Documentación de la Aplicación de Roles y Usuarios

Esta documentación describe la estructura, flujo de datos y navegación de los módulos de **Roles** y **Usuarios**. Incluye diagramas, endpoints utilizados por cada componente y análisis de posibles fallos lógicos.

Diagrama de Flujo de Navegación y Datos 🗺

A continuación se muestra el flujo de navegación entre componentes y las interacciones con el servidor API.

Ver diagrama completo de navegación y flujo de datos

Endpoints utilizados por cada componente

Cada componente interactúa con el servidor a través de llamadas HTTP definidas en `hacerPeticionApi`.

- **Roles** (Roles.tsx)
- GET `/api/roles`
- **RolesAgregar** (RolesAgregar.tsx)
- POST `/api/roles`

Body: `{ rolNombre: string }`

- **RolesEditar** (RolesEditar.tsx)
- PATCH `/api/roles`

Body: `{ rolId: number, rolNombre: string, editar: true }`

- DELETE `/api/roles`

Body: `{ rolId: number }`

- **RolesPermisoPrimario / Secundario / Terciario**
- PATCH `/api/roles`

Body: `{ rolId: number, rolNombre: string, rutas: string[], nivel: number }`

- **Usuarios** (Usuarios.tsx)
- GET `/api/usuarios`
- GET `/api/roles`
- **UsuariosAgregar** (UsuariosAgregar.tsx)
- POST `/api/usuarios`

Body: `{ username, nombre, email?, password }`

- **UsuariosEditar** (UsuariosEditar.tsx)
- PATCH `/api/usuarios`

Body options:

- Datos: `{ usuarioId, usernameNuevo?, nombreCompletoNuevo?, emailNuevo? }`
- Contraseña: `{ usuarioId, passwordActual, passwordNuevo }`
- **UsuariosPermiso** (UsuariosPermiso.tsx)
- PUT `/api/usuarios/{id}/roles`

Body: `{ id, rolId, rolNombre, suscribir: boolean, username }`

- **UsuariosTabla** (UsuariosTabla.tsx)
- PATCH `/api/usuarios`

Body: `{ habilitar: boolean, username, usuarioId }`

Componentes

RolesAgregar.tsx

Propósito: Componente para crear un **nuevo rol** en el sistema.

- Recibe:
- `actualizarDatosRoles`: función para refrescar lista de roles.
- `setMostrarAgregarRol`: controla visibilidad del formulario.
- Estado local:
- `nombre`: nombre del rol.
- `guardando`: indicador de proceso.
- Validaciones:
- Longitud mínima de 5 caracteres.
- Solo caracteres alfanuméricos y los símbolos: `& - _ /`.
- Flujo:
- Se presiona **Aceptar** o tecla Enter.
- Valida `nombre`.
- Llama a `hacerPeticionApi({ method: 'POST', body })`.
- Actualiza lista de roles y cierra formulario.

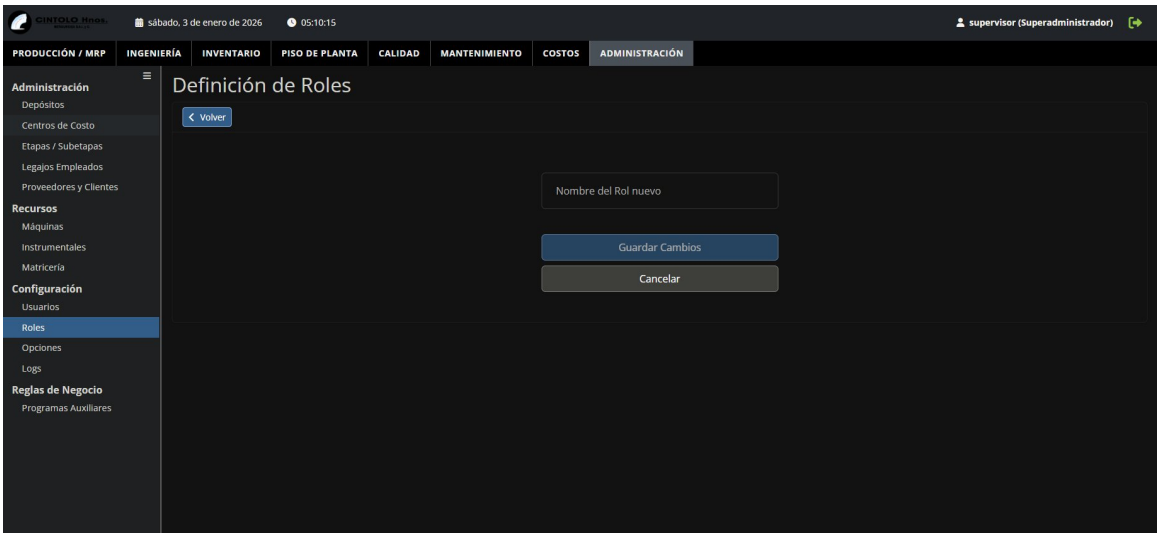

Figura 1 – Formulario de alta de un nuevo Rol con carga de nombre de Rol nuevo.

tsx

```
const RolesAgregar: FC<Props> = ({ actualizarDatosRoles, setMostrarAgregarRol }) => {
  const [nombre, setNombre] = useState("");
  const esValido = useMemo(() => nombre.length > 4, [nombre]);

  const crearRolHandler = async () => {
    if (!esValido) return;
    const nombreRol = nombre.trim();
    if (!/^[a-zA-Z0-9 /-_&]+$/.test(nombreRol)) {
      dispatch(abrirModalReducer({ /* error caracteres */ }));
      return;
    }
    setGuardando(true);
    const respuesta = await hacerPeticionApi({
      method: 'POST',
      body: { rolNombre: nombreRol }
    });
    setGuardando(false);
    if (!respuesta.datos?.datosRoles) {
      mostrarError(respuesta);
      return;
    }
    actualizarDatosRoles(respuesta.datos.datosRoles);
    setMostrarAgregarRol(false);
  };

  useTeclaEnter(crearRolHandler);

  return (
    <div className='container'>
      {/* input y botones */}
    </div>
  );
};
```

RolesEditar.tsx

Propósito: Editar o eliminar el **nombre** de un rol existente.

- Propiedades:

- `rol`: datos del rol a editar.
- `setIdMostrarEditar`: oculta vista edición.
- Funciones:
- `editarRol`: valida y PATCH del nombre.
- `eliminarRol`: muestra confirmación y DELETE.
- Reglas:
- Nombre mínimo 5 caracteres.
- Mismos caracteres permitidos que en RolesAgregar.

tsx

```
const RolesEditar: FC<Props> = ({ actualizarDatosRoles, rol, setIdMostrarEditar }) => {
  const [nombre, setNombre] = useState(rol.nombre);
  const esValido = useMemo(() => nombre.length > 4, [nombre]);


  const editarRol = async () => { /* PATCH */ };
  const eliminarRol = () => { /* abrirModalReducer DELETE */ };


  useTeclaEnter(editarRol);


  return (
    <tr>
      <td colSpan={4}>
        {/* formulario y botones AceptarCancelarBtn */}
      </td>
    </tr>
  );
};
```

RolesTabla.tsx

Propósito: Mostrar la **lista de roles** y permitir acciones de edición o permisos.

- Recibe:
- `datosRoles`: array de roles o `null`.
- `actualizarDatosRoles`: refrescar datos.
- `tablaRef`: referencia para exportar/imprimir.
- Estado local:
- `idMostrarEditar`: rol en modo edición.
- `idMostrarPermisos`: rol en modo permisos.
- Renderiza:
- Fila principal con botones **Editar** y **Permisos**.
- Componentes hijos: `RolesPermisoPrimario` y `RolesEditar`.
- Si no hay roles, `NoHayElementos`.

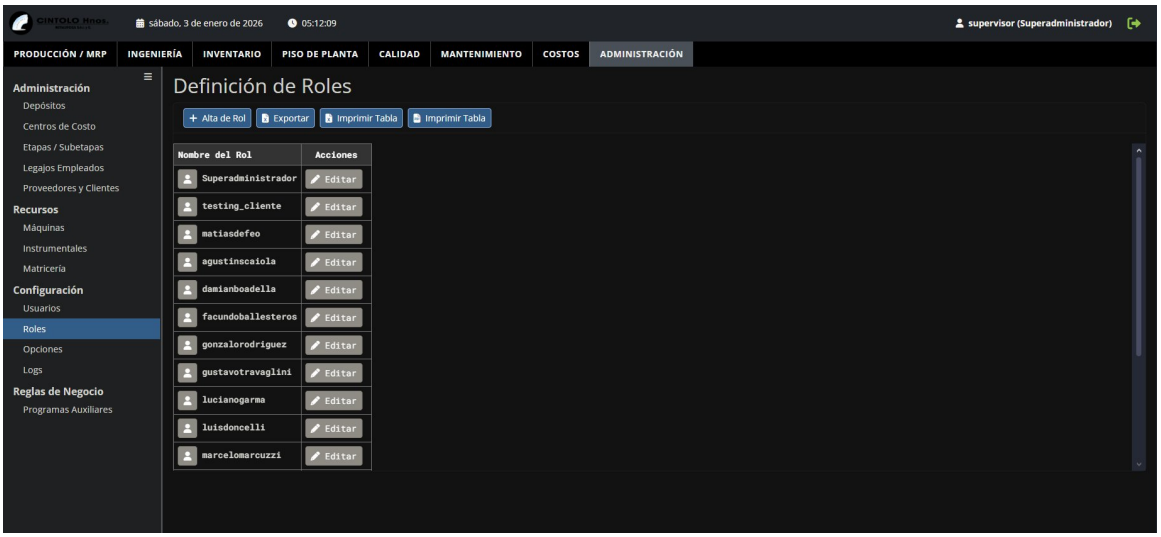

Figura 1 – Pantalla principal de Roles con listado y acciones de alta, habilitado, editar roles, exportación e impresión.

tsx

```
const RolesTabla: FC<Props> = ({ actualizarDatosRoles, datosRoles, tablaRef }) => {
  const [idMostrarEditar, setIdMostrarEditar] = useState(0);
  const [idMostrarPermisos, setIdMostrarPermisos] = useState(0);

  return (
    <div className='div-tabla'>
      <table ref={tablaRef}>
        <thead>{/* encabezado */}</thead>
        <tbody>
          {datosRoles?.map(r => (
            <Fragment key={r.nombre}>
              {/* fila principal */}
              {idMostrarPermisos === r.id && traerRutas().map(ruta => (
                <RolesPermisoPrimario ... />
              ))}
              {idMostrarEditar === r.id && (
                <RolesEditar .../>
              )}
            </Fragment>
          )) || <NoHayElementos />}
        </tbody>
      </table>
    </div>
  );
};
```

RolesPermisoPrimario.tsx

Propósito: Gestionar permisos de nivel **primario** para cada módulo.

- Uso de hook `useTraerUsuario` para verificar nivel de usuario.
- Estado `permiso`: nivel actual (0–4).
- Al cambiar nivel:
- Si `nivel !== 4`, se actualizan todas las rutas de la sección.
- Llama a `hacerPeticionApi({ method: 'PATCH', body: { rutas, nivel } })`.

- Renderiza botones con `SeleccionGrupo`.
- Si `permiso === 4`, muestra subsecciones con `RolesPermisoSecundario`.

tsx

```
const RolesPermisoPrimario: FC<Props> = ({ actualizarDatosRoles, rol, ruta }) => {
  const [permiso, setPermiso] = useState(/* nivel inicial */);

  const cambiarAccesoDeRolHandler = async (nivel: number) => {
    if (usuario.nivel < 3) return;
    setPermiso(nivel);
    const rutasEditar = [`/${ruta.path}`];
    if (nivel !== 4) {
      ruta.subsecciones.forEach(ss =>
        ss.items.forEach(i => {
          rutasEditar.push(`/${ruta.path}/${i.link}`);
          i.subitems?.forEach(si =>
            rutasEditar.push(`/${ruta.path}/${i.link}/${si.path}`)
          );
        })
      );
    }
    const respuesta = await hacerPeticionApi({
      method: 'PATCH',
      body: { rolId: rol.id, rolNombre: rol.nombre, rutas: rutasEditar, nivel }
    });
    /* manejar respuesta */
  };

  return (
    <>
      <tr>{/* selección primaria */}</tr>
      {permiso === 4 && ruta.subsecciones.map(subsec =>
        subsec.items.map(item => (
          <RolesPermisoSecundario .../>
        ))
      )}
    </>
  );
};
```

RolesPermisoSecundario.tsx

Propósito: Gestionar permisos de nivel **secundario** y desplegar nivel terciario.

- Estado `permiso`: si el padre es personalizado (4), se ajusta a hijo o padre.
- Al cambiar nivel, PATCH de una ruta única.
- Renderiza:
- Botones de permiso con `SeleccionGrupo`.
- Lista de `subitems` con `RolesPermisoTerciario`.

tsx

```
const RolesPermisoSecundario: FC<Props> = ({ item, permisoPadre, permisoHijo, rol, ruta, ... })
=> {
  const [permiso, setPermiso] = useState(
    permisoPadre === 4 ? permisoHijo : permisoPadre
  );

  const cambiarAccesoDeRolHandler = async (nivel: number) => {
    setPermiso(nivel);
```

```
    const rutas = [`/${ruta.path}/${item.link}`];
    /* PATCH */
  };

  return (
    <>
      <tr>{/* fila secundaria */}</tr>
      {item.subitems?.map(subitem => (
        <RolesPermisoTerciario .../>
      ))}
    </>
  );
};
```

RolesPermisoTerciario.tsx

Propósito: Ajustar permisos **detallados** sobre subrutas.

- Estado `permiso`: nivel actual.
- Al cambiar nivel, PATCH de una ruta concreta.
- Renderiza una fila con ruta completa y `SeleccionGrupo`.

tsx

```
const RolesPermisoTerciario: FC<Props> = ({ subitem, permisoPadre, permisoHijo, rol,
ruta, ... }) => {
  const [permiso, setPermiso] = useState(
    permisoPadre === 4 ? permisoHijo : permisoPadre
  );

  const cambiarAccesoDeRolHandler = async (nivel: number) => {
    setPermiso(nivel);
    const rutas = [`/${ruta.path}/${item.link}/${subitem.path}`];
    /* PATCH */
  };

  return (
    <tr>
      <td>/{ruta.path}/{item.link}/{subitem.path}</td>
      <td colSpan={2}>
        {/* SeleccionGrupo */}
      </td>
    </tr>
  );
};
```

SeleccionGrupo.tsx

Propósito: Renderizar un grupo de botones de tipo **radio** para seleccionar un nivel de permiso.

- Recibe:
- `callback`: función que se ejecuta al cambiar.
- `botones`: array con propiedades de cada opción.
- Genera un `<input type="radio">` y `<label>` para cada botón.
- Garantiza estilo y accesibilidad.

tsx

```
const SeleccionGrupo = (callback: Function, botones: ButtonConfig[]) => (
  <div role="group" className='btn-group'>
    {botones.map(boton => (
      <Fragment key={boton.id}>
        <input
          type="radio"
          name={boton.nombre}
          id={boton.id}
          checked={boton.checked}
          disabled={boton.disabled}
          onChange={() => callback(boton.valor)}
        />
        <label htmlFor={boton.id}>{boton.texto}</label>
      </Fragment>
    ))}
  </div>
);
```

UsuariosAgregar.tsx

Propósito: Formulario para **crear** un nuevo usuario.

- Estado local: `nombre`, `email`, `username`, `password`, `guardando`.
- Validaciones:
- `username`: ≥4 caracteres, sin espacios, solo alfanumérico.
- `email`: formato válido si no está vacío.
- `password`: ≥8 caracteres.
- Al crear, POST `/api/usuarios` y añade al listado.

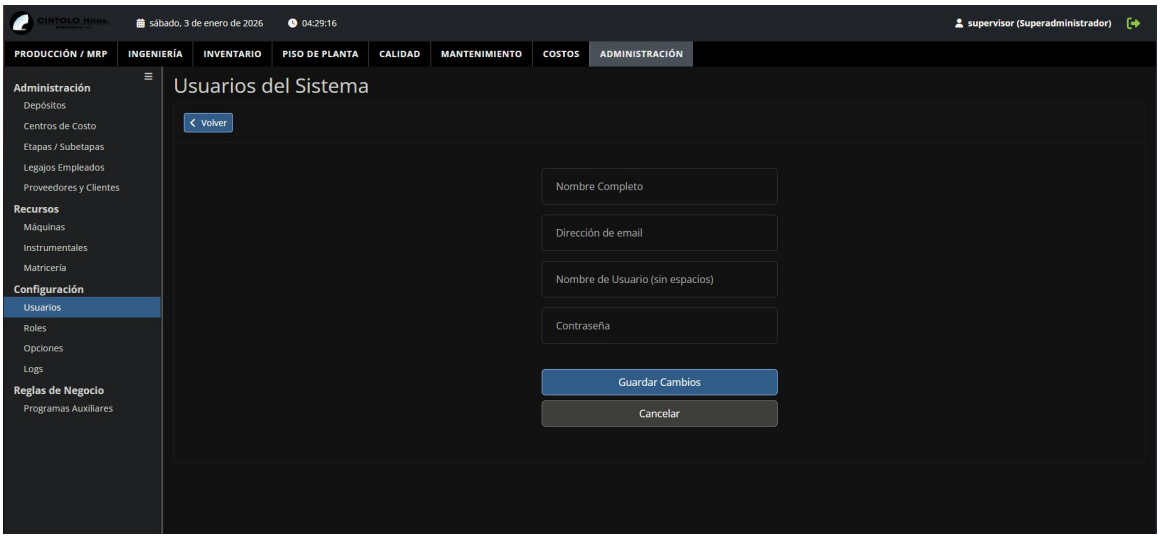

Figura x – Formulario de alta de un nuevo Usuario del Sistema con carga de Nombre completo, email, nombre de usuario y contraseña.

tsx

```
const UsuariosAgregar: FC<Props> = ({ actualizarDatosUsuarios, setMostrarAgregarUsuario }) => {
  const crearUsuarioHandler = async () => { /* validaciones y POST */ };
  useTeclaEnter(crearUsuarioHandler);
  return (
    <div>
      {/* campos y AceptarCancelarBtn */}
    </div>
  );
```

```
};
```

UsuariosEditar.tsx

Propósito: Editar datos o contraseña de un usuario existente.

- Secciona en dos bloques:
- **Datos**: username, nombre completo, email.
- Valida sin espacios y expresiones regulares para email.
- PATCH `/api/usuarios`.
- **Contraseña**: actual y nueva.
- Longitud mínima de 8.
- PATCH `/api/usuarios`.

tsx

```
const UsuariosEditar: FC<Props> = ({ datosUsuario, setEdicionUsuarioId,
actualizarDatosUsuarios }) => {
  const cambiarDatos = () => { /* abrir modal y PATCH datos */ };
  const cambiarPassword = () => { /* abrir modal y PATCH contraseña */ };
  return (
    <tr>
      <td colSpan={5}>
        {/* formularios y AceptarCancelarBtn */}
      </td>
    </tr>
  );
};
```

UsuariosTabla.tsx

Propósito: Mostrar lista de usuarios y permitir habilitar/deshabilitar, editar o asignar roles.

- Propiedades:
- `datosUsuarios`, `datosRoles`.
- `actualizarDatosUsuarios`, `tablaRef`.
- Funciones:
- `habilitarUsuarioHandler`: PATCH habilitado del usuario.
- Renderiza:
- Fila principal con datos básicos y botones de acción.
- Si se solicita, despliega `UsuariosEditar` o lista de `UsuariosPermiso`.

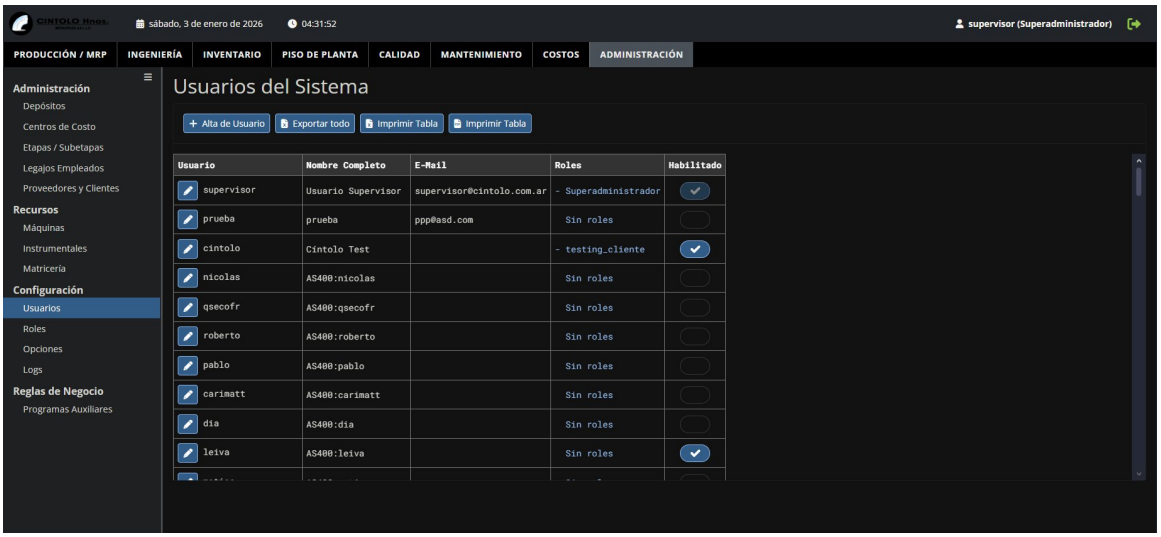

Figura x – Pantalla principal de Usuarios con listado y acciones de alta, habilitado, asignar roles, exportación e impresión.

tsx

```
const UsuariosTabla: FC<Props> = ({ datosUsuarios, datosRoles, actualizarDatosUsuarios,
tablaRef }) => {
  const [edicionUsuarioId, setEdicionUsuarioId] = useState(0);
  const [idMostrarEditar, setIdMostrarEditar] = useState(0);

  return (
    <div>
      <table ref={tablaRef}>
        <thead>{/* encabezado */}</thead>
        <tbody>
          {datosUsuarios?.map(u => (
            <Fragment key={u.id}>
              {/* fila principal */}
              {!!idMostrarEditar && /* UsuariosPermiso */}
              {!!edicionUsuarioId && /* UsuariosEditar */}
            </Fragment>
          )) || <NoHayElementos />}
        </tbody>
      </table>
    </div>
  );
};
```

UsuariosPermiso.tsx

Propósito: Suscribir o desuscribir un usuario a un rol.

- Estado local `permiso`: si el usuario ya tiene ese rol.
- Validación con `usuario.nivel` para permisos de edición.
- Al cambiar, muestra confirmación y realiza PUT `/api/usuarios/{id}/roles`.

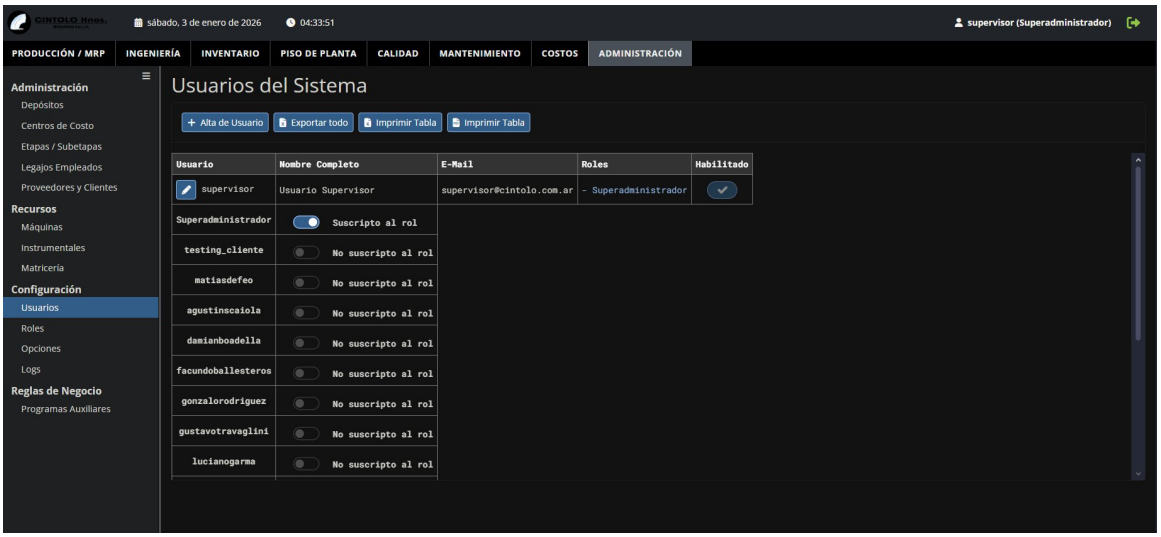

Figura x - Vista detallada de un usuario seleccionado con listado de roles.

tsx

```
const UsuariosPermiso: FC<Props> = ({ datosUsuario, rol, actualizarDatosUsuarios }) => {
  const [permiso, setPermiso] = useState(/* existe el rol */);

  const suscribirUsuarioARolHandler = async (suscribir: boolean) => {
    if (!esValido) return;
    dispatch(abrirModalReducer({ /* confirmación */ }));
  };

  return (
    <td>
      <input
        type='checkbox'
        checked={permiso}
        onChange={() => suscribirUsuarioARolHandler(!permiso)}
        disabled={!esValido}
      />
      <label>{permiso ? 'Suscripto al rol' : 'No suscripto al rol'}</label>
    </td>
  );
};
```

Fallos detectados en la lógica

- **Estado inexistente** en `Roles.tsx` y `Usuarios.tsx`:

Se usa `cargandoLocal` y `setCargandoLocal` sin declarar el estado correspondiente.

- **Fuga de memoria** por `setInterval` sin limpieza:

El `useEffect` usa `setInterval` para desactivar un loader y nunca lo limpia con `clearInterval`.

- **Uso de `setInterval` en vez de `setTimeout`**:

Se ejecuta cada 400 ms en lugar de una sola vez.

- **Dependencias faltantes** en efectos:

`useEffect` que llama a la API no declara dependencias para evitar advertencias.

- **Manejo genérico de errores**:

En algunos PATCH/DELETE no se controla el estado de `guardando` si la petición falla antes de asignar `setGuardando(false)`.

- **Identificadores dinámicos** en `RolesPermisoTerciario`:

El `key` de `<tr>` usa `subitem.path` sin concatenación consistente, podría colisionar si hay rutas iguales.