
      const isDone = faker.datatype.boolean({ probability: doneRatio }) ? 1
: 0;

      return {
        id,
        title: faker.word.words({ count: { min: 2, max: 5 } }),
```

```
        description: faker.lorem.sentence(),
        estimatedTime: faker.number.int({ min: 1, max: 12 }),
        done: isDone,
        story_Points: faker.number.int({ min: 1, max: 8 }),
        moduleId: faker.helpers.arrayElement(modules),
        responsible: faker.helpers.arrayElement(responsibles),
        actualTime: isDone ? faker.number.int({ min: 1, max: 15 }) : 0,
      };
    });
}

/**
 * Generates dynamic mock module data
 */
function generateMockModules(count = 3, options = {}) {
  // Set up seed for deterministic results if needed
  if (options.deterministic) {
    faker.seed(options.seed || 789);
  }

  return Array(count)
    .fill()
    .map((_, index) => {
      const id = index + 1;
      return {
        id,
        title: `Sprint ${id}`,
      };
    });
}

// Create static test tasks that we can reliably reference in tests
const staticTestTasks = [
  {
    id: 999,
    title: "Static Test Task 1",
    description: "This task has predictable values for testing",
    estimatedTime: 5,
    done: 0,
    story_Points: 3,
    moduleId: 1,
    responsible: 1,
    actualTime: 0,
  },
  {
    id: 1000,
    title: "Static Test Task 2",
    description: "This is another predictable task for testing",
```

```
    estimatedTime: 8,
    done: 0,
    story_Points: 5,
    moduleId: 2,
    responsible: 2,
    actualTime: 0,
  },
];

// Generate mock data with deterministic seeds for reproducible tests
const mockEmployees = generateMockEmployees(5, {
  deterministic: true,
  seed: 123,
});
const mockModules = generateMockModules(4, { deterministic: true, seed: 789
});

// Get the IDs from the generated employees to use as responsibles
const employeeIds = mockEmployees.map((employee) => employee.id);

// Generate mock tasks with the employees and modules
const dynamicMockTasks = generateMockTasks(15, {
  deterministic: true,
  seed: 456,
  doneRatio: 0.3,
  modules: mockModules.map((module) => module.id),
  responsibles: employeeIds,
});

// Combine dynamic tasks with static test tasks
const mockTasks = [...dynamicMockTasks, ...staticTestTasks];
```

Mocking HTTP Requests with Mock Server Worker

JavaScript

```
// Setup MSW server for mocking HTTP requests
const server = setupServer(
  // Mock GET request for fetching tasks
  rest.get(API_LIST, (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(mockTasks));
  }),

  // Mock GET request for fetching modules
```

```
  rest.get(API_MODULES, (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(mockModules));
  }),

  // Mock PUT request for updating a task - dynamic version
  rest.put(`${API_LIST}/:id`, (req, res, ctx) => {
    const { id } = req.params;
    const taskIndex = mockTasks.findIndex((task) => task.id.toString() ===
id);

    if (taskIndex === -1) {
      return res(ctx.status(404));
    }

    const task = { ...mockTasks[taskIndex], ...req.body };
    return res(ctx.status(200), ctx.json(task));
  }),
```

Estructura de Pruebas

Hemos implementado pruebas para los siguientes componentes:

1. **Report y ReportKPI** - Para verificar la generación de reportes KPI
2. **DashboardTasksTable** - Para probar la visualización y manipulación de tareas
3. **DashboardInput** - Para validar el formulario de ingreso de nuevas tareas
4. **DashboardRealHours** - Para verificar el registro de horas reales de trabajo
5. **API Module** - Para verificar la comunicación con el backend

Configuración del Entorno de Pruebas

Para todas las pruebas, utilizamos MSW (Mock Service Worker) para simular respuestas del servidor, lo que nos permite probar componentes que realizan llamadas a API sin depender de un backend real.

JavaScript
```
// Ejemplo de configuración de MSW
const server = setupServer(
  rest.get(API_TEAM_DATA, (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(mockTeamData));
  }),
  rest.get(API_MODULES, (req, res, ctx) => {
```

```
    return res(ctx.status(200), ctx.json(mockModuleData));
  })
);

beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());
afterAll(() => server.close());
```

También creamos mocks para los componentes de visualización de gráficos (Recharts) para evitar problemas con ResizeObserver en el entorno de pruebas:

JavaScript
```
jest.mock("recharts", () => {
  const OriginalModule = jest.requireActual("recharts");
  return {
    ...OriginalModule,
    ResponsiveContainer: ({ children, width, height }) => (
      <div data-testid="responsive-container" style={{ width, height }}>
        {children}
      </div>
    ),
    // Otros componentes mockeados...
  };
});
```

Detalle de Pruebas por Componente

1. Report y ReportKPI

Estas pruebas se enfocan en verificar la funcionalidad de generación de reportes KPI para equipos y miembros individuales filtrados por sprint.

Casos de prueba:
Generación de reportes KPI por equipo y sprint

Verifica que al seleccionar un equipo y un sprint, se muestren correctamente:

- Tareas pendientes y completadas
- Horas estimadas y trabajadas
- Tareas por hora

Generación de reportes KPI por persona y sprint

Verifica que al seleccionar un miembro específico del equipo y un sprint, se muestren correctamente:

- Tareas pendientes y completadas para ese miembro
- Horas estimadas y trabajadas para ese miembro
- Tareas por hora para ese miembro

- **Manejo correcto de errores de API**
  Verifica que se muestre un mensaje de error cuando la API responde con un código de estado 500.

2. DashboardTasksTable

Estas pruebas verifican la visualización y manipulación de tareas en la tabla del dashboard.

Casos de prueba:

- **Renderizado de la tabla con datos correctos**
  Verifica la presencia de encabezados, datos de tareas y acciones disponibles.

- **Filtrado de tareas por módulo**
  Verifica que solo se muestren las tareas del módulo/sprint seleccionado.

- **Filtrado de tareas por estado**
  Verifica que se muestren las tareas según su estado (pendientes o completadas).

- **Visibilidad de la tabla**
  Verifica que la tabla pueda ocultarse y mostrarse al hacer clic en el botón correspondiente.

- **Funcionalidad de marcar como completada**
  Verifica que al hacer clic en el botón "Done", se llame a la función para registrar horas reales.

- **Funcionalidad de eliminación de tareas**
  Verifica que al hacer clic en el botón de eliminar, se llame a la función correspondiente con el ID correcto.

- **Funcionalidad de mostrar tareas completadas por Sprint**
  Verifica que se muestren las tareas y su información minima

3. DashboardInput

Estas pruebas verifican el formulario de ingreso de nuevas tareas.

Casos de prueba:

- **Renderizado del componente con elementos correctos**
  Verifica la presencia de todos los campos del formulario.

- **Envío del formulario con datos válidos**
  Verifica que al completar todos los campos y hacer clic en "Add", se llame a la función addItem con los datos correctos.

- **Validación de datos del formulario**
  Verifica que no se pueda enviar el formulario con campos vacíos.

- **Estado de inserción**
  Verifica que cuando isInserting es true, no se pueda enviar el formulario.

4. DashboardRealHours

Estas pruebas verifican el componente para registrar las horas reales trabajadas en una tarea.

Casos de prueba:

- **Renderizado del componente con elementos correctos**
  Verifica la presencia del título, descripción, campo de entrada y botones.

- **Actualización del valor de entrada**
  Verifica que el valor del campo se actualice cuando el usuario escribe.

- **Funcionalidad del botón Cancelar**
  Verifica que al hacer clic en "Cancel", se llame a la función isHidden con true.

- **Funcionalidad del botón Guardar**
  Verifica que al hacer clic en "Save", se llame a la función confirm_Real_Hours con el valor correcto.

- **Funcionalidad de la tecla Enter**
  Verifica que al presionar Enter, se llame a la función confirm_Real_Hours con el valor correcto.

- **Estado inicial**
  Verifica que el valor inicial del campo sea "0".

5. API Module

Estas pruebas verifican la comunicación con el backend a través del módulo API.

Casos de prueba:

- **Constantes API definidas correctamente**
  Verifica que las URLs y headers estén definidos correctamente.

- **Obtención de tareas**
  Verifica que se puedan obtener todas las tareas correctamente.

- **Obtención de una tarea específica**
  Verifica que se pueda obtener una tarea por su ID.

- **Creación de tareas**
  Verifica que se pueda crear una nueva tarea.

- **Actualización de tareas**
  Verifica que se pueda actualizar una tarea existente.

- **Eliminación de tareas**
  Verifica que se pueda eliminar una tarea existente.

- **Obtención de empleados**
  Verifica que se puedan obtener todos los empleados correctamente.

- **Obtención de módulos**
  Verifica que se puedan obtener todos los módulos correctamente.

- **Obtención de datos de equipo**
  Verifica que se puedan obtener los datos de equipo correctamente.

- **Manejo de errores**
  Verifica que se manejen correctamente los errores de la API.

Conclusiones

Las pruebas implementadas cubren los aspectos principales de la aplicación, verificando:

1. La correcta generación de reportes KPI por equipo y miembro
2. La visualización y manipulación de tareas
3. La entrada de datos para nuevas tareas
4. El registro de horas trabajadas
5. La comunicación con el backend

Este enfoque de pruebas nos permite detectar problemas temprano en el ciclo de desarrollo y garantizar que la aplicación se comporte como se espera ante diferentes situaciones.