Instituto Politécnico Formosa

Trabajo Práctico Integrador III

Objetivo:

Desarrollar desde cero una aplicación frontend completa en React que consuma una API REST existente para gestionar autenticación de usuarios y un CRUD completo de tareas. El proyecto debe implementar todas las tecnologías vistas en clase: componentes funcionales, hooks (useState, useEffect, custom hooks), React Router DOM con rutas protegidas, formularios controlados, y estilos con Tailwind CSS/Bootstrap acompañado de archivos CSS.

Criterios de Evaluación

1. Presentación del código

- **Limpieza y organización**: Código limpio, ordenado y bien indentado.
- **Manejo de errores**: Uso obligatorio de **try-catch** en todas las peticiones asíncronas.
- **Estructura de proyecto**: Organización correcta en carpetas según lo establecido
- **Modularización**: Uso exclusivo de ESModules (import/export).
- **Funcionalidad**: El código debe ser funcional, sin errores de ejecución.
- **Nomenclatura**: Nombres descriptivos y consistentes para componentes, funciones y variables.

2. Componentes y Props

- **Componentización adecuada**: Separación correcta de responsabilidades.
- **Componentes reutilizables**: Navbar, Footer, Loading.
- **Uso correcto de props**: Paso de datos entre componentes padre-hijo.
- **Renderizado de listas**: Uso correcto de **.map()** con **key** única.
- **Composición de componentes**: Construcción de interfaces complejas mediante composición.

3. Hooks y Estado

- **useState**: Manejo correcto del estado local en componentes.
- **useEffect**: Implementación adecuada para efectos secundarios (fetch de datos, limpieza).
- **Custom Hooks**: Creación y uso de 1 custom hooks:
    - **useForm:** Manejo de formularios con validaciones.
- **Gestión de dependencias**: Arrays de dependencias correctos en useEffect.
- **Optimización**: Evitar re-renders innecesarios.

4. Rutas y Navegación

- **React Router DOM**: Configuración correcta de **BrowserRouter**, **Routes**, **Route**.
- **Rutas públicas**: Login, Register (sólo accesibles sin estar autenticado).
- **Rutas privadas**: Home, Tasks, Profile (solo accesibles estando autenticado).
- **Componente PrivateRoute**: Protección de rutas verificando autenticación.
- **Componente PublicRoute**: Redirección si el usuario ya está autenticado.
- **Navegación programática**: Uso de **useNavigate** para redirecciones.
- **Enlaces**: Uso correcto de **Link** o **NavLink** para navegación.

5. Integración con API Backend

- **Configuración de fetch**: Uso correcto de **credentials: 'include'** en todas las peticiones.
- **Endpoints implementados**:
    - **AUTH**: **/api/register, /api/login, /api/profile, /api/logout**
    - **TASKS**: **/api/tasks-by-user** (GET), **/api/tasks** (POST), **/api/tasks/:id** (PUT, DELETE)
- **Manejo de respuestas**: Procesamiento correcto de respuestas exitosas y errores.
- **Códigos HTTP**: Manejo apropiado de diferentes códigos (200, 201, 400, 401, 404, 500).
- **Loading states**: Indicadores de carga durante peticiones asíncronas.
- **Feedback al usuario**: Mensajes claros de éxito o error en operaciones.

Entrega mediante Git y GitHub – Uso de ramas y commits

Requisitos obligatorios para el control de versiones:

1. **Crear repositorio** llamado **"trabajo-practico-integrador-2-juan-perez"**.
2. **Estructura de ramas**:
    - Crear repositorio con README inicial en rama main.
    - Crear rama **develop** desde **main**.
3. **Durante el desarrollo** en rama **develop**:
    - Realizar **mínimo 10 commits** con mensajes claros y descriptivos.
4. **Al finalizar el trabajo**:
    - Hacer un **merge limpio** de **develop** hacia **main**, sin conflictos.
    - Ambas ramas **(develop y main)** deben quedar sincronizadas.

Consignas

1. Configuración inicial del proyecto

Crear la estructura base desde cero:

1. Inicializar proyecto con Vite:
    - Seleccionar: React → JavaScript + SWC
    - Navegar al directorio del proyecto
    - Instalar dependencias
2. Instalar dependencias necesarias:
    - react-router-dom
    - **tailwindcss** o **bootstrap**
3. Crear estructura de carpetas completa:

```
src/
  ├── components/
  │    └── Navbar
  │    ├── Footer
  │    └── Loading
  ├── hooks/
  │    └── useForm.js
  ├── pages/
  │    ├── Login
  │    ├── Register
  │    ├── Home.jsx
  │    ├── Tasks.jsx
  │    └── Profile.jsx
  ├── router/
  │    ├── AppRouter.jsx
  │    ├── PrivateRoute.jsx
  │    └── PublicRoute.jsx
  ├── App.jsx
  ├── main.jsx
  └── index.css
```

4. Configurar archivo **.gitignore**:
    - Incluir node_modules, dist, .env, .env.local

---

2. Implementación de Custom Hook

2.1. Custom Hook: useForm

Ubicación: src/hooks/useForm.js

Funcionalidades requeridas:

- Gestionar estado del formulario con múltiples campos
- Función para actualizar valores de inputs al escribir
- Función para resetear el formulario a valores iniciales
- Retornar el estado actual del formulario y las funciones de manejo

El hook debe ser lo suficientemente genérico para manejar diferentes formularios (login, register, tasks).

---

3. Implementación de Componentes Comunes

3.1. Navbar

Ubicación: src/components/Navbar.jsx

Requisitos:

- Mostrar logo o nombre de la aplicación
- Si el usuario está autenticado, mostrar enlaces a: Home, Tasks, Profile
- Si el usuario está autenticado, mostrar botón de Logout
- Si el usuario NO está autenticado, mostrar enlaces a: Login, Register
- Diseño responsive.

3.2. Loading

Ubicación: src/components/Loading.jsx

Requisitos:

- Componente reutilizable para mostrar estados de carga
- Puede ser un spinner, animación o mensaje "Cargando..."
- Estilos centrados y visibles

3.3. Footer

Ubicación: src/components/Footer.jsx

Requisitos:

- Footer con información básica: copyright, año actual, nombre del alumno
- Posicionado correctamente al final de la página

---

4. Implementación del Sistema de Rutas

4.1. AppRouter

Ubicación: src/router/AppRouter.jsx

Requisitos:

- Configurar BrowserRouter como componente raíz
- Incluir componente Navbar
- Incluir componente Footer
- Definir las siguientes rutas dentro de Routes:
    - **Rutas públicas** (envueltas en PublicRoute):
        - /login → Página Login
        - /register → Página Register
            - **Rutas por defecto**:
                - / → Redireccionar a /login
                - * → Redireccionar a /login
    - **Rutas privadas** (envueltas en PrivateRoute):
        - /home → Página Home
        - /tasks → Página Tasks
        - /profile → Página Profile
            - **Rutas por defecto**:
                - / → Redireccionar a /home
                - * → Redireccionar a /home

4.2. PrivateRoute

Ubicación: src/router/PrivateRoute.jsx

Requisitos:

- Proteger rutas que requieren autenticación
- Verificar si hay usuario autenticado consultando el endpoint /api/profile
- Si NO hay usuario, redireccionar a /login
- Si hay usuario, renderizar el componente hijo
- Mostrar Loading mientras se verifica la autenticación

4.3. PublicRoute

Ubicación: src/router/PublicRoute.jsx

Requisitos:

- Proteger rutas públicas para usuarios ya autenticados
- Verificar si hay usuario autenticado
- Si hay usuario, redireccionar a /home
- Si NO hay usuario, renderizar el componente hijo
- Mostrar Loading mientras se verifica la autenticación

---

5. Implementación de Páginas de Autenticación

5.1. Página Login

Ubicación: src/pages/auth/Login.jsx

Requisitos:

- Formulario con campos: username y password
- Utilizar custom hook useForm para manejar el estado del formulario
- Validar que los campos no estén vacíos antes de enviar
- Mostrar componente Loading durante la petición
- Al login exitoso, redireccionar a /home
- Mostrar mensajes de error si la autenticación falla
- Incluir enlace a la página de Register

5.2. Página Register

Ubicación: src/pages/auth/Register.jsx

Requisitos:

- Formulario con campos:
    - username
    - email
    - password
    - firstname
    - lastname
    - dni
- Utilizar custom hook useForm para manejar el estado del formulario
- Implementar validaciones básicas en los campos
- Mostrar componente Loading durante la petición
- Al registro exitoso, redireccionar a /home
- Mostrar mensajes de error si el registro falla
- Incluir enlace a la página de Login

5.3. Página Profile

Ubicación: src/pages/Profile.jsx

Requisitos:

- Mostrar información del usuario:
    - id
    - name
    - lastname
- Incluir botón de Logout que consulte al endpoint /api/logout
- Después del logout, redireccionar a /login
- Mostrar componente Loading mientras se cargan los datos

---

6. Implementación de Página Home

Ubicación: src/pages/Home.jsx

Requisitos:

- Página de bienvenida para usuarios autenticados
- Mostrar mensaje de bienvenida con el nombre del usuario
- Mostrar resumen estadístico:
    - Total de tareas
    - Tareas completadas
    - Tareas pendientes
- Incluir enlaces o botones para navegar a la página de Tasks
- Diseño atractivo con cards o tarjetas

---

7. Implementación de CRUD de Tareas

7.1. Página Tasks

Ubicación: src/pages/Tasks.jsx

Requisitos:

- Utilizar custom hook useForm para los formularios de creación y edición

Funcionalidades a implementar:

1. **Listar tareas**:
    - Obtener tareas del usuario desde /api/tasks-by-user
    - Renderizar lista de tareas usando .map()
    - Mostrar para cada tarea: título, descripción, estado (completada/pendiente), fecha
    - Mostrar componente Loading mientras se cargan las tareas
    - Si no hay tareas, mostrar mensaje apropiado
2. **Crear tarea**:
    - Formulario con campos: title, description, is_completed (checkbox)
    - Validar que los campos obligatorios no estén vacíos
    - Enviar petición POST a /api/tasks
    - Actualizar la lista de tareas después de crear
    - Mostrar mensaje de éxito
3. **Editar tarea**:
    - Pre-llenar formulario con datos de la tarea seleccionada
    - Permitir modificar: title, description, is_completed
    - Enviar petición PUT a /api/tasks/:id
    - Actualizar la lista de tareas después de editar
    - Mostrar mensaje de éxito
4. **Eliminar tarea**:
    - Mostrar confirmación antes de eliminar
    - Enviar petición DELETE a /api/tasks/:id
    - Actualizar la lista de tareas después de eliminar
    - Mostrar mensaje de éxito
5. **Marcar como completada**:
    - Checkbox o botón para cambiar el estado
    - Enviar petición PUT a /api/tasks/:id con is_completed actualizado
    - Actualizar visualmente el estado de la tarea

Diseño:

- Las tareas completadas deben tener un estilo visual diferente (texto tachado, color diferente, etc.)
- Botones claros para las acciones: Editar, Eliminar, Marcar como completada
- El formulario puede estar en la misma página o en un modal

Especificaciones Técnicas Adicionales

Configuración de Fetch con Credentials

Todas las peticiones a la API deben incluir credentials: 'include' para enviar y recibir cookies de sesión.

Estructura básica de configuración:

- Método: GET, POST, PUT o DELETE según corresponda
- Headers: Content-Type: application/json para peticiones con body
- Credentials: include (obligatorio)
- Body: JSON.stringify(data) solo en POST y PUT

Manejo de Errores

Requisitos:

- Usar bloques try-catch en todas las peticiones asíncronas
- Verificar el código de estado de la respuesta
- Mostrar mensajes de error claros al usuario
- Manejar diferentes tipos de errores (validación, autenticación, servidor)

Loading States

Implementar indicadores de carga en:

- Verificación de autenticación en rutas protegidas
- Peticiones de login y register
- Carga inicial de datos (perfil, tareas)
- Operaciones CRUD de tareas
- Deshabilitar botones durante peticiones

Estilos y Diseño

Requisitos:

- Aplicación completamente responsive (mobile-first)
- Elección libre entre:
    - Tailwind CSS
    - Bootstrap
- Diseño limpio, moderno y profesional
- Feedback visual para interacciones:

    - Estados hover en botones y enlaces
    - Estados focus en inputs
    - Estados disabled visibles
- Mensajes de error en color destacado
- Mensajes de éxito en color positivo
- Loading spinners visibles
- Paleta de colores consistente
- Tipografía legible

Componentes que deben tener estilos:

- Navbar responsive con navegación clara
- Formularios con inputs bien diseñados
- Botones con diferentes estilos según acción
- Lista de tareas con cards o diseño atractivo
- Footer posicionado correctamente

Estructura del README.md

El archivo README.md debe incluir:

1. **Título del proyecto**
2. **Descripción breve**
3. **Instrucciones de instalación**
4. **Configuración del archivo .env**

Aclaraciones Finales

- El backend ya está desarrollado y funcionando. El estudiante solo debe consumir los endpoints proporcionados.
- No se permite el uso de librerías de estado global (Context API, Redux, Zustand) ya que no fueron vistas en clase.
- Todo el manejo del estado se realiza mediante custom hooks, useState y props.
- La aplicación debe funcionar correctamente conectada al backend proporcionado.
- Se valorará la creatividad en el diseño, siempre manteniendo usabilidad.
- El código debe ser original del estudiante.
- Cualquier duda debe consultarse al docente antes de la fecha de entrega.
- La aplicación debe ser completamente funcional sin errores en consola.