Prueba técnica Tecnocom: Tienda React con Fake Store Api

1.Descripcion general del proyecto.

Este proyecto es una aplicación web desarrollada principalmente con React, en ella se muestra un catalogó de productos falsos obtenidos e implementados mediante la API publica “Fake Store API”. La aplicación permite a los usuarios ingresar mediante un sistema de login que soporta persistencia de sesión, también Incluye funcionalidades de filtrado y ordenamiento, paginación con scroll infinito, y soporte para temas (modo claro/oscuro). Utiliza React Router para la navegación y Styled Components para el estilo.

2. Implementación de Login.

El login fue implementado de manera simulada, sin un back-end real de autenticación. Esto permite evaluar la persistencia del usuario y la navegación protegida dentro de la aplicación, sin complicaciones externas.

Estructura general del login

- Se creó una página de login *(LoginPage.jsx)* con un formulario que solicita el nombre de usuario.
- Al enviar el formulario, se despacha una acción de Redux que almacena el usuario en el estado global *(slice auth).*
- Una vez autenticado, el usuario es redirigido al catálogo de productos.

Persistencia del login

- Para mantener al usuario autenticado incluso tras recargar la página, se utilizó *localStorage*.
- En la carga inicial, el estado del usuario se inicializa desde *localStorage* si existe.
- Si el usuario hace clic en “Cerrar sesión”, se elimina la información tanto del Redux store como del *localStorage*.

Protección de rutas

- La ruta del catálogo está protegida: si no hay un usuario autenticado, la aplicación redirige automáticamente al login usando *useNavigate* de *react-router-dom*.

Tecnologías involucradas:

- *react-router-dom*: navegación y protección de rutas.

- *Redux Toolkit*: para almacenar el usuario autenticado.
- *localStorage*: para persistencia de sesión.

3. Integración de API Fake Store.

La aplicación utiliza la Fake Store API para obtener la información de los productos a tiempo real, simulando un back-end de productos sin necesidad de una base de datos. ("https://fakestoreapi.com")

Integración en el front-end

La integración se realizó desde el componente *ProductsPage.jsx*, apoyada por Redux Toolkit con un slice de productos (*productsSlice.js*), que maneja el estado, carga y errores de la API.

Integración:

1. **Thunk asíncrono con Redux Toolkit:** Se creó una función *fetchProducts* con *createAsyncThunk* que realiza una llamada *axios.get("https://fakestoreapi.com/products").*

```
export const fetchProducts = createAsyncThunk(
  "products/fetchProducts",
  async () => {
    const res = await axios.get("https://fakestoreapi.com/products");
    return res.data;
  }
);
```

2. **Estado de productos en Redux:** El slice *productsSlice* maneja el estado *list*, *status* y *error*, cambiando según el estado de la petición (*pending*, *fulfilled*, *rejected*).

3. **Uso en el componente principal:** En *ProductsPage*, al cargar el componente, se verifica si el estado es *idle* y se dispara *dispatch(fetchProducts()).*

```
useEffect(() => {
  if (status === "idle") {
    dispatch(fetchProducts());
  }
}, [dispatch, status]);
```

4. **Categorías sin Redux:** Para simplificar, las categorías se obtienen directamente con *axios.get(...)* dentro de un *useEffect* y se guardan en un *useState*.

Optimización de datos

- Se usó *useMemo()* para aplicar filtros y ordenamientos a los productos sin recalcular todo innecesariamente.
- El filtrado y ordenado se aplica antes del paginado por *visibleCount*.

Manejo de errores:

- Si la carga falla, se muestra un mensaje de error en pantalla y se ofrece un botón para reintentar.

4. Pruebas unitarias.

Se realizaron pruebas unitarias automatizadas utilizando Jest como framework de testing y React Testing Library para realizar pruebas sobre los componentes React y así garantizar la funcionalidad de los componentes claves de este proyecto.

ProductsPage.test.jsx: catálogo de productos

Prueba	Descripción	Resultado
muestra mensaje de carga mientras se obtienen productos	Verifica que se muestre el mensaje "Cargando productos..." cuando los datos aún no se han cargado.	✅ Exitosa
muestra productos cuando la carga fue exitosa	Asegura que los productos rendericen correctamente cuando la API responde con éxito.	✅ Exitosa
muestra mensaje de error si falla la carga	Comprueba que se muestre un mensaje de error cuando falla la carga de productos.	✅ Exitosa

Mocking

Para simular estados como "loading", "success" o "failed", se utiliza *redux-mock-store* y *configureStore* para pasar estados controlados al store.

```
PASS src/pages/ProductsPage.test.jsx
  √ muestra mensaje de carga mientras se obtienen productos (126 ms)
  √ muestra productos cuando la carga fue exitosa (45 ms)
  √ muestra mensaje de error si falla la carga (10 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.301 s
Ran all test suites matching /ProductsPage/i.

Active Filters: filename /ProductsPage/
```

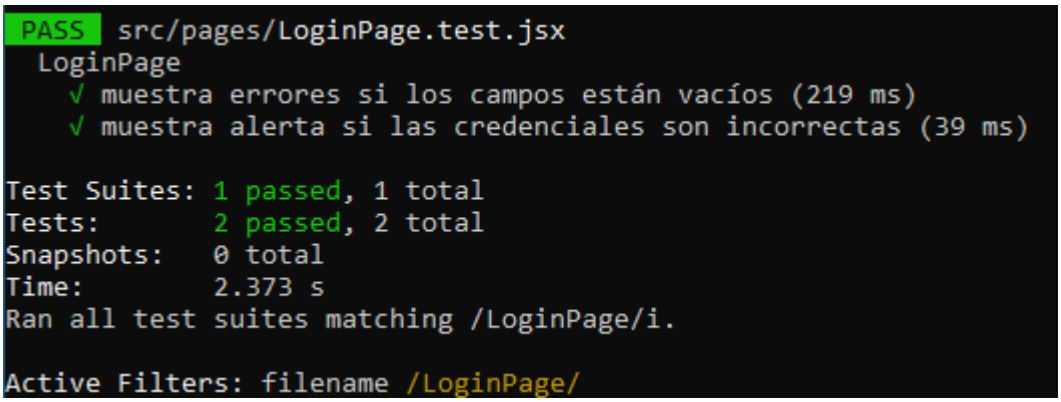
LoginPage.test.jsx: formulario de inicio de sesión.

Prueba	Descripción	Resultado
muestra campos de usuario y contraseña	Asegura que los campos de entrada se rendericen correctamente.	✅ Exitosa

muestra error si el usuario intenta enviar vacío	Verifica que aparezcan mensajes de validación cuando el formulario se envía vacío.	✅ Exitosa
envía el formulario correctamente	Simula el ingreso de datos válidos y verifica que la función de login sea llamada.	✅ Exitosa

Validación

- Se testea la validación de formularios implementada con react-hook-form.
- Las funciones como *mockLogin* o *mockNavigate* son usadas para comprobar que la lógica de login se activa correctamente.



Como correr pruebas unitarias

- Para correr todas las pruebas: npm test
- para correr un archivo en específico: npm *nombre del archivo*

5.Problemas durante el desarrollo.

Durante el desarrollo del proyecto surgieron diversos retos, tanto técnicos como de compatibilidad, especialmente al integrar pruebas automatizadas con las versiones más recientes de ciertas librerías.

Incompatibilidad entre *react-router-dom* v6+ y Testing Library / Jest

Problema:

Al usar *react-router-dom* versión 6.15.0 y superiores, se presentaron errores al correr pruebas unitarias, especialmente al simular navegación y renderizado de componentes con *<Routes>* y *<Navigate>.*

Síntomas:

- Las pruebas no detectaban correctamente rutas.
- El *MemoryRouter* fallaba al simular navegación.

- Algunos componentes no se renderizaban correctamente dentro de las pruebas.

Solución aplicada:

Se realizó un downgrade a la versión *react-router-dom@6.16.0*, la cual es totalmente compatible con Jest y Testing Library sin necesidad de mocks complejos ni wrappers adicionales.

Problemas con *styled-components* y selectores en las pruebas

Problema:

styled-components genera clases dinámicas que dificultan la selección de elementos durante las pruebas.

Síntomas:

- Los *getByText*, *getByRole*, etc., no encontraban elementos porque los estilos interferían.
- A veces el texto estaba dividido entre nodos o se encontraba encapsulado.

Solución aplicada:

- Se mejoró la accesibilidad de los componentes añadiendo *aria-label* a los elementos *<select>* en la vista de productos.
- Se usaron funciones personalizadas en *getByText((text) => text.includes(...))* para detectar texto de forma más flexible.
- Se agregaron encabezados *<h4>* a los títulos de productos para facilitar su identificación en los tests.

Error de renderizado al simular el estado del store

Problema:

Al realizar pruebas en *ProductsPage*, se generaba el error: *TypeError: products is not iterable*, causado por una estructura incorrecta del estado simulado.

Causa:

El componente esperaba una propiedad *list* en *state.products*, pero en algunos mocks se usaba incorrectamente *items* o faltaba el campo.

Solución aplicada:

Se estandarizó el estado simulado utilizando:

```
test("muestra mensaje de carga mientras se obtienen productos", () => {
  const store = mockStore({
    products: {
      list: [],
      status: "loading",
      error: null,
    },
    categories: {
      items: [],
      loading: false,
      error: null,
    },
    auth: {
      user: { name: "Test User" },
    },
  });

  renderWithProviders(store);
  expect(screen.getByText(/cargando productos/i)).toBeInTheDocument();
});
```

Y se aplicó una estructura coherente en todos los tests usando configureStore o mockStore.

Problemas con dependencias modernas y Jest (Ecosistema actual de Vite + React 18)

Problema:

Dependencias modernas como *axios@1.x*, *react@18*, *react-dom@18* y el uso de Vite como bundler generan incompatibilidades con Jest por su forma de compilar ESModules.

Síntomas:

- Importaciones fallaban.
- Se requería configuración avanzada de *jest.config.js*, *babel-jest*, *vite-node*, etc.

Solución aplicada:

- Se optó por usar solo dependencias compatibles con Jest directamente sin necesidad de transformar módulos.
- Se evitaron características como *React.lazy*, *Suspense*, o configuraciones de router asíncronas.
- Se utilizaron mocks y contextos controlados manualmente en las pruebas.

6.Guia para levantar el proyecto.

Requisitos previos

- Node.js *>=16.x*
  Verificar con: *node-v*
- Npm *>= 8.x*
  Verificar con: *npm -v*

1. Clonar el repositorio

- *git clone https://github.com/tu-usuario/tu-repo.git*
- *cd tu-repo*

2. Instalar dependencias

Este proyecto usa versiones específicas para compatibilidad con pruebas. Ejecuta:

npm install

Advertencia: Algunas dependencias fueron ajustadas a versiones anteriores para evitar conflictos con Jest. No actualices sin verificar compatibilidad.

3. Arrancar el proyecto

Se utilizó create-react-app para la creación y desarrollo de este proyecto

npm start

Abrirá automáticamente la dirección del servidor

4. Ejecutar pruebas unitarias

El proyecto incluye pruebas automatizadas con Jest y React Testing Library.

npm test

Este comando ejecutará las pruebas en los siguientes archivos:

- *src/pages/ProductsPage.test.jsx*
  - Verifica:
    - Mensaje de carga
    - Renderizado de productos
    - Manejo de errores
- *src/pages/LoginPage.test.jsx*
  - Verifica:
    - Validación de formularios
    - Mensajes de error
    - Interacción de login

5. Dependencias clave

- *react*, *react-dom*: Interfaz principal
- *react-router-dom@6.16.0*: Manejo de rutas
- *redux*, *@reduxjs/toolkit*: Manejo de estado global
- *styled-components*: Estilado con componentes
- *axios*: Llamadas HTTP
- *jest*, *@testing-library/react*: Pruebas unitarias

Notas finales

- El login es simulado (no hay back-end), por lo que cualquier usuario válido con una contraseña arbitraria será aceptado.
- Se estableció un usuario
    - Nombre de usuario: admin
    - Contraseña: 1234
- El catálogo de productos y categorías proviene directamente de la Fake Store API.
- Soporta **modo oscuro** con alternancia desde el botón del header.
- Se ha optimizado el rendimiento con *useMemo*, *useCallback*, y scroll infinito.