Introduccion

La presente documentación técnica describe el desarrollo e implementación de una plataforma web diseñada para el **CIATEQ**, cuyo propósito principal es facilitar la **gestión integral de investigadores, estudiantes, proyectos, artículos científicos, eventos académicos y líneas de investigación** dentro de la institución.

Este sistema centralizado busca digitalizar y simplificar procesos que anteriormente se realizaban de forma manual o dispersa, brindando una herramienta moderna, escalable y segura para el registro, consulta y administración de la información científica y académica de los distintos actores del ecosistema de investigación.

Funcionalidades principales:

- **Registro y administración de investigadores** y su relación con artículos, proyectos, eventos y líneas de investigación.
- **Gestión de estudiantes** vinculados a investigadores, incluyendo información sobre carrera, tipo de estudiante, escuela y fechas de estancia.
- **Seguimiento de proyectos** de investigación, sus herramientas asociadas y los ingresos generados.
- **Control de eventos** académicos y científicos con vinculación a investigadores.
- **Administración de artículos científicos** con sus respectivos datos bibliográficos y coautorías.
- **Vinculación de herramientas tecnológicas y líneas de investigación** a proyectos e investigadores respectivamente.

Roles y accesos:

La plataforma cuenta con un sistema de autenticación que distingue dos tipos principales de usuarios:

- **Vista de Administrador**: Acceso completo a todas las funcionalidades de gestión y administración de los datos en el sistema.
- **Vista de Usuario Común**: Acceso restringido a la visualización y consulta de datos relevantes, según los permisos otorgados.

Tecnologías utilizadas:

Base de datos:

- **PostgreSQL 17.4**

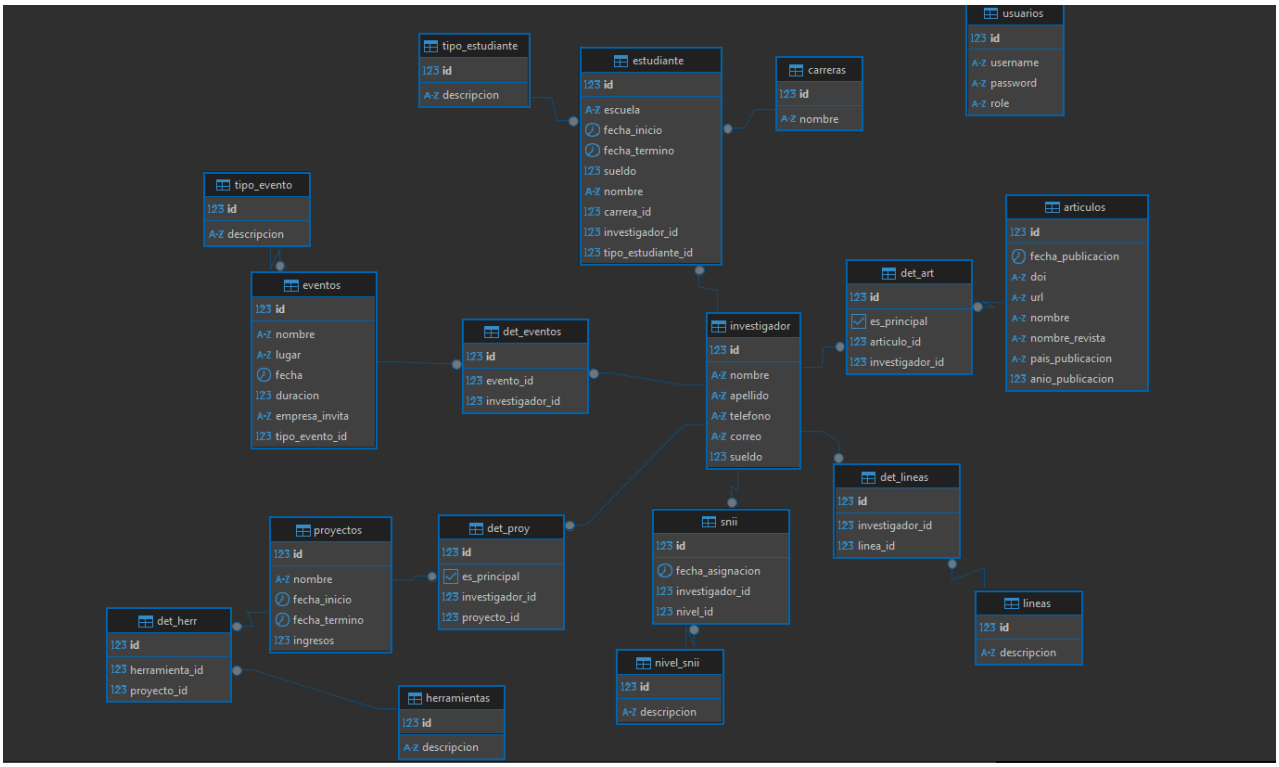
Backend:

- **NginX**
- **Python *3.13*** con:
    - Django 5.1.7
    - djangorestframework
    - django-filter
    - django-cors-headers
    - djangorestframework_simplejwt

Frontend:

- **Node *v23.7.0* y NPM *10.9.2*** con:
    - React usando Vite *6.2.0*
    - Tailwind CSS *4.0.15*
    - Lucide React Icons

Diagrama Relacional de La Base de Datos



La tabla **Investigador** es la entidad principal del sistema, y en ella se almacena información relevante de cada investigador.
Tiene una relación de **uno a uno** con la tabla **SNII**, encargada de registrar el nivel SNII asignado a cada investigador. Esta relación se implementa con una restricción que obliga a que investigador_id sea único. Además, SNII incluye una llave foránea hacia la tabla **NivelSnii**, que funciona como un catálogo de niveles del sistema SNII.

Investigador también mantiene una relación de **uno a muchos** con la tabla **Estudiante**, que almacena información sobre los estudiantes vinculados a cada investigador. Esta tabla incluye, además, dos llaves foráneas: una hacia **TipoEstudiante**, que es un catálogo de tipos de estudiantes, y otra hacia **Carreras**, catálogo que lista las distintas carreras académicas.

En cuanto a la relación con publicaciones, **Investigador** se vincula en una relación de **muchos a muchos** con la tabla **Articulos**, que contiene los datos de cada artículo

registrado en la plataforma. Esta relación se maneja mediante la tabla intermedia **DetArt**, la cual también registra si un investigador es el autor principal del artículo.

De manera similar, la relación entre **Investigador** y **Lineas** (líneas de investigación) también es de muchos a muchos, y se gestiona mediante la tabla intermedia **DetLineas**.

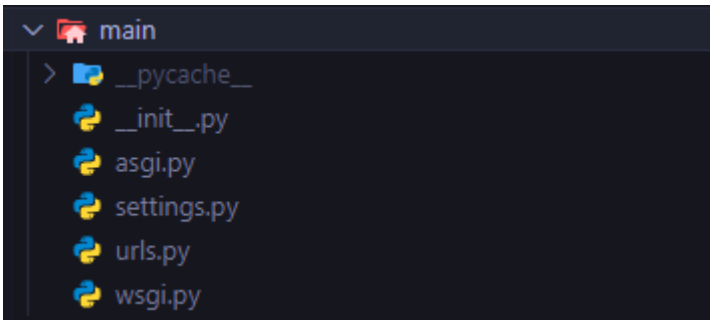
Investigador y **Eventos** están conectados igualmente mediante una relación de **muchos a muchos**, gestionada por la tabla intermedia **DetEventos**. La tabla **Eventos**, además de registrar información general de cada evento, contiene una llave foránea hacia la tabla **TipoEvento**, que actúa como catálogo de tipos de eventos.

Finalmente, **Investigador** tiene una relación de **muchos a muchos** con **Proyectos**, gestionada mediante la tabla intermedia **DetProy**, la cual indica además si un investigador es el responsable principal del proyecto.
Por su parte, **Proyectos** también se relaciona de forma **muchos a muchos** con la tabla **Herramientas**, vínculo que se implementa a través de la tabla intermedia **DetHerr**.

Backend

Estructura del proyecto:

Toda la API está dentro de la carpeta “backend”. Dentro, nos encontramos con 3 ficheros: manage.py, que es el archivo que gestiona Django. La carpeta main, y la carpeta gestión.



Main contiene los archivos base de Django. settings.py es de suma importancia: aquí es en donde Django se conecta a la base de datos mediante el uso de la librería psycopg2.

Esto es en settings.py, aquí se declara la conexión a la DB.

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "ciateq_investigadores_django",
        "USER": "postgres",
        "PASSWORD": "postgres",
        "HOST": "localhost",
        "PORT": "5432",
    }
}
```

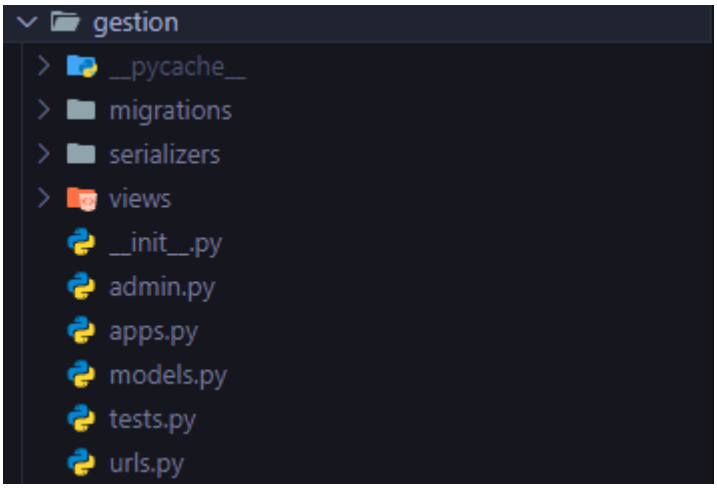
Aquí también se maneja el CORS usando la librería de django “django-cors-headers”. Esta librería funciona para que la API pueda ser accesible desde el servidor frontend. En esta sección es en donde se especifican las rutas permitidas, en este caso, es la ruta del servidor frontend de desarrollo.

```
CORS_ALLOWED_ORIGINS = [
    "http://localhost:5173",
    "http://localhost:5174",
]
```

En urls.py se hace referencia hacia las urls definidas en “gestion”

```python
urlpatterns = [
    path("admin/", admin.site.urls),
    path("gestion/", include("gestion.urls")),
]
```

La carpeta **gestion** contiene toda la API, aquí se declaran los modelos, vistas, serializers, urls de la aplicación, y se generan los archivos de migración.

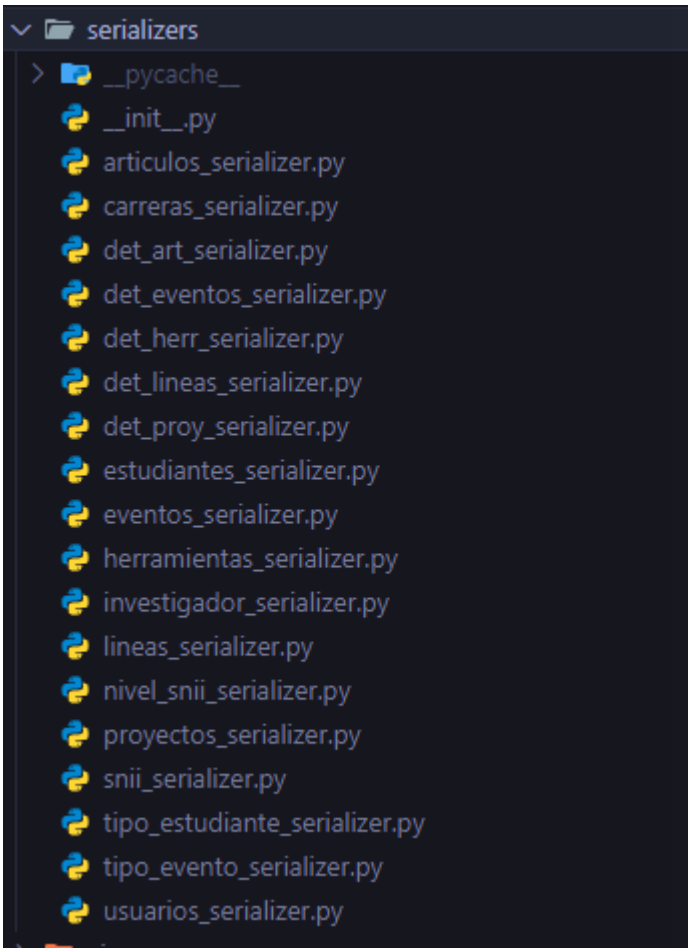


En models.py se escriben los modelos de las tablas para un manejo más sencillo si se requiere hacer modificaciones.

```python
class Articulos(models.Model):
    id = models.AutoField(primary_key=True)
    fecha_publicacion = models.DateField()
    doi = models.CharField(max_length=100, blank=True, null=True)
    url = models.CharField(max_length=200, blank=True, null=True)
    nombre = models.CharField(max_length=200)
    nombre_revista = models.CharField(max_length=200)
    pais_publicacion = models.CharField(max_length=100, blank=True, null=True)
    anio_publicacion = models.IntegerField(blank=True, null=True)

    class Meta:
        db_table = "articulos"
```

Este es uno de los modelos, se declaran los campos junto con su tipo de dato y restricciones si es que aplica. Además, se hace referencia a la tabla de la base de datos con la que se tiene que comunicar este modelo.

La carpeta serializers almacena un serializer por tabla. Se hizo uso de django-rest-framework para la creación de serializers



Articulos, carreras, herramientas, investigador, líneas, nivel snii, proyectos, tipo estudiante, tipo evento, usuarios, tienen todos una estructura como la siguiente:

```python
from rest_framework import serializers
from gestion.models import Articulos


class ArticulosSerializer(serializers.ModelSerializer):
    class Meta:
        model = Articulos
        fields = "__all__"
```

Se tiene una clase en donde se hace referencia al modelo del cual tiene que obtener la información, y se especifican los campos que debe de obtener el serializer, en este caso, todos.

Los demás serializers tienen la siguiente estructura:

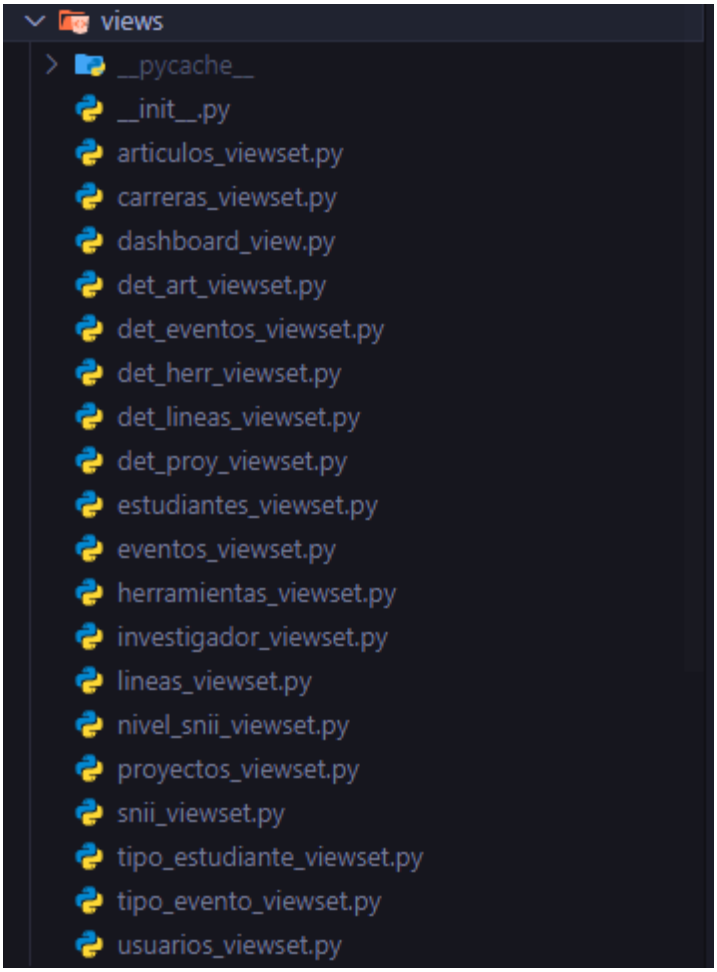
```python
from rest_framework import serializers
from gestion.models import Estudiante


class EstudiantesSerializer(serializers.ModelSerializer):
    nombre_investigador = serializers.CharField(
        source="investigador.nombre", read_only=True
    )
    nombre_carrera = serializers.CharField(source="carrera.nombre", read_only=True)
    descripcion_tipo_estudiante = serializers.CharField(
        source="tipo_estudiante.descripcion", read_only=True
    )

    class Meta:
        model = Estudiante
        fields = [
            "id",
            "investigador",
            "carrera",
            "tipo_estudiante",
            "escuela",
            "fecha_inicio",
            "fecha_termino",
            "sueldo",
            "nombre",
            "nombre_investigador",
            "nombre_carrera",
            "descripcion_tipo_estudiante",
        ]
```

Esta estructura la llevan los serializers los cuales pertebecen a tablas que contienen 1 o más campos como llaves foráneas. Esto se hace para que a la hora de acceder a la información, en lugar de solamente mostrar el ID al que está referenciando, muestre también un campo de esa otra tabla asociado a ese ID.

Por ejemplo, la tabla estudiante tiene 3 llaves foráneas: id_investigador, id_carrera, id_tipo_estudiante. Si tuviéramos un registro con id_investigador = 3, id_carrera = 2, id_tipo_estudiante = 1, además de ver cada ID, veriamos el nombre del investigador, el nombre de la carrera, y la descripción del tipo de estudiante, todo desde un único serializer.

Es importante mencionar que dichos campos foráneos que no pertenecen como tal a la tabla estudiantes se obtienen como “read_only”.

La carpeta views contiene todas las viewsets. Los viewsets se crearon a partir del rest-framework de django. Esto permite crear el CRUD completo de cada endpoint de manera sencilla y directa.



Carreras, líneas, nivel snii, tipo estudiante, tipo evento, y usuarios llevan la siguiente estructura:

```python
from rest_framework import viewsets
from gestion.models import TipoEstudiante
from gestion.serializers import TipoEstudianteSerializer


class TipoEstudianteViewSet(viewsets.ModelViewSet):
    queryset = TipoEstudiante.objects.all()
    serializer_class = TipoEstudianteSerializer
```

Se importa el modelo al cual queremos acceder, así como el serializer correspondiente a dicho modelo. En queryset especificamos que queremos toda la información de la tabla, y en serializer_class se escribe el serializer.

Los demás viewsets tienen una estructura similar, solo que se agregan campos de filtrado y/o ordenado según sea el caso. Esto se hizo usando la librería django-filters

```python
from rest_framework import viewsets, filters
from django_filters.rest_framework import DjangoFilterBackend
from gestion.models import Estudiante
from gestion.serializers import EstudiantesSerializer


class EstudiantesViewSet(viewsets.ModelViewSet):
    queryset = Estudiante.objects.all()
    serializer_class = EstudiantesSerializer
    filter_backends = [DjangoFilterBackend, filters.OrderingFilter]

    filterset_fields = ["investigador", "carrera", "tipo_estudiante"]

    ordering_fields = [
        "id",
        "escuela",
        "fecha_inicio",
        "fecha_termino",
        "investigador",
        "carrera",
        "tipo_estudiante",
    ]
    ordering = ["id"]
```

Después de especificar el serializer, se declaran los campos por los cuales se puede filtrar la información, y los campos por los que se puede ordenar la información. Además, se define un campo por defecto de ordenación.

“dashboard_view” es diferente al resto de vistas, esta es una api_view, mientras las demás son viewsets.

```python
from rest_framework.decorators import api_view
from rest_framework.response import Response
from datetime import date
from gestion.models import Investigador, Proyectos, Articulos, Estudiante, Eventos


@api_view(["GET"])
def dashboard_data(request):
    hoy = date.today()
    anio = hoy.year
    mes = hoy.month

    data = {
        "investigadores": Investigador.objects.count(),
        "proyectos_total": Proyectos.objects.count(),
        "proyectos_este_anio": Proyectos.objects.filter(
            fecha_inicio__year=anio
        ).count(),
        "articulos_este_anio": Articulos.objects.filter(anio_publicacion=anio).count(),
        "articulos_este_mes": Articulos.objects.filter(
            fecha_publicacion__year=anio, fecha_publicacion__month=mes
        ).count(),
        "estudiantes": Estudiante.objects.count(),
        "estudiantes_terminan_este_anio": Estudiante.objects.filter(
            fecha_termino__year=anio
        ).count(),
        "eventos_total": Eventos.objects.count(),
        "eventos_proximos": Eventos.objects.filter(fecha__gte=hoy).count(),
        "eventos_este_anio": Eventos.objects.filter(fecha__year=anio).count(),
    }

    return Response(data)
```

Se hizo de esta manera ya que solo interesa obtener información relevante para mostrar en el dashboard, no interesa hacer un CRUD completo.

En urls.py se crean todos los endpoints de la aplicación mediante el uso de “DefaultRouter”

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from gestion.views import (
    InvestigadorViewSet,
    SniiViewSet,
    NivelSniiViewSet,
    LineasViewSet,
    DetLineasViewSet,
    DetArtViewSet,
    ArticulosViewSet,
    TipoEventoViewSet,
    EventosViewSet,
    DetEventosViewSet,
    HerramientasViewSet,
    DetHerrViewSet,
    ProyectosViewSet,
    DetProyViewSet,
    CarrerasViewSet,
    TipoEstudianteViewSet,
    EstudiantesViewSet,
    UsuariosViewSet,
    dashboard_data,
)
```

Se importan todas las vistas

```
router = DefaultRouter()
router.register(r"investigadores", InvestigadorViewSet)
router.register(r"nivelsnii", NivelSniiViewSet)
router.register(r"snii", SniiViewSet)
router.register(r"lineas", LineasViewSet)
router.register(r"detlineas", DetLineasViewSet)
router.register(r"detart", DetArtViewSet)
router.register(r"articulos", ArticulosViewSet)
router.register(r"tipoevento", TipoEventoViewSet)
router.register(r"eventos", EventosViewSet)
router.register(r"deteventos", DetEventosViewSet)
router.register(r"herramientas", HerramientasViewSet)
router.register(r"detherr", DetHerrViewSet)
router.register(r"proyectos", ProyectosViewSet)
router.register(r"detproy", DetProyViewSet)
router.register(r"carreras", CarrerasViewSet)
router.register(r"tipoestudiante", TipoEstudianteViewSet)
router.register(r"estudiantes", EstudiantesViewSet)
router.register(r"usuarios", UsuariosViewSet)


urlpatterns = [
    path("api/", include(router.urls)),
    path("api/dashboard/", dashboard_data, name="dashboard-data"),
]
```

Se declara el default router, y se añaden los registros, cada registro contiene el nombre que tendrá la URL del endpoint, y la vista a la que debe de acceder.

Los endpoints quedarán de la siguiente forma: http://127.0.0.1:8000/gestion/api/investigadores/ (cambia “investigadores” por el nombre especificado en cada registro para acceder a un endpoint diferente)

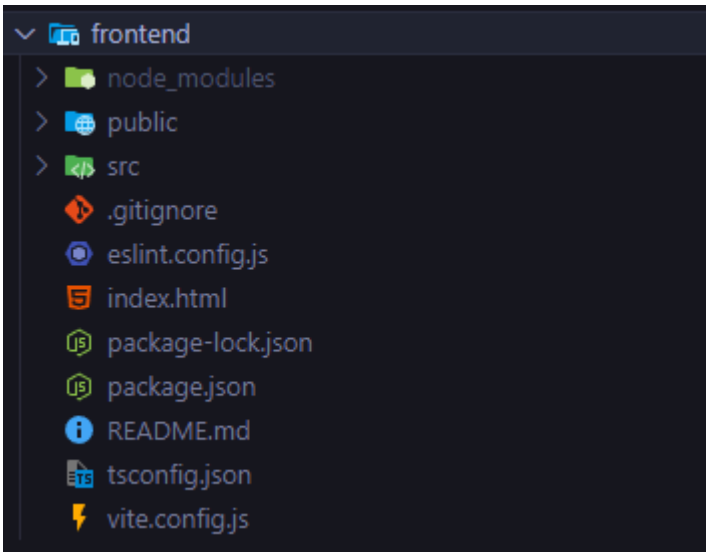
Al levantar el servidor de Python y acceder a un endpoint, verás lo siguiente:

```
[
  {
    "id": 1,
    "nombre": "Juan",
    "apellido": "Perez",
    "telefono": "1234567890",
    "correo": "juan@example.com",
    "sueldo": "35800.00"
  },
  {
    "id": 2,
    "nombre": "Pedro",
    "apellido": "Lopez",
    "telefono": "1234567890",
    "correo": "pedro@example.com",
    "sueldo": "45200.00"
  },
  {
    "id": 3,
    "nombre": "Alicia",
    "apellido": "Gomez",
    "telefono": "1234567890",
    "correo": "alicia@example.com",
    "sueldo": "35800.00"
  },
```

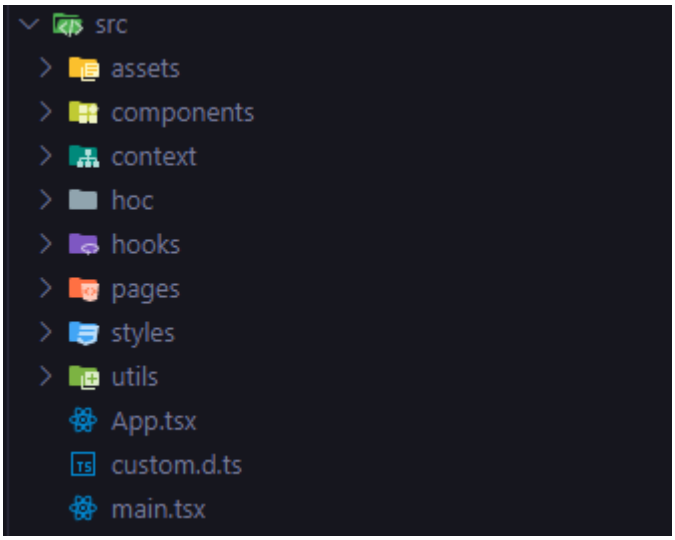

Con esto tenemos nuestra API 100% funcional y lista para alimentar al frontend.

Frontend

Todo el frontend está dentro de la carpeta “frontend”



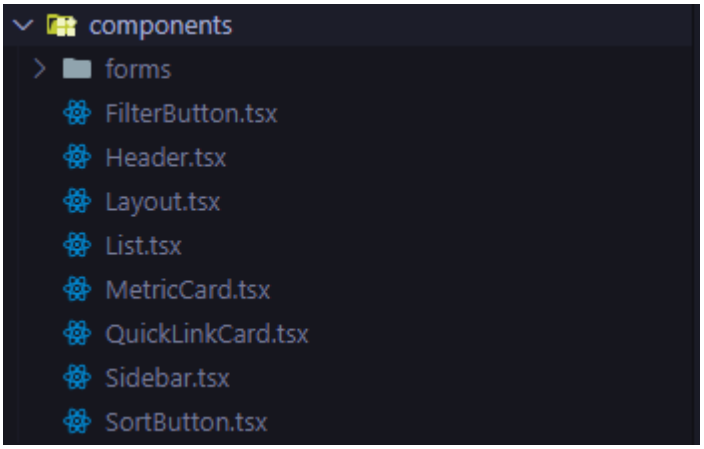
En “src” encontramos lo sustancial de la aplicación



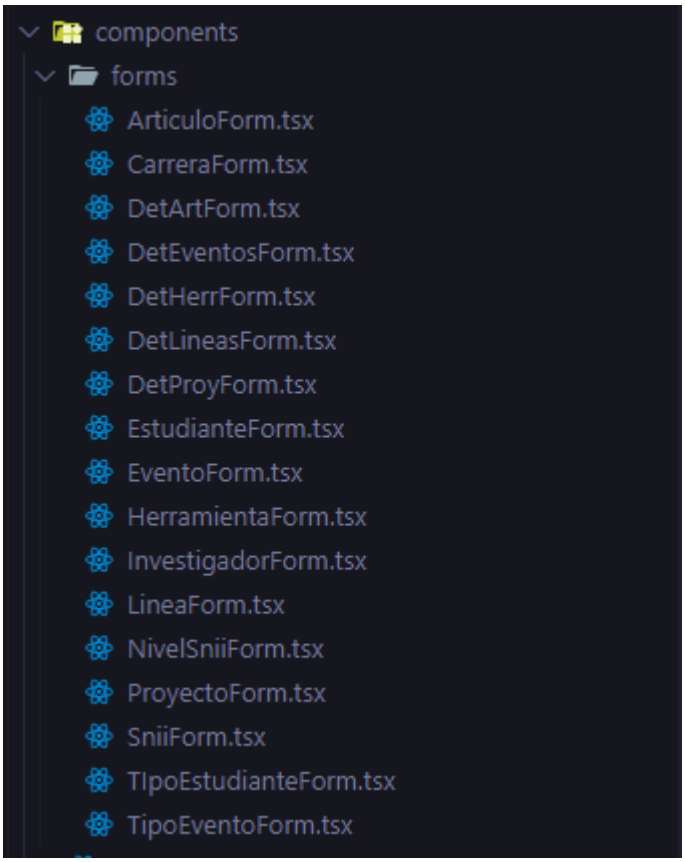
Vamos carpeta por carpeta:

Assets almacena imágenes usadas en la página.

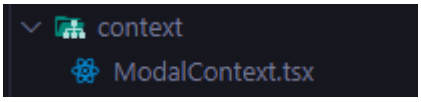
Components,



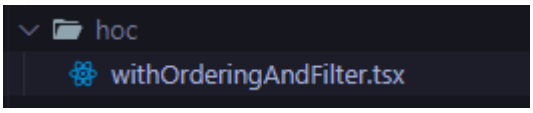
Tenemos componentes generales usados en toda la aplicación, dentro de la carpeta forms se encuentra un componente personalizado para cada formulario



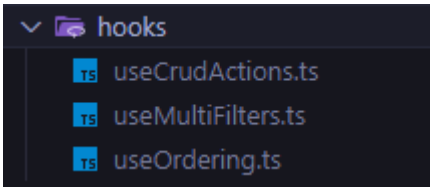
Context, aquí se declaran los contextos globales. Contiene un contexto global para el uso de modales



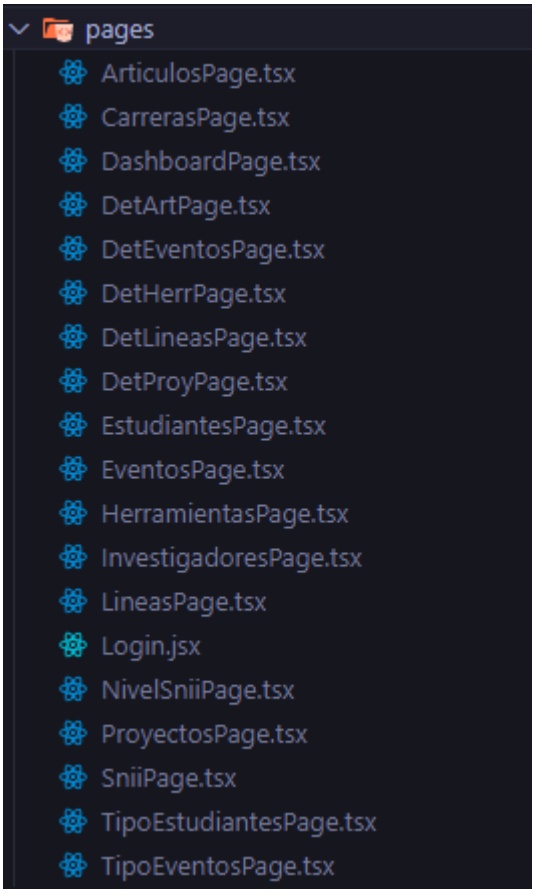
HOC, se almacenan los High Order Components. Contiene un HOC el cual le agrega propiedades de filtrado y/o ordenado a un componente



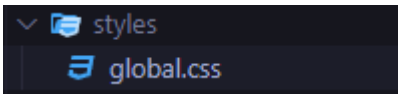
Hooks, los hooks personalizados usados en la aplicación. Nos encontramos con 3: useCrudActions, un hook para manejar el CRUD, useMultiFilters, para manejar el filtrado, y useOrdering, para manejar el ordenamiento.



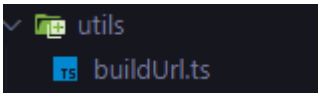
Pages, todas las páginas de la aplicación



Styles, se encuentra el archivo de estilos globales, que es en donde se importa tailwind



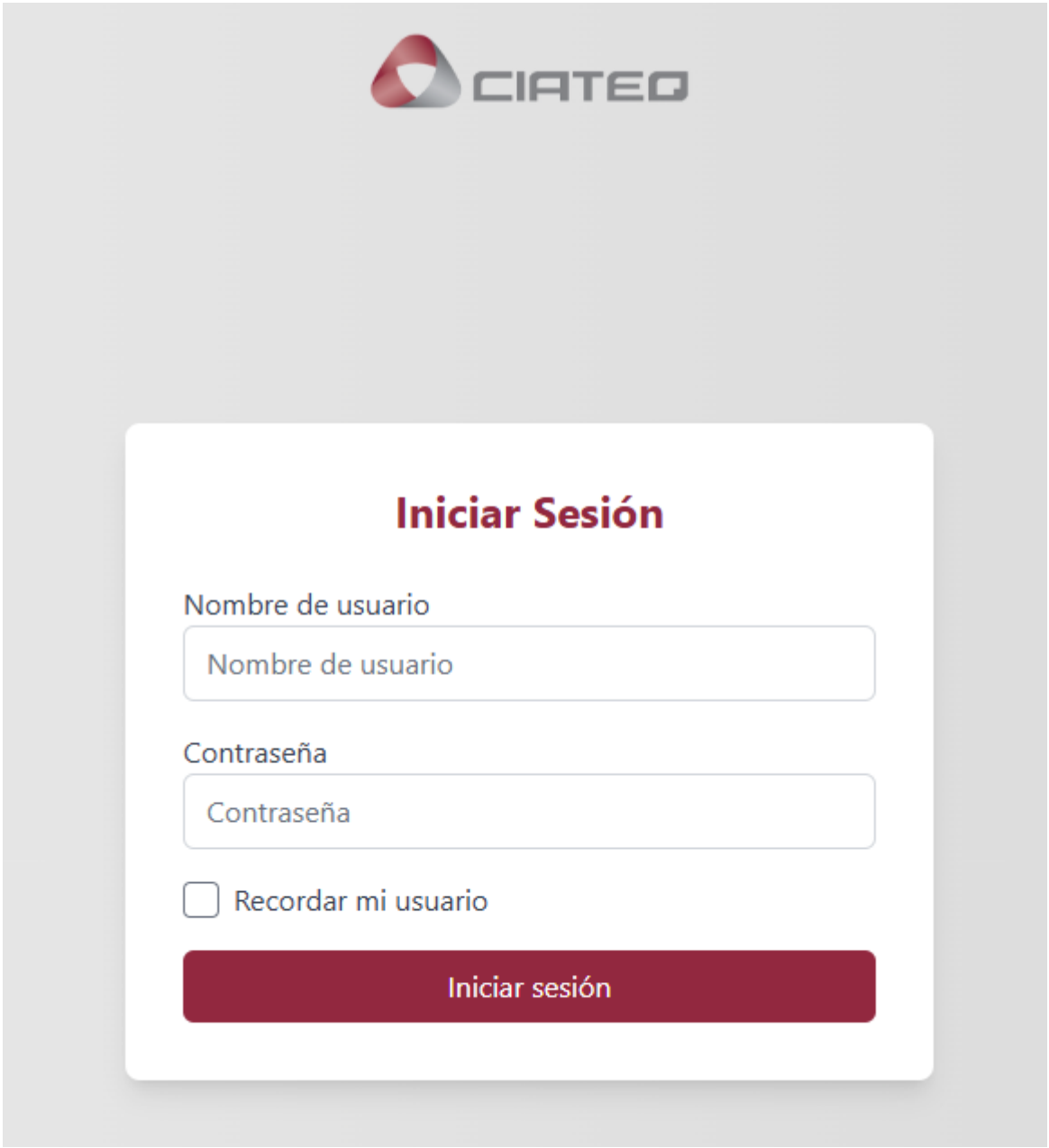
Utils, diferentes utilidades para reutilizarse dentro de la aplicación. Nos encontramos con la utilidad buildUrl, la cual funciona para construir URL's según el tipo de filtrado y/o ordenamiento que necesitemos.



En App.tsx se declaran todas las rutas, usando la librería “react-router-dom”

```tsx
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/login" element={<Login />} />
        <Route path="/dashboard" element={<DashboardPage />} />
        <Route path="/articulos" element={<ArticulosPage />} />
        <Route path="/proyectos" element={<ProyectosPage />} />
        <Route path="/herramientas" element={<HerramientasPage />} />
        <Route path="/lineas" element={<LineasPage />} />
        <Route path="/tipoeventos" element={<TipoEventosPage />} />
        <Route path="/carreras" element={<CarrerasPage />} />
        <Route path="/nivelsnii" element={<NivelSniiPage />} />
        <Route path="/tipoestudiante" element={<TipoEstudiantePage />} />
        <Route path="/investigadores" element={<InvestigadoresPage />} />
        <Route path="/snii" element={<SniiPage />} />
        <Route path="/eventos" element={<EventosPage />} />
        <Route path="/estudiantes" element={<EstudiantesPage />} />
        <Route path="/detart" element={<DetArtPage />} />
        <Route path="/deteventos" element={<DetEventosPage />} />
        <Route path="/detlineas" element={<DetLineasPage />} />
        <Route path="/detproy" element={<DetProyPage />} />
        <Route path="/detherr" element={<DetHerrPage />} />
        <Route path="*" element={<Navigate to="/dashboard" replace />} />
      </Routes>
    </Router>
  );
}

export default App;
```

Lo primero que ve el usuario al entrar es el login



Se utilizan useState y useEffect para manejar el estado y los efectos secundarios, respectivamente.

Funcionalidades clave:

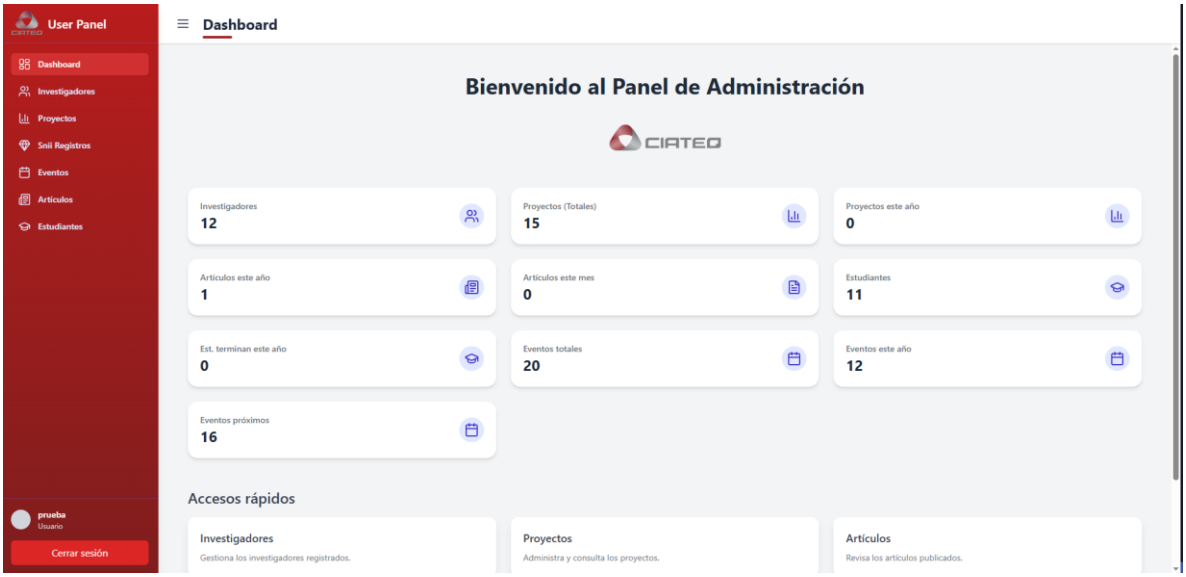
1. **Manejo de formularios y validaciones:**
    - Permite ingresar el nombre de usuario y la contraseña.
    - Valida que ambos campos estén completos antes de enviar la solicitud.

- Muestra mensajes de error personalizados en caso de campos vacíos o credenciales inválidas.

2. **Autenticación de usuario:**
    - Al enviar el formulario, el componente hace una petición GET al endpoint de usuarios (/gestion/api/usuarios/) para verificar si las credenciales ingresadas coinciden con algún usuario registrado.
    - Si la autenticación es exitosa, se guarda el usuario y su rol en localStorage o sessionStorage, dependiendo de si el usuario seleccionó la opción "Recordar mi usuario".
    - Luego, redirige automáticamente al **dashboard** de la plataforma (/dashboard).
3. **Persistencia de sesión:**
    - Al cargar el componente, useEffect revisa si existe un usuario previamente guardado en localStorage o sessionStorage. En caso afirmativo, rellena el campo de usuario automáticamente y redirige al dashboard.

Una vez que ingresa credenciales validas, si es admin, verá lo siguiente:

Si es usuario normal, verá lo siguiente:



El componente **DashboardPage** representa el panel principal de administración de la plataforma. Se encarga de mostrar al usuario un conjunto de métricas clave relacionadas con investigadores, proyectos, artículos, estudiantes y eventos.

Al cargarse, el componente:

- Verifica si hay un usuario autenticado, redirigiendo a la pantalla de login en caso contrario.
- Consulta los datos del dashboard desde el backend mediante una solicitud a la API.
- Muestra tarjetas métricas (MetricCard) con valores agregados del sistema.
- Incluye accesos rápidos (QuickLinkCard) a las secciones principales de gestión de la plataforma.
- Está envuelto dentro de un componente Layout

Componentes UI

El componente **Layout** actúa como **estructura base** de todas las páginas del sistema. Define el diseño principal con un **sidebar lateral** (menú de navegación) y una **cabecera superior** (Header), además de un área de contenido central donde se renderiza cada página (children).

Funcionalidades clave:

- Controla la visibilidad del sidebar mediante el estado sidebarOpen.
- Recibe el título de la página como prop (title) para mostrarlo en el encabezado.
- Proporciona una experiencia de navegación consistente y responsiva a lo largo de toda la plataforma.

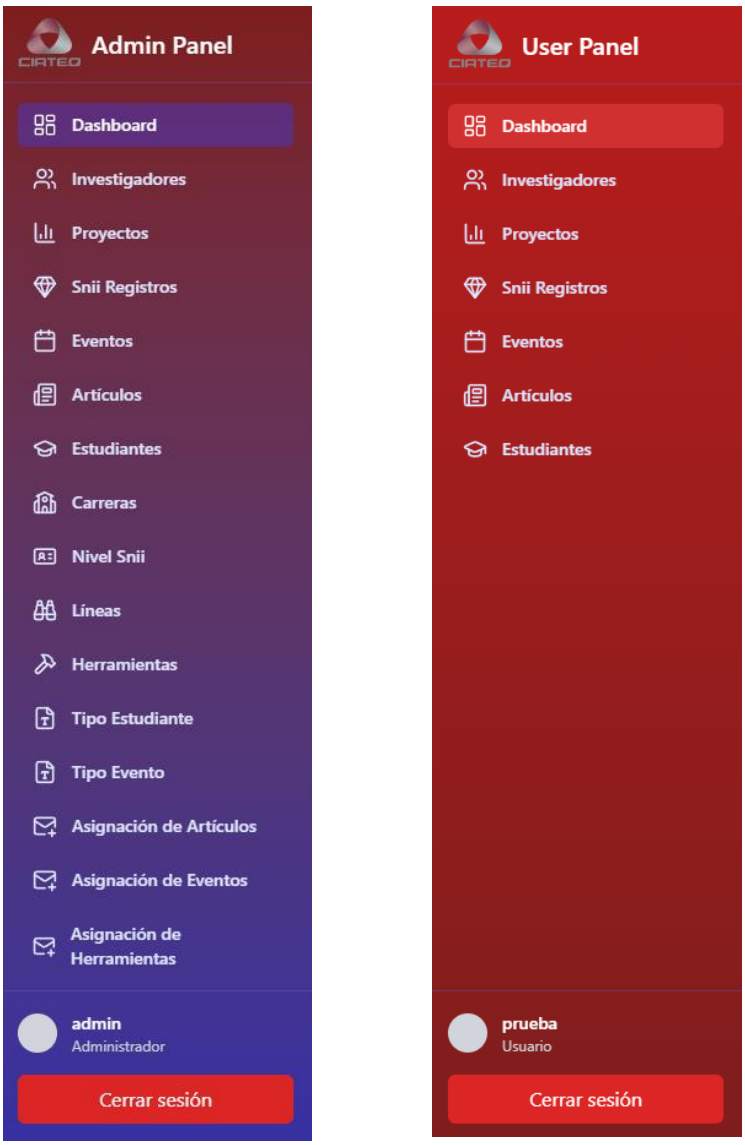
Este componente se utiliza como envoltorio en todas las páginas.

El componente **Sidebar** representa el **menú lateral de navegación principal** de la plataforma. Su función es proporcionar acceso rápido a las distintas secciones del sistema, diferenciando entre vistas de **usuario estándar** y **administrador**.

Funcionalidades clave:

- **Control de visibilidad** mediante la prop open, que permite mostrar u ocultar el sidebar en dispositivos móviles.
- **Personalización por rol**: El menú cambia de color y muestra diferentes opciones según si el usuario es **Administrador** o **Usuario**.
- **Navegación dinámica**: Usa React Router (Link, useLocation, useNavigate) para gestionar rutas internas y resaltar la ruta activa.
- **Persistencia de sesión**: Obtiene el nombre y rol del usuario desde localStorage o sessionStorage para mostrarlos en el perfil inferior.
- **Acción de cierre de sesión**: Elimina los datos del usuario de la sesión y redirige al login.

Sidebar Admin/Usuario



El componente **Header** representa la **barra superior** dentro del layout de la plataforma. Muestra el título de la sección actual y permite al usuario **abrir o cerrar el sidebar**, especialmente útil en dispositivos móviles.

Funcionalidades clave:

- Recibe el título dinámico de la página como prop (title) y lo muestra en la parte superior con un estilo destacado.
- Incorpora un botón con icono de menú (Menu de Lucide) para alternar la visibilidad del Sidebar.
- Está diseñado como un componente fijo (sticky) para mantener su visibilidad durante el desplazamiento del contenido.

El componente **List** es un componente **genérico y reutilizable** que permite mostrar una tabla interactiva con datos provenientes de cualquier entidad del sistema (como estudiantes, investigadores, artículos, etc.).

Funcionalidades principales:

- **Visualización de datos tabulares**: Recibe un arreglo de datos (data) y una definición de columnas (columns) para renderizar una tabla dinámica.
- **Ordenamiento**: Soporta ordenamiento por columnas cuando se combina con el componente SortButton. El campo ordenado y el orden ascendente/descendente son gestionados externamente e inyectados como props.
- **Filtrado por columnas**: Si se combina con el componente FilterButton, la tabla permite aplicar filtros por columna. Las opciones y valores filtrados también son manejados externamente.
- **Acciones integradas**:
    - **Edición (onEdit)**: Se habilita si se pasa la función onEdit como prop. Al hacer clic en un botón de edición en la fila, se dispara esa función con el ítem correspondiente.
    - **Eliminación (onDelete)**: Si se proporciona onDelete, se muestra un botón de eliminación por fila, que ejecuta dicha función al confirmar la acción.

Props principales:

- columns: Un arreglo de objetos que define cada columna (nombre visible, accessor para extraer el valor, y renderizado opcional).
- data: El arreglo de objetos a mostrar.
- rowKey: Nombre del campo que actúa como identificador único de cada fila.
- onEdit?: Función opcional para editar un ítem.
- onDelete?: Función opcional para eliminar un ítem.

La carpeta **/components/forms/** contiene una colección de formularios reutilizables, diseñados específicamente para cada entidad principal del sistema (como

Estudiantes, Investigadores, Proyectos, Artículos, etc.). Cada formulario permite **crear y editar registros** a través de una interfaz consistente, modular y controlada.

Estructura y funcionamiento general

Cada archivo dentro de la carpeta (EstudianteForm.tsx, InvestigadorForm.tsx, etc.) define un **componente React funcional** que sigue un patrón común:

- Utiliza **useState** para manejar los datos del formulario (formData) de forma controlada.
- Utiliza **useEffect** para precargar datos cuando se edita un registro existente.
- Recibe listas de opciones externas (por ejemplo, investigadores, carreras, tipos, etc.) para alimentar los campos tipo select.
- Convierte automáticamente los valores al tipo correcto antes de enviarlos (number, float, string, date).

Props estandarizadas en todos los formularios:

- entidad: (opcional) objeto con los datos del registro actual si se está editando. Si es null, se asume creación.
- onSubmit: función que se ejecuta al guardar el formulario, y que recibe el objeto con los datos.
- onClose: función para cerrar el formulario/modal.
- opciones...: arreglos con datos de catálogo que alimentan selectores desplegables (por ejemplo, lista de investigadores, carreras, niveles, etc.).

Contextos globales

El ModalContext es un **contexto global** que gestiona la apertura y cierre de modales en toda la aplicación. Proporciona una forma centralizada y reutilizable de mostrar contenido en ventanas emergentes desde cualquier componente.

Componentes clave:

- **ModalProvider**: Componente proveedor que encapsula toda la lógica y el renderizado del modal. Debe envolver la aplicación (o una parte relevante de ella).

- **useModal**: Hook personalizado que permite acceder a las funciones openModal y closeModal desde cualquier componente dentro del proveedor.

Funcionalidades:

- **openModal(content, title)**: Muestra un modal con el contenido proporcionado. Opcionalmente, puede incluir un título.
- **closeModal()**: Cierra el modal actual.
- **Renderizado con ReactDOM.createPortal**: El modal se renderiza en un nodo HTML externo (#modal-root) para evitar conflictos de estilo y garantizar que se muestre por encima de todo.

Hooks y HOCs

El hook **useCrudActions<T>** encapsula la lógica de operaciones **CRUD (Create, Read, Update, Delete)** para cualquier entidad del sistema, permitiendo reutilizar el mismo patrón en todas las páginas que interactúan con la API REST.

Funcionalidades clave:

- Permite cargar (fetchData), crear (create), actualizar (update) y eliminar (remove) registros.
- Maneja estados de carga (loading), errores (error) y edición (editingItem).
- Mantiene en memoria la lista actualizada de registros (data) de forma reactiva.
- Se utiliza en combinación con tablas (List) y formularios (Form) en las páginas CRUD.

El hook **useMultiFilters** proporciona una forma sencilla y reutilizable de manejar múltiples filtros dinámicos dentro de una página. Es usado principalmente en las páginas CRUD para controlar los filtros aplicados a las tablas.

Funcionalidades clave:

- **Inicialización de filtros**: Recibe un objeto con los filtros iniciales y los almacena en el estado.
- **Modificación individual de filtros**: Permite cambiar el valor de un filtro específico mediante setFilter(field, value).

- **Reinicio de filtros**: Restaura todos los filtros a su estado inicial con clearFilters().

El hook **useOrdering** gestiona el **estado de ordenamiento** de columnas en tablas de datos. Es utilizado junto con componentes como SortButton y la tabla List para permitir que el usuario ordene los registros por diferentes campos.

Funcionalidades clave:

- **Estado de ordenamiento (ordering)**: Representa el campo por el cual se ordenan los datos. Si es negativo (-campo), indica orden descendente.
- **toggleOrdering(field)**: Alterna el orden del campo seleccionado entre ascendente y descendente. Si se selecciona un nuevo campo, lo ordena de forma ascendente por defecto.

El **High Order Component (HOC)** withOrderingAndFilter encapsula y reutiliza la lógica de **ordenamiento** y **filtrado** para componentes de tipo CRUD. Permite inyectar automáticamente estas funcionalidades a cualquier componente que maneje tablas de datos.

Funcionalidades clave:

- Inyecta las props:
    - ordering: campo actual por el que se ordena.
    - toggleOrdering: función para alternar el orden (ascendente/descendente).
    - filters: objeto con los filtros activos.
    - setFilter: función para modificar filtros individualmente.
    - clearFilters: función para restablecer todos los filtros.
- Internamente, usa los hooks useOrdering y useMultiFilters para manejar el estado.

Páginas CRUD

Cada entidad principal del sistema (como Estudiantes, Investigadores, Proyectos, etc.) cuenta con su propia página dedicada, construida bajo una estructura reutilizable orientada a mostrar, filtrar, ordenar y administrar registros mediante una interfaz en forma de tabla.

Funcionalidades clave:

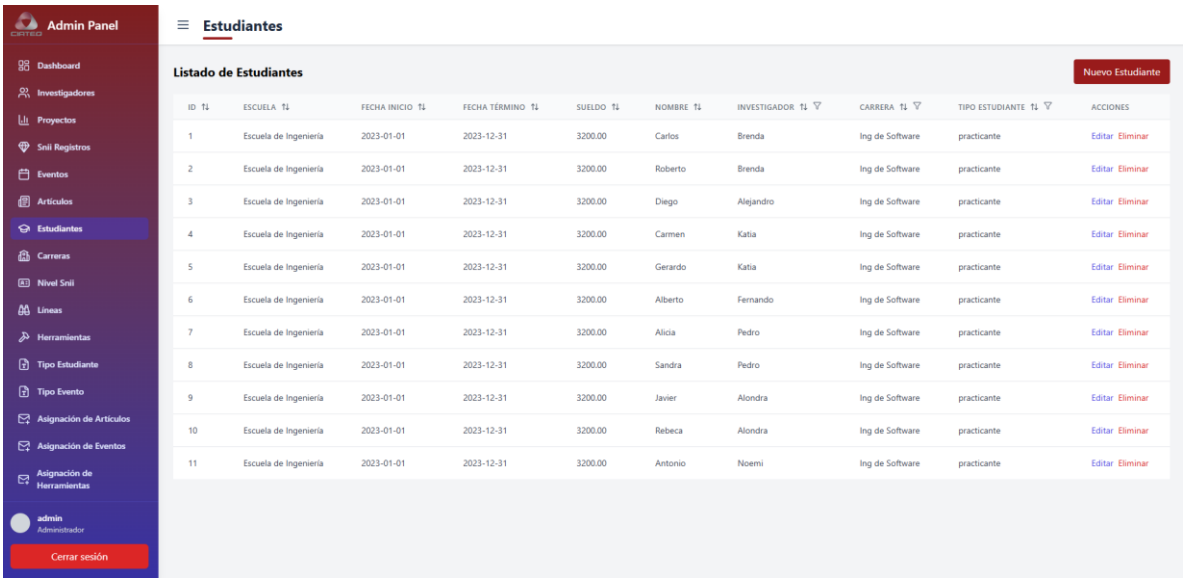
- **Autenticación protegida**: Verifica si el usuario ha iniciado sesión antes de mostrar la página.
- **Control de permisos**: Solo los usuarios con rol de administrador pueden crear, editar o eliminar registros. Los usuarios normales tienen acceso únicamente de lectura.
- **Tabla interactiva**: Se utiliza un componente genérico List que permite:
    - Visualizar todos los registros.
    - Ordenar por columnas mediante SortButton.
    - Filtrar datos mediante FilterButton.
    - Acciones de edición y eliminación (si el usuario tiene permisos).
- **Modal de formulario**: Al crear o editar un registro, se abre un formulario (EstudianteForm) en un modal para capturar la información.
- **Abstracción del CRUD**: Se utiliza un hook genérico useCrudActions que encapsula las operaciones de fetch, create, update y delete contra la API REST.
- **Carga dinámica de filtros**: Las opciones de los filtros (como investigadores, carreras, etc.) se cargan dinámicamente desde endpoints específicos al iniciar la página.

Organización de código:

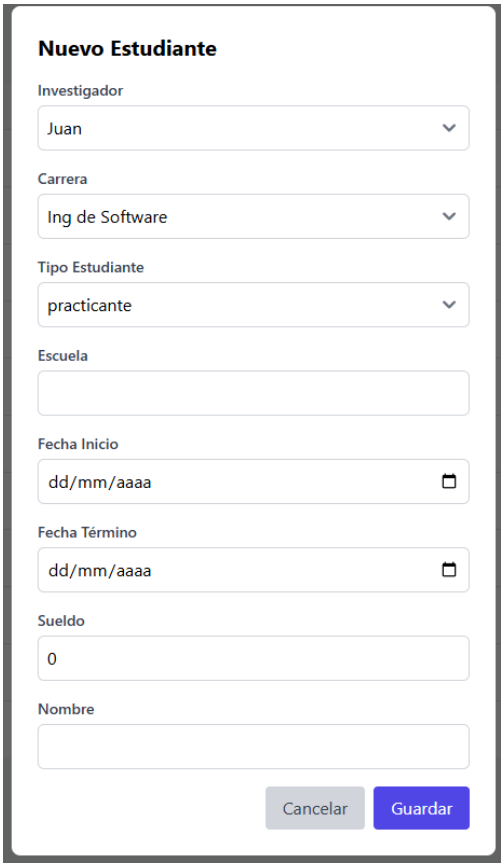
- Las props como ordenamiento y filtros son inyectadas por un HOC (withOrderingAndFilter) que estandariza este comportamiento entre todas las páginas CRUD.
- Las URL base y endpoints auxiliares se declaran al inicio de cada archivo para mantener la mantenibilidad.

Explicando flujo de la aplicación

Una vez iniciado sesión y ser redirigido al dashboard, navega en la sidebar y clickea algún botón, este te llevara a la página correspondiente. Prueba los botones de filtrado y ordenamiento.



Al dar click en Nuevo Estudiante, se abrirá un modal para crear un nuevo registro



Al editar un registro, se abrirá un modal pero con los datos precargados de ese item.