UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS

Facultad de Ingeniería de Sistemas e Informática

“Segundo Entregable”

CURSO

Procesos de software

DOCENTE

Arredondo Castillo, Gustavo

INTEGRANTES GRUPO 6

Arboleda Sanchez, Ericka Sofia	24200047
Cerna Sifuentes, Augusto Nicolas	24200053
Siesquen Torres, Katherine Lizbeth	24200066
Postigo Vega, Diana Carolina	24200167
Grijalva De La Cruz, Rony Smith	24200058
Tonconi Isidro, Maria Sunori	24200176

Lima, Perú
2025

Índice

Proyecto: Intranet Académica – Colegio Cristo Redentor

I. Resumen Ejecutivo

El presente documento corresponde al Segundo Entregable del proyecto "Intranet Académica – Colegio Cristo Redentor", una plataforma web diseñada para centralizar y optimizar la gestión académica de la institución. El sistema tiene como objetivo principal digitalizar los procesos de registro de notas, cálculo de promedios y generación de libretas, facilitando la interacción entre los roles de Directora/Director, Tutor y Polidocente, y garantizando la integridad y trazabilidad de la información.

Esta segunda entrega reporta el avance consolidado de la etapa de desarrollo, donde se han implementado los módulos y funcionalidades clave, incorporando afinamientos estratégicos respecto al primer entregable. Los cambios más significativos incluyen la eliminación del módulo de Asistencia y del rol Alumno para simplificar el alcance, la redefinición del módulo de Notas Mensuales para que sea configurable por el Tutor, y la división del módulo de Libretas Académicas en dos submódulos especializados: Libretas Bimestrales (PDF) y reportes para UGEL (Excel). Asimismo, se ha establecido un flujo de trabajo controlado para el envío y consolidación de calificaciones entre polidocentes y tutores.

El proyecto se desarrolla sobre una arquitectura de 4 capas, utilizando un stack tecnológico compuesto por Django para el backend, React para el frontend y postgreSQL como gestor de base de datos. Actualmente, el equipo se encuentra en la fase de desarrollo e integración, con un avance significativo en todos los módulos asignados, respaldado por la ejecución de pruebas unitarias y funcionales que aseguran la correcta implementación de las reglas de negocio y la consistencia de los datos.

II. Contexto y Alcance

Contexto

El proyecto surge como respuesta a la necesidad de digitalizar los procedimientos manuales que los colegios suelen realizar en hojas de cálculo o formatos físicos, los cuales dificultan el control de versiones, la consolidación de notas y la comunicación entre docentes y dirección. La Intranet permite centralizar toda la información académica en una sola plataforma, ofreciendo seguridad, trazabilidad y acceso controlado por roles (Directora/Director, Tutor y Polidocente).

El presente proyecto tiene como objetivo principal implementar una plataforma web que centralice y optimice la gestión académica de la institución educativa, facilitando la interacción entre los diferentes roles administrativos y docentes. La Intranet busca automatizar los procesos relacionados con registro de notas mensuales, exámenes bimestrales, libretas académicas y reportes para UGEL, garantizando la integridad de los datos, la trazabilidad de las acciones y la correcta generación de documentos oficiales.

Alcance

El alcance de esta segunda etapa abarca la implementación y afinamiento funcional de los módulos principales, considerando las siguientes mejoras respecto al primer entregable:

- Eliminación del módulo de Asistencia y del rol Alumno, simplificando la estructura de acceso y las vistas del sistema.
- Desarrollo y consolidación de los módulos Notas Mensuales, Examen Bimestral, Libretas Académicas Bimestrales (PDF) y UGEL (Excel).
- Integración de la Calculadora numérica a literal y del catálogo de comentarios dentro del flujo de generación UGEL.

- Definición de flujos Polidocente → Tutor para el envío, validación y cierre de calificaciones.
- Consolidación automática de datos bimestrales y generación de documentos finales (PDF y Excel) con estructuras estandarizadas.

El proyecto se encuentra en la etapa de desarrollo e integración, donde cada integrante tiene responsabilidades definidas por módulo. En esta fase, se prioriza la consistencia de datos entre el backend y el frontend, la validación de reglas de negocio (cierres, promedios, permisos) y la ejecución de pruebas unitarias y funcionales que respalden la calidad del producto.

III. Cambios y Afinamientos con Respecto al Primer Entregable

Durante la segunda etapa del proyecto **"Intranet Académica – Colegio Cristo Redentor"**, se llevaron a cabo diversas modificaciones y afinamientos en comparación con el primer entregable, con el propósito de optimizar las funcionalidades planteadas, fortalecer la arquitectura del sistema y mejorar la interacción de los usuarios dentro de la plataforma. Estas actualizaciones responden a los requerimientos técnicos, funcionales y de usabilidad identificados en la fase inicial del desarrollo.

1. Eliminación de módulos y simplificación de roles

Uno de los principales cambios consistió en la eliminación del módulo de **Asistencia**, debido a que su desarrollo no resultaba prioritario dentro del alcance actual del proyecto. De igual manera, se retiró el **rol de Alumno** tanto de la interfaz como del sistema de permisos, con el fin de simplificar la estructura de usuarios y centrar la gestión en los tres perfiles principales: **Directora o Director**, **Tutor** y **Polidocente**.

Esta decisión permitió optimizar los flujos de trabajo, reducir la complejidad del sistema y

concentrar los esfuerzos en las funcionalidades directamente relacionadas con la gestión académica.

2. Redefinición de los módulos académicos

El módulo de **Notas Mensuales** fue rediseñado para permitir una configuración flexible por parte del tutor, quien ahora puede definir los rubros de evaluación, así como la cantidad y tipo de notas según el curso, sección y mes.

Asimismo, se implementó el **Examen Bimestral** como un registro independiente dentro del sistema, estableciéndose la fórmula de cálculo bimestral:

(Promedio Mensual + Examen Bimestral) / 2, con redondeo a dos decimales.

Este cambio permitió obtener resultados más precisos, trazables y acordes con las reglas académicas de la institución.

3. División y mejora del módulo de libretas académicas

El módulo de **Libretas Académicas** fue reorganizado en dos submódulos con propósitos específicos:

- **Libretas Bimestrales (PDF):** orientadas a la generación de reportes por nivel educativo (Inicial, Primaria y Secundaria) con formato estandarizado para impresión.
- **Libretas UGEL (Excel):** diseñadas para consolidar las notas de los cuatro bimestres (B1–B4), integrando la **calculadora de equivalencias numéricas a letras** y un **catálogo editable de comentarios** por grado, curso y letra.

Esta separación permitió una mayor claridad funcional, una salida de datos más ordenada y una mejor adaptación a los formatos oficiales solicitados por la UGEL.

4. Ajustes en la estructura de la base de datos

La base de datos en **PostgreSQL** fue modificada para adaptarse a las nuevas necesidades del sistema. Se eliminaron las tablas relacionadas con el módulo de asistencia y se añadieron nuevas entidades que ofrecen una gestión más detallada y flexible de los registros académicos. Entre las tablas incorporadas destacan: *Nota*, *BoletaDetalle*, *Boleta*, *GradoTrabajado* y *AsignaturaTrabajada*.

Estas estructuras garantizan la integridad referencial, facilitan la consolidación de datos y permiten una trazabilidad completa del proceso de evaluación.

5. Optimización del flujo de trabajo docente

Se redefinió el flujo de trabajo entre los roles **Polidocente** y **Tutor**, estableciendo un proceso colaborativo y controlado. En este nuevo esquema, los polidocentes registran las calificaciones correspondientes a sus cursos y las envían al tutor, quien revisa, valida y finalmente cierra el mes.

Este procedimiento asegura la coherencia de la información antes del cálculo bimestral, evita duplicidades y refuerza la trazabilidad de los datos académicos.

6. Afinamientos técnicos y de interfaz

A nivel técnico, se realizaron mejoras tanto en el backend como en el frontend.

En el **backend (Django)**, se fortalecieron los mecanismos de autenticación y autorización mediante la implementación de *middlewares* y *guards* por rol. Además, se incorporaron auditorías básicas para registrar acciones críticas, como el cierre de mes o la exportación de reportes.

En el **frontend (React)**, se mejoró la experiencia de usuario al implementar validaciones en tiempo real, tablas dinámicas con cabeceras fijas, mensajes de error descriptivos y un diseño accesible basado en la tipografía institucional. Estas mejoras aumentan la eficiencia de uso y la accesibilidad general del sistema.

7. Consolidación técnica y estado actual del proyecto

El proyecto mantiene su **arquitectura en cuatro capas (Django, React, PostgreSQL y CI/CD)**, lo que asegura una separación clara entre las capas de presentación, lógica de negocio, persistencia y despliegue continuo.

En la actualidad, el sistema presenta un **avance global estimado del 80 %**, destacando la estabilidad funcional de los módulos, la integración exitosa entre el frontend y el backend, y la ejecución de pruebas unitarias y funcionales que garantizan la calidad del software desarrollado.

IV. Arquitectura y Stack

Se ha seleccionado una arquitectura tecnológica basada en el Patrón de 4 capas. Esta decisión se fundamenta en la necesidad de una organización clara del código y un mantenimiento sencillo, lo cual es crucial para facilitar el trabajo en equipo. Este patrón permite segmentar la aplicación en partes manejables y ofrece flexibilidad para futuras adaptaciones, como cambiar la interfaz de usuario sin afectar la lógica del negocio.

El stack tecnológico seleccionado combina Python y Javascript. Python, conocido por su sintaxis clara, se utiliza para el backend mediante el framework Django. Django fue elegido porque trae muchos componentes listos facilitando la gestión de usuarios y la generación de reportes. Para el frontend, se utiliza Javascript con la librería React.js, permitiendo la creación de interfaces de usuario dinámicas y adaptables.

La estructura de 4 capas del proyecto se define de la siguiente manera:

- **Capa de Presentación (UI):** Es la interfaz de usuario que verán los distintos perfiles (directora, docente). Se implementará con React.js.
- **Capa de Lógica de Negocio:** Contiene las reglas del sistema, como el cálculo de promedios o la conversión de notas. Esta lógica se manejará con Python y Django.

- **Capa de Acceso a Datos (DAL):** Se encarga de la comunicación directa con la base de datos , gestionando operaciones como guardar, leer y actualizar información.
- **Capa de Base de Datos**: Es el sistema físico donde se almacenan todas las tablas (alumnos, profesores, notas, etc.). El equipo utilizará PostgreSQL para esta capa, reemplazando la decisión inicial de MySQL.

V. Desarrollo de Aplicaciones

1. Módulo 1: React UI

Módulo principal — Directora

Este componente representa la vista central del **rol Directora/Director**, donde se agrupan los accesos a los diferentes módulos administrativos:

- Gestión de Usuarios
- Gestión UGEL (cargar/asignar/exportar)
- Reportes Generales
- Libretas Bimestrales (solo lectura)

A nivel técnico, el DirectorDashboard organiza su contenido en tarjetas (Card) y botones (Button) reutilizables, dispuestos en una rejilla (grid) adaptable al tamaño de la pantalla.
Cada módulo se encapsula como un **subcomponente o vista secundaria**, permitiendo mantener el código ordenado y fácil de extender.

Los elementos visuales se diseñaron para transmitir jerarquía y claridad:

- Colores neutros institucionales.
- Íconos de soporte visual (importados de lucide-react).

- Espaciados amplios para evitar sobrecarga de información.

Esto facilita la navegación y permite que el director/a identifique rápidamente cada función disponible sin confusión.

```jsx
import { FileSpreadsheet, Users, FileText, Eye } from 'lucide-react';
import { Card, CardContent, CardHeader, CardTitle } from '../ui/card';
import { Button } from '../ui/button';
import { Badge } from '../ui/badge';
import {
  Table,
  TableBody,
  TableCell,
  TableHead,
  TableHeader,
  TableRow,
} from '../ui/table';
import { mockSecciones, mockEstadosSecciones, mockUsers } from '../../lib/mock-data';

export function DirectorDashboard({ onNavigate }) {
  const getEstadoBadge = (estado) => {
    const variants = {
      'abierto': { variant: 'default', label: 'Abierto' },
      'en-revision': { variant: 'secondary', label: 'En revisión' },
      'cerrado': { variant: 'outline', label: 'Cerrado' }
    };
    const config = variants[estado] || variants['abierto'];
    return (
      <Badge variant={config.variant} className={estado === 'abierto' ? 'bg-success' : ''}>
        {config.label}
      </Badge>
    );
  };

  return (
    <div className="p-8 space-y-8">
      <div>
        <h1 className="mb-2">Panel de Control</h1>
        <p className="text-muted-foreground">Vista general del sistema académico</p>
      </div>
```

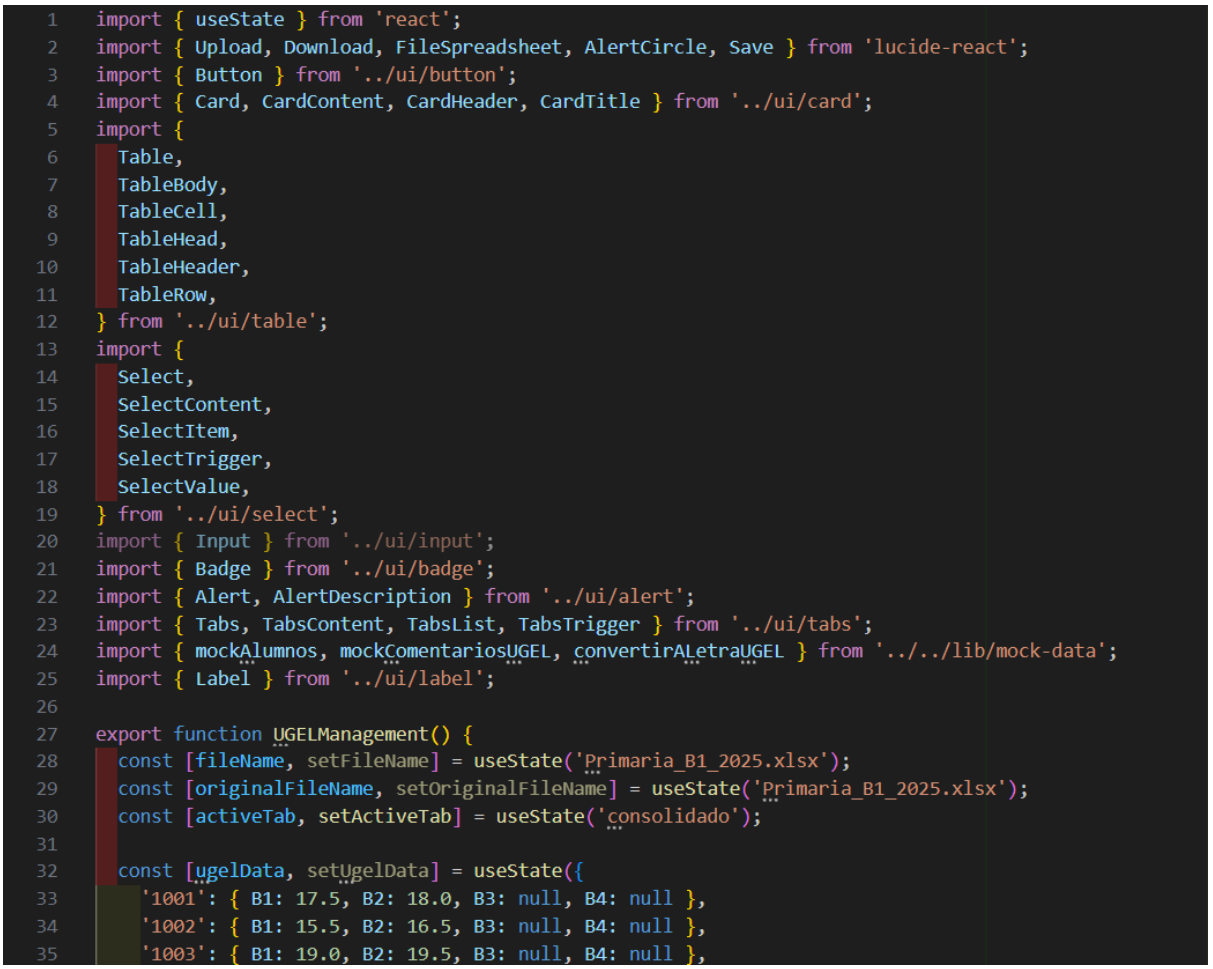
Módulo de Gestión UGEL

El módulo de UGEL fue diseñado para que el director/a pueda subir archivos Excel (.xlsx), revisar la información procesada y exportarla nuevamente con los datos corregidos.

Se utiliza una zona de carga tipo “dropzone” donde el usuario arrastra el archivo y el sistema muestra una vista previa del contenido.

Desde el lado frontend, este componente prioriza la **experiencia visual de interacción**:

- Bordes punteados para el área de carga.
- Mensajes dinámicos según el estado (sin archivo, cargando, archivo listo).
- Tabla resumen del contenido detectado.



```tsx
import { useState } from 'react';
import { Upload, Download, FileSpreadsheet, AlertCircle, Save } from 'lucide-react';
import { Button } from '../ui/button';
import { Card, CardContent, CardHeader, CardTitle } from '../ui/card';
import {
  Table,
  TableBody,
  TableCell,
  TableHead,
  TableHeader,
  TableRow,
} from '../ui/table';
import {
  Select,
  SelectContent,
  SelectItem,
  SelectTrigger,
  SelectValue,
} from '../ui/select';
import { Input } from '../ui/input';
import { Badge } from '../ui/badge';
import { Alert, AlertDescription } from '../ui/alert';
import { Tabs, TabsContent, TabsList, TabsTrigger } from '../ui/tabs';
import { mockAlumnos, mockComentariosUGEL, convertirALetraUGEL } from '../../lib/mock-data';
import { Label } from '../ui/label';

export function UGELManagement() {
  const [fileName, setFileName] = useState('Primaria_B1_2025.xlsx');
  const [originalFileName, setOriginalFileName] = useState('Primaria_B1_2025.xlsx');
  const [activeTab, setActiveTab] = useState('consolidado');

  const [ugelData, setUgelData] = useState({
    '1001': { B1: 17.5, B2: 18.0, B3: null, B4: null },
    '1002': { B1: 15.5, B2: 16.5, B3: null, B4: null },
    '1003': { B1: 19.0, B2: 19.5, B3: null, B4: null },
```

Rol Tutor/a

El **Tutor/a** tiene un rol clave en la gestión de calificaciones mensuales.

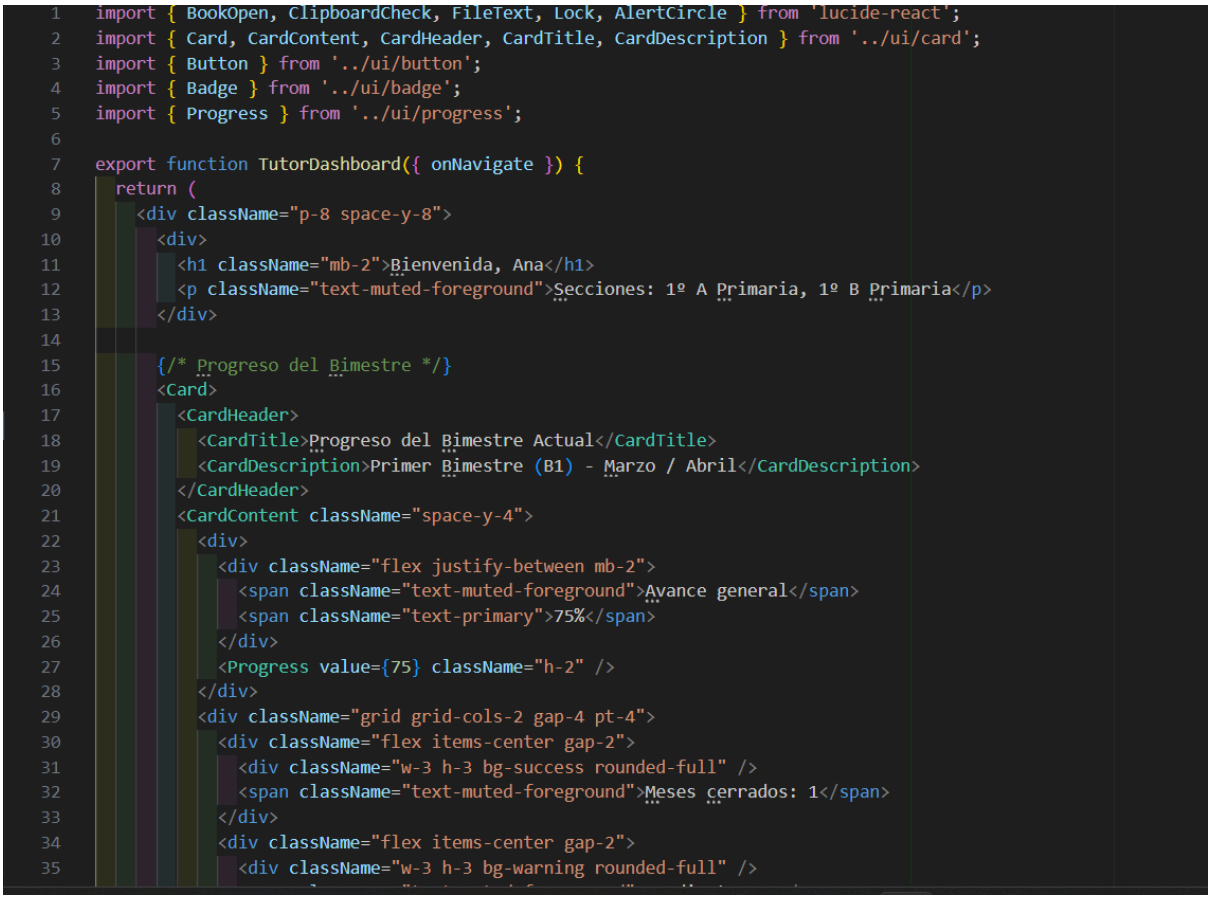
Su interfaz combina distintos flujos dentro de un mismo módulo, con una navegación simple entre ellos:

- **Configuración de Rubros:** selección de categorías y ponderaciones (T/P).
- **Registro de Notas Mensuales:** tabla editable con alumnos × rubros.
- **Examen Bimestral:** tabla independiente con la fórmula 50–50 (mensual + examen).

- **Libretas Bimestrales:** checklist con validaciones previas y botón para “Generar PDF”.

El frontend se centró en optimizar la **usabilidad de las tablas**:

- Campos numéricos validados (0–20, con 2 decimales).
- Promedios automáticos actualizados en tiempo real.
- Estado visual de cierre (“candado”) al finalizar el mes.



```
import { BookOpen, ClipboardCheck, FileText, Lock, AlertCircle } from 'lucide-react';
import { Card, CardContent, CardHeader, CardTitle, CardDescription } from '../ui/card';
import { Button } from '../ui/button';
import { Badge } from '../ui/badge';
import { Progress } from '../ui/progress';

export function TutorDashboard({ onNavigate }) {
  return (
    <div className="p-8 space-y-8">
      <div>
        <h1 className="mb-2">Bienvenida, Ana</h1>
        <p className="text-muted-foreground">Secciones: 1º A Primaria, 1º B Primaria</p>
      </div>

      {/* Progreso del Bimestre */}
      <Card>
        <CardHeader>
          <CardTitle>Progreso del Bimestre Actual</CardTitle>
          <CardDescription>Primer Bimestre (B1) - Marzo / Abril</CardDescription>
        </CardHeader>
        <CardContent className="space-y-4">
          <div>
            <div className="flex justify-between mb-2">
              <span className="text-muted-foreground">Avance general</span>
              <span className="text-primary">75%</span>
            </div>
            <Progress value={75} className="h-2" />
          </div>
          <div className="grid grid-cols-2 gap-4 pt-4">
            <div className="flex items-center gap-2">
              <div className="w-3 h-3 bg-success rounded-full" />
              <span className="text-muted-foreground">Meses cerrados: 1</span>
            </div>
            <div className="flex items-center gap-2">
              <div className="w-3 h-3 bg-warning rounded-full" />
```

Rol Polidocente

El **Polidocente** maneja únicamente los **rubros asignados** por el tutor, por lo que su interfaz es más acotada, enfocada en la carga rápida de notas y el control de envíos.

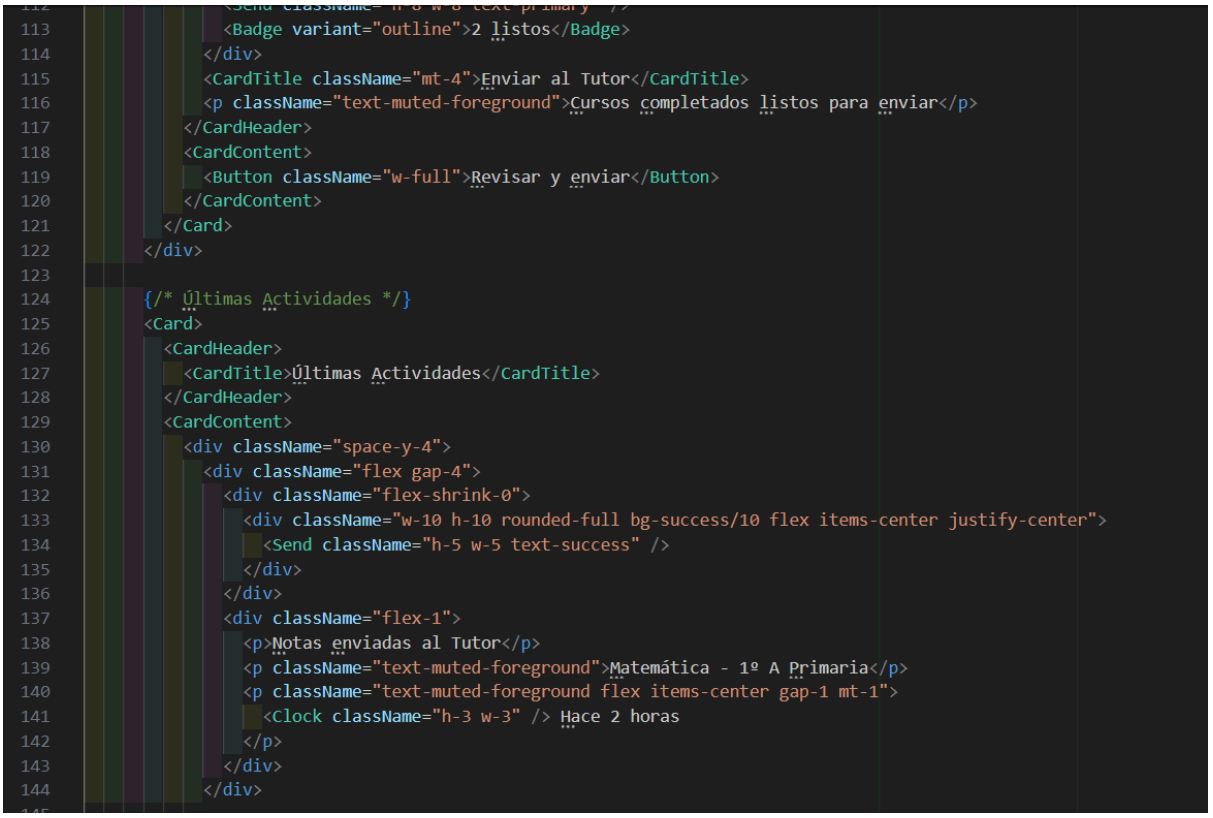
La pantalla presenta:

- Una **tabla simplificada** con alumnos y rubros.
- Botón principal **“Enviar al Tutor”**.
- Línea de tiempo (timeline) visual con estados: **Borrador → Enviado**.

El diseño enfatiza la **claridad visual y la retroalimentación inmediata**:

Los botones cambian de estado según la acción, y los banners de notificación confirman los envíos realizados.

Esto ayuda al docente a evitar errores y mantener comunicación visual con el tutor sobre el estado de sus registros.



```
            <Badge variant="outline">2 listos</Badge>
          </div>
          <CardTitle className="mt-4">Enviar al Tutor</CardTitle>
          <p className="text-muted-foreground">Cursos completados listos para enviar</p>
        </CardHeader>
        <CardContent>
          <Button className="w-full">Revisar y enviar</Button>
        </CardContent>
      </Card>
    </div>

    {/* Últimas Actividades */}
    <Card>
      <CardHeader>
        <CardTitle>Últimas Actividades</CardTitle>
      </CardHeader>
      <CardContent>
        <div className="space-y-4">
          <div className="flex gap-4">
            <div className="flex-shrink-0">
              <div className="w-10 h-10 rounded-full bg-success/10 flex items-center justify-center">
                <Send className="h-5 w-5 text-success" />
              </div>
            </div>
            <div className="flex-1">
              <p>Notas enviadas al Tutor</p>
              <p className="text-muted-foreground">Matemática - 1º A Primaria</p>
              <p className="text-muted-foreground flex items-center gap-1 mt-1">
                <Clock className="h-3 w-3" /> Hace 2 horas
              </p>
            </div>
          </div>
```

Design System y Componentes Base

Este módulo permite al director/a **visualizar reportes consolidados** y **consultar libretas bimestrales** en formato PDF.

En el frontend, se implementaron componentes de tabla con cabeceras fijas, scroll interno y botones de acción que simulan la descarga del reporte.

Los elementos se adaptan automáticamente al tamaño de la pantalla, manteniendo legibilidad en pantallas pequeñas.

Los botones de acción están diseñados con estados visuales (hover, disabled, loading) para mejorar la comprensión del usuario durante la interacción.

Técnicamente, este módulo combina:

- Componentes Table, Button y Card.
- Manejo condicional de clases mediante cn() para aplicar estilos de error, carga o éxito.
- Mensajes visuales de confirmación con toasts o banners.

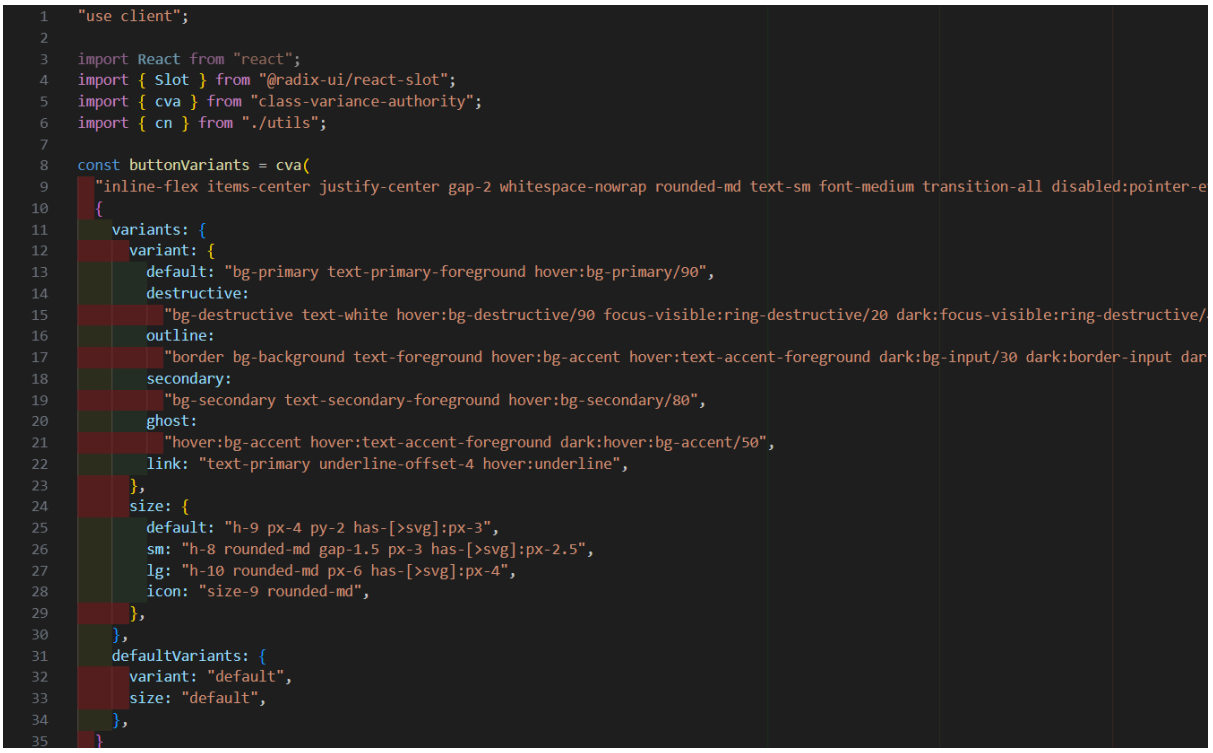
```tsx
"use client";

import { useState } from "react";
import { FileText, Download, CheckCircle, Loader2, ArrowLeft, Eye } from "lucide-react";
import { Button } from "../ui/button";
import { Card, CardContent, CardHeader, CardTitle } from "../ui/card";
import {
  Select,
  SelectContent,
  SelectItem,
  SelectTrigger,
  SelectValue,
} from "../ui/select";
import { Badge } from "../ui/badge";
import { Alert, AlertDescription } from "../ui/alert";
import { mockAlumnos, calcularNotaBimestral } from "../../lib/mock-data";

export function ReportCards({ onBack }) {
  const [isGenerating, setIsGenerating] = useState(false);
  const [pdfGenerated, setPdfGenerated] = useState(false);
  const [selectedAlumno, setSelectedAlumno] = useState("1001");
  const [showPreview, setShowPreview] = useState(false);

  // Datos de ejemplo para la vista previa
  const promediosMensuales = {
    "1001": 17.14,
    "1002": 16.43,
    "1003": 19.29,
    "1004": 14.57,
    "1005": 17.86,
  };
```

Componentes reutilizables del design system

Para mantener la coherencia visual, se construyó un pequeño **design system** con componentes reutilizables:

- **Input.jsx**: campos de texto con validación visual (error, focus, disabled).
- **Button.jsx**: botones estilizados según el nivel de acción (primary, secondary, danger).
- **Card.jsx**: contenedores con sombras y bordes redondeados, usados para agrupar contenido.

```jsx
import React from "react";
import { cn } from "./utils";

export function Input({ className = "", type = "text", ...props }) {
  return (
    <input
      type={type}
      data-slot="input"
      className={cn(
        "file:text-foreground placeholder:text-muted-foreground selection:bg-primary selection:text-primary-foreground dark:bg-inp
        "focus-visible:border-ring focus-visible:ring-ring/50 focus-visible:ring-[3px]",
        "aria-invalid:ring-destructive/20 dark:aria-invalid:ring-destructive/40 aria-invalid:border-destructive",
        className
      )}
      {...props}
    />
  );
}
```



```jsx
"use client";

import React from "react";
import { Slot } from "@radix-ui/react-slot";
import { cva } from "class-variance-authority";
import { cn } from "./utils";

const buttonVariants = cva(
  "inline-flex items-center justify-center gap-2 whitespace-nowrap rounded-md text-sm font-medium transition-all disabled:pointer-e
  {
    variants: {
      variant: {
        default: "bg-primary text-primary-foreground hover:bg-primary/90",
        destructive:
          "bg-destructive text-white hover:bg-destructive/90 focus-visible:ring-destructive/20 dark:focus-visible:ring-destructive/
        outline:
          "border bg-background text-foreground hover:bg-accent hover:text-accent-foreground dark:bg-input/30 dark:border-input dar
        secondary:
          "bg-secondary text-secondary-foreground hover:bg-secondary/80",
        ghost:
          "hover:bg-accent hover:text-accent-foreground dark:hover:bg-accent/50",
        link: "text-primary underline-offset-4 hover:underline",
      },
      size: {
        default: "h-9 px-4 py-2 has-[>svg]:px-3",
        sm: "h-8 rounded-md gap-1.5 px-3 has-[>svg]:px-2.5",
        lg: "h-10 rounded-md px-6 has-[>svg]:px-4",
        icon: "size-9 rounded-md",
      },
    },
    defaultVariants: {
      variant: "default",
      size: "default",
    },
  }
```

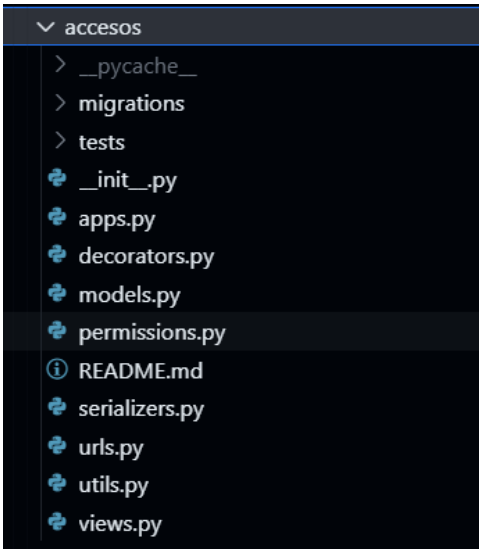
```jsx
import React from "react";
import { cn } from "./utils";

export function Card({ className, ...props }) {
  return (
    <div
      data-slot="card"
      className={cn(
        "bg-card text-card-foreground flex flex-col gap-6 rounded-xl border",
        className
      )}
      {...props}
    />
  );
}

export function CardHeader({ className, ...props }) {
  return (
    <div
      data-slot="card-header"
      className={cn(
        "@container/card-header grid auto-rows-min grid-rows-[auto_auto] items-start gap-1.5 px-6 pt-6 has-data-[slot=card-action]:
        className
      )}
      {...props}
    />
  );
}

export function CardTitle({ className, ...props }) {
  return (
    <h4
      data-slot="card-title"
      className={cn("leading-none", className)}
      {...props}
```

2. Módulo 2: Backend Accesos y Permisos

En términos generales, el módulo cubre la autenticación y autorización por rol, la seguridad de rutas y APIs, la delegación operativa (UGEL), el cierre de períodos y una auditoría mínima que registra acciones críticas. Se trató de mantener el código claro y reutilizable, de modo que otras partes del sistema (por ejemplo el módulo de libretas) puedan invocar permisos y registros de auditoría sin duplicar lógica.

La implementación se organizó en una única aplicación Django ubicada en backend/apps/accesos/, cuyas piezas principales son modelos de soporte, permisos reutilizables, vistas/endereços REST y decoradores para reglas transversales. En el diseño se priorizó la simplicidad para la fase actual: las decisiones apuntan a ofrecer controles por rol (Directora/Director, Tutor, Polidocente y roles afines), a permitir delegaciones puntuales (cuando la Directora delega la gestión UGEL a un Tutor), y a asegurar que operaciones

irreversibles —como el cierre de un periodo académico— queden registradas y sean rechazadas si se intenta repetir la operación.



Los modelos de datos reflejan esa filosofía: existe una entidad de auditoría que conserva el actor, la acción, el tipo y el identificador del recurso afectado y un campo de metadatos en JSON para extensiones futuras; existe un modelo de delegación que expresa que un Tutor está autorizado temporalmente para actuar en nombre de la Directora; hay una tabla de tokens mock para pruebas de recuperación de contraseña y, para facilitar la demo de cierre de periodo, un modelo de Period con un flag closed. Esta base de datos relacional permite hoy utilizar SQLite en desarrollo y mantener la puerta abierta para migrar a PostgreSQL en producción.

```python
backend > apps > accesos > models.py > ...
from django.db import models
from django.conf import settings
from django.utils import timezone
import uuid


class AuditLog(models.Model):
    actor = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, null=True, related_name='audit_actions')
    action = models.CharField(max_length=100)
    target_type = models.CharField(max_length=100, null=True, blank=True)
    target_id = models.CharField(max_length=100, null=True, blank=True)
    timestamp = models.DateTimeField(default=timezone.now)
    metadata = models.JSONField(null=True, blank=True)

    def __str__(self):
        return f"{self.timestamp} - {self.actor} - {self.action}"


class UGELDelegation(models.Model):
    tutor = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='ugel_delegations')
    delegated_by = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, null=True, related_name='delegations_made')
    active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    expires_at = models.DateTimeField(null=True, blank=True)

    def __str__(self):
        return f"UGEL Delegation: {self.tutor} active={self.active}"
```

```python
class PasswordResetMock(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='password_reset_tokens')
    token = models.UUIDField(default=uuid.uuid4, editable=False, unique=True)
    created_at = models.DateTimeField(auto_now_add=True)
    expires_at = models.DateTimeField(null=True, blank=True)
    used = models.BooleanField(default=False)

    def __str__(self):
        return f"PasswordResetMock {self.user} used={self.used}"


# Modelo demo para Periodos (Mes/Bimestre)
class Period(models.Model):
    name = models.CharField(max_length=200)
    closed = models.BooleanField(default=False)
    closed_by = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, null=True, blank=True, related_name='periods_closed')
    closed_at = models.DateTimeField(null=True, blank=True)

    def __str__(self):
        return f"Period {self.name} closed={self.closed}"
```

La capa de autorización se implementó con dos piezas clave: una utilidad para resolver el rol de un usuario y permisos específicos de Django REST Framework que aplican las reglas. La función encargada de resolver el rol examina tres fuentes posibles en orden: un atributo role directo en el usuario, un atributo profile.role y finalmente el primer Group al que pertenezca el usuario. Sobre esta fuente de verdad flexible se construyeron dos permisos: uno genérico que permite a una vista declarar una lista allowed_roles y aceptar solo usuarios cuyo rol esté en esa lista, y otro permiso específico para la gestión UGEL que autoriza siempre a la Directora y, además, autoriza a un Tutor solo si existe una delegación activa asociada a su cuenta. Esta aproximación por permisos en las vistas hace que la protección quede localizada y reutilizable por otras APIs del proyecto.

backend > apps > accesos > utils.py

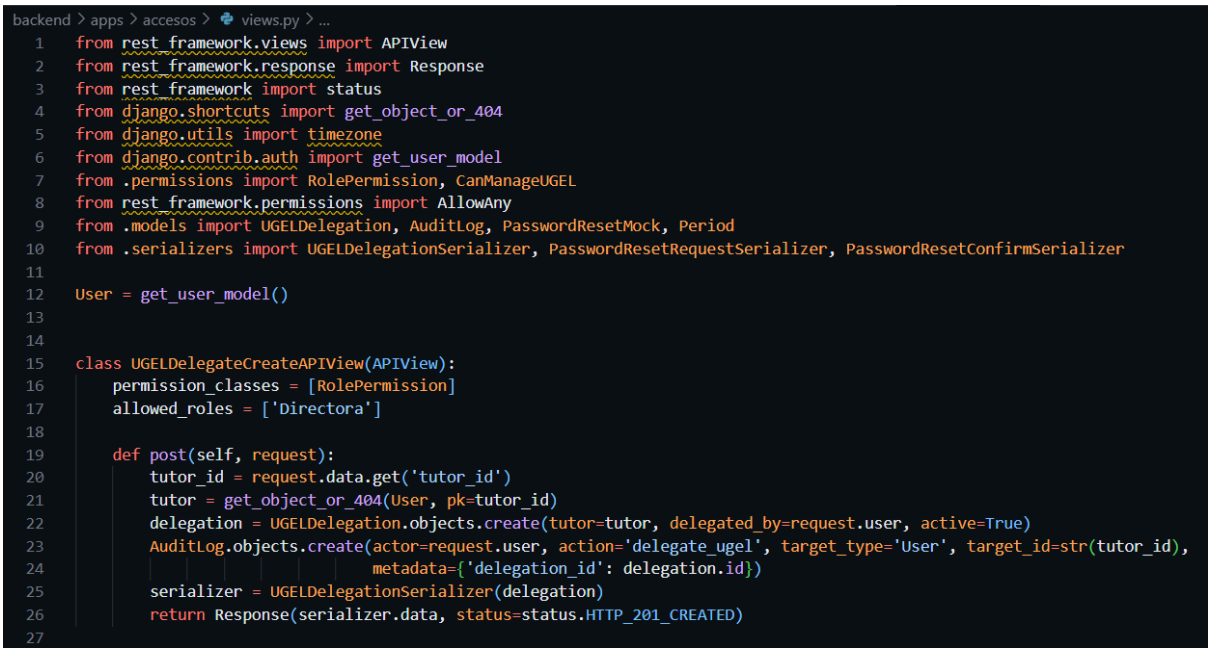
```python
from django.contrib.auth.models import Group


def get_user_role(user):
    """
    Intentar obtener rol del usuario de forma flexible:
    - user.role
    - user.profile.role
    - primer group de Django
    """
    if not user or user.is_anonymous:
        return None
    role = getattr(user, 'role', None)
    if role:
        return role
    profile = getattr(user, 'profile', None)
    if profile:
        role = getattr(profile, 'role', None)
        if role:
            return role
    groups = user.groups.all()
    if groups.exists():
        return groups.first().name
    return None
```

backend > apps > accesos > permissions.py > ...

```python
from rest_framework.permissions import BasePermission
from .utils import get_user_role
from .models import UGELDelegation


class RolePermission(BasePermission):
    """
    Permite acceso solo si request.user tiene un rol dentro de view.allowed_roles (lista).
    Uso: en la view defina allowed_roles = ['Directora','Coordinador', ...]
    """
    def has_permission(self, request, view):
        allowed = getattr(view, 'allowed_roles', None)
        if not allowed:
            return True
        role = get_user_role(request.user)
        return role in allowed


class CanManageUGEL(BasePermission):
    """
    Directora puede siempre; tutor si existe delegación activa.
    """
    def has_permission(self, request, view):
        role = get_user_role(request.user)
        if role == 'Directora':
            return True
        return UGELDelegation.objects.filter(tutor=request.user, active=True).exists()
```

En las vistas REST se implementaron los flujos operativos necesarios para la demo funcional: la Directora puede delegar a un Tutor mediante un endpoint que crea la delegación y registra la operación en la tabla de auditoría; hay un endpoint de verificación de acceso UGEL que

responde con un 200 si la persona tiene permiso para gestionar UGEL (útil para la UI); existe un endpoint para cerrar periodos que marca el periodo como cerrado, guarda el usuario que cerró y el timestamp y retorna HTTP 409 si se intenta cerrar un periodo ya cerrado; y existe un endpoint de export mock que registra la acción de exportación en la auditoría. Adicionalmente, para facilitar pruebas de integración sin infraestructura de correo, se implementó un flujo de recuperación de contraseña en modo mock: se crea un token en la tabla PasswordResetMock y se devuelve ese token en la respuesta HTTP; con el token se puede confirmar el cambio de contraseña llamando al endpoint de confirmación. Este flujo es intencionadamente sencillo para pruebas y se documenta como tal; en producción debe sustituirse por el mecanismo estándar de Django combinado con envío de correo.



```python
class UGELManageAPIView(APIView):
    permission_classes = [CanManageUGEL]

    def get(self, request):
        return Response({'detail': 'Acceso UGEL concedido'}, status=status.HTTP_200_OK)


class ClosePeriodAPIView(APIView):
    permission_classes = [RolePermission]
    allowed_roles = ['Directora', 'Coordinador']

    def post(self, request, period_id):
        period = get_object_or_404(Period, pk=period_id)
        if getattr(period, 'closed', False):
            return Response({'detail': 'Periodo ya cerrado.'}, status=status.HTTP_409_CONFLICT)
        period.closed = True
        period.closed_by = request.user
        period.closed_at = timezone.now()
        period.save()
        AuditLog.objects.create(actor=request.user, action='close_period', target_type='Period', target_id=str(period_id))
        return Response({'detail': 'Periodo cerrado.'}, status=status.HTTP_200_OK)

```

```python
class ExportPDFMockAPIView(APIView):
    permission_classes = [RolePermission]
    allowed_roles = ['Directora', 'Coordinador', 'Tutor', 'Docente']

    def get(self, request, report_id=None):
        AuditLog.objects.create(actor=request.user, action='export_pdf', target_type='Report', target_id=str(report_id))
        return Response({'detail': 'PDF generado (mock).'}, status=status.HTTP_200_OK)


class PasswordResetRequestAPIView(APIView):
    permission_classes = [AllowAny]
    def post(self, request):
        ser = PasswordResetRequestSerializer(data=request.data)
        ser.is_valid(raise_exception=True)
        email = ser.validated_data['email']
        try:
            user = User.objects.get(email=email)
        except User.DoesNotExist:
            return Response({'detail': 'Usuario no encontrado.'}, status=status.HTTP_404_NOT_FOUND)
        token = PasswordResetMock.objects.create(user=user, expires_at=timezone.now() + timezone.timedelta(hours=1))
        return Response({'detail': 'Token creado (mock).', 'token': str(token.token)}, status=status.HTTP_201_CREATED)

```

```python
class PasswordResetConfirmAPIView(APIView):
    permission_classes = [AllowAny]
    def post(self, request):
        ser = PasswordResetConfirmSerializer(data=request.data)
        ser.is_valid(raise_exception=True)
        token = ser.validated_data['token']
        new_password = ser.validated_data['password']
        pr = get_object_or_404(PasswordResetMock, token=token, used=False)
        if pr.expires_at and pr.expires_at < timezone.now():
            return Response({'detail': 'Token expirado.'}, status=status.HTTP_400_BAD_REQUEST)
        user = pr.user
        user.set_password(new_password)
        user.save()
        pr.used = True
        pr.save()
        AuditLog.objects.create(actor=user, action='password_reset', target_type='User', target_id=str(user.pk))
        return Response({'detail': 'Contraseña actualizada (mock).'}, status=status.HTTP_200_OK)
```

La suite de pruebas acompaña los flujos principales: los tests automáticos comprueban que la Directora puede crear delegaciones, que un Tutor sin delegación recibe 403 y que con delegación puede acceder, que el cierre de periodo responde 200 la primera vez y 409 si se repite, y que el flujo mock de password reset genera token y permite la confirmación de la

contraseña. Estas pruebas ofrecen una cobertura funcional de los escenarios de uso más relevantes y sirven de contrato para futuras refactorizaciones.

Desde la perspectiva de seguridad operativa, la implementación actual cubre el control de acceso a nivel de vista y la trazabilidad de acciones críticas vía AuditLog, lo que ya posibilita auditorías básicas. Lo que no se incluyó en esta fase —por decisión de alcance y para priorizar entregables— fue un middleware global de seguridad que aplique políticas transversales (por ejemplo logging de IP y user agent para todos los requests, rate limiting o bloqueo por intentos fallidos). Tampoco está integrado el envío de correo para password reset, y por tanto el flujo que existe es un mock pensado para pruebas.

3. Módulo 3: Backend Notas Mensuales + Cierre + Examen Bimestral

Este módulo de backend, desarrollado en Django Rest Framework (DRF), es fundamental para la gestión académica del sistema de intranet. Su objetivo principal es digitalizar y automatizar el flujo de trabajo de la evaluación docente, cubriendo desde la **configuración flexible de rubros** hasta el **cálculo automatizado de promedios** (simple y 50-50), y el control estricto del **cierre mensual y bimestral**.

El desarrollo se enfocó en tres fases clave: Modelos (Estructura de Datos), Lógica de Negocio (Cálculo) y Desarrollo de APIs (Comunicación).

Fase 1: Estructura de Datos (Modelos)

La Fase 1 se centró en la creación de los modelos de Django (models.py) necesarios para almacenar los datos de evaluación con precisión y control de flujo. Se diseñaron cuatro modelos principales para manejar los requisitos del sistema:

Modelo	Propósito Principal	Campos Clave
Rubro	Permite a los Polidocentes configurar los criterios de evaluación (ej. Tareas, Participación).	nombre
NotaMensual	Almacena cada calificación individual (0.00 a 20.00).	calificacion (DecimalField), rubro, mes
ExamenBimestral	Guarda la nota del examen (50% del promedio bimestral).	calificacion (DecimalField), bimestre
EstadoCierreMensual	Controla el flujo de trabajo: ABIERTO, REVISION o CERRADO (bloqueo).	estado, seccion, mes

backend/models.py

backend > models.py

```python
from django.db import models
from django.db.models import DecimalField, PROTECT, SET_NULL

# NOTA: Se asume que los modelos base (Alumno, Curso, Profesor, Seccion)
# ya están definidos y disponibles en el ORM de Django para las ForeignKey.

# --- 1. Rubro: Define los rubros configurables (ej. Cuaderno, Tareas) ---
# Permite al Tutor configurar rubros.
class Rubro(models.Model):
    nombre = models.CharField(max_length=50, unique=True)

    def __str__(self):
        return self.nombre

# --- 2. NotaMensual: Registro detallado de la calificación ---
# Almacena cada nota (0-20) con precisión de 2 decimales.
class NotaMensual(models.Model):
    alumno = models.ForeignKey('Alumno', on_delete=models.CASCADE)
    curso = models.ForeignKey('Curso', on_delete=models.CASCADE)
    rubro = models.ForeignKey(Rubro, on_delete=PROTECT)

    # max_digits=4 (ej. 20.00), decimal_places=2 (para precisión)
    calificacion = DecimalField(max_digits=4, decimal_places=2)
    mes = models.IntegerField()

    registrado_por = models.ForeignKey('Profesor', on_delete=SET_NULL, null=True)
    fecha_registro = models.DateTimeField(auto_now_add=True)

    class Meta:
        # Asegura que un alumno no tenga dos notas para el mismo rubro en el mismo curso/mes
        unique_together = ('alumno', 'curso', 'rubro', 'mes')

# --- 3. ExamenBimestral: Nota separada para el cálculo 50-50 ---
# Registro separado del examen bimestral.
class ExamenBimestral(models.Model):
    alumno = models.ForeignKey('Alumno', on_delete=models.CASCADE)
    curso = models.ForeignKey('Curso', on_delete=models.CASCADE)
    bimestre = models.IntegerField()

    # Nota del examen que usará la regla bimestral 50-50.
    calificacion = DecimalField(max_digits=4, decimal_places=2)

    class Meta:
        # Un alumno solo puede tener una nota de examen por curso y bimestre
        unique_together = ('alumno', 'curso', 'bimestre')

# --- 4. EstadoCierreMensual: Gestión del flujo de envío y bloqueo ---
# Controla el flujo de cierre de mes.
ESTADOS_CIERRE = (
    ('ABIERTO', 'Abierto'),
    ('REVISION', 'En Revisión'),
    ('CERRADO', 'Cerrado'), # Indica bloqueo de edición
)

class EstadoCierreMensual(models.Model):
    seccion = models.ForeignKey('Seccion', on_delete=models.CASCADE)
    mes = models.IntegerField()

    # Estados de flujo: Abierto, En Revisión, Cerrado (bloqueado)
    estado = models.CharField(max_length=10, choices=ESTADOS_CIERRE, default='ABIERTO')

    fecha_cierre = models.DateTimeField(null=True, blank=True)

    class Meta:
        # Solo puede haber un estado de cierre por sección y mes
        unique_together = ('seccion', 'mes')
```

Se utilizó el campo DecimalField(max_digits=4, decimal_places=2) para la calificación de notas y exámenes, asegurando el cumplimiento del requerimiento de precisión de dos decimales en los promedios. Además, el modelo EstadoCierreMensual se diseñó con un campo estado basado en las opciones de flujo: **'ABIERTO'**, **'REVISION'** y **'CERRADO'**, para implementar el bloqueo de edición de notas una vez finalizado el proceso de revisión.

Fase 2: Lógica de Negocio (Servicios)

La Fase 2 se implementó en la capa de servicios (backend/services/grade_logic.py) para centralizar las reglas de cálculo de promedios y asegurar su correcto cumplimiento, independientemente del *endpoint* que lo consuma. Se utilizaron objetos Decimal y la función quantize de Python para garantizar la precisión decimal en los cálculos.

Funciones Implementadas

1. **calcular_promedio_mensual(notas):** Calcula el promedio simple de una lista de notas mensuales, redondeando a dos decimales.
2. **calcular_promedio_bimestral(promedio_mensual, examen_bimestral):** Implementa la regla principal del proyecto: el promedio bimestral resulta del cálculo **50% de las notas mensuales y 50% del examen bimestral**.

backend/grade_logic.py

backend > services > grade_logic.py

```python
from decimal import Decimal, ROUND_HALF_UP

def calcular_promedio_mensual(notas: list[Decimal]) -> Decimal:
    """
    Calcula el promedio simple de una lista de notas del mes.
    El resultado se redondea a 2 decimales.
    """
    if not notas:
        return Decimal('0.00')

    suma_notas = sum(notas)
    promedio = suma_notas / Decimal(len(notas))
    # Redondeo a 2 decimales (al medio hacia arriba)
    return promedio.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)


def calcular_promedio_bimestral(promedio_mensual: Decimal, examen_bimestral: Decimal) -> Decimal:
    """
    Aplica la Regla Bimestral 50-50: (Promedio Mensual + Examen Bimestral) / 2.
    El resultado se redondea a 2 decimales.
    """
    # Ambos promedios pesan 50%
    suma_total = promedio_mensual + examen_bimestral
    promedio_final = suma_total / Decimal(2)

    # Redondeo final a 2 decimales
    return promedio_final.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)
```

La función calcular_promedio_bimestral implementa la regla crítica del 50-50, donde se garantiza que la suma de promedio_mensual y examen_bimestral se divida por dos, y el resultado final se redondee a dos decimales utilizando el modo **ROUND_HALF_UP** (redondeo al medio hacia arriba), como es práctica estándar en la contabilidad y calificación.

Fase 3: Desarrollo de APIs y Endpoints

Esta fase vincula la lógica (Fase 2) y los modelos (Fase 1) con el frontend a través de *endpoints* RESTful, utilizando Django Rest Framework (DRF).

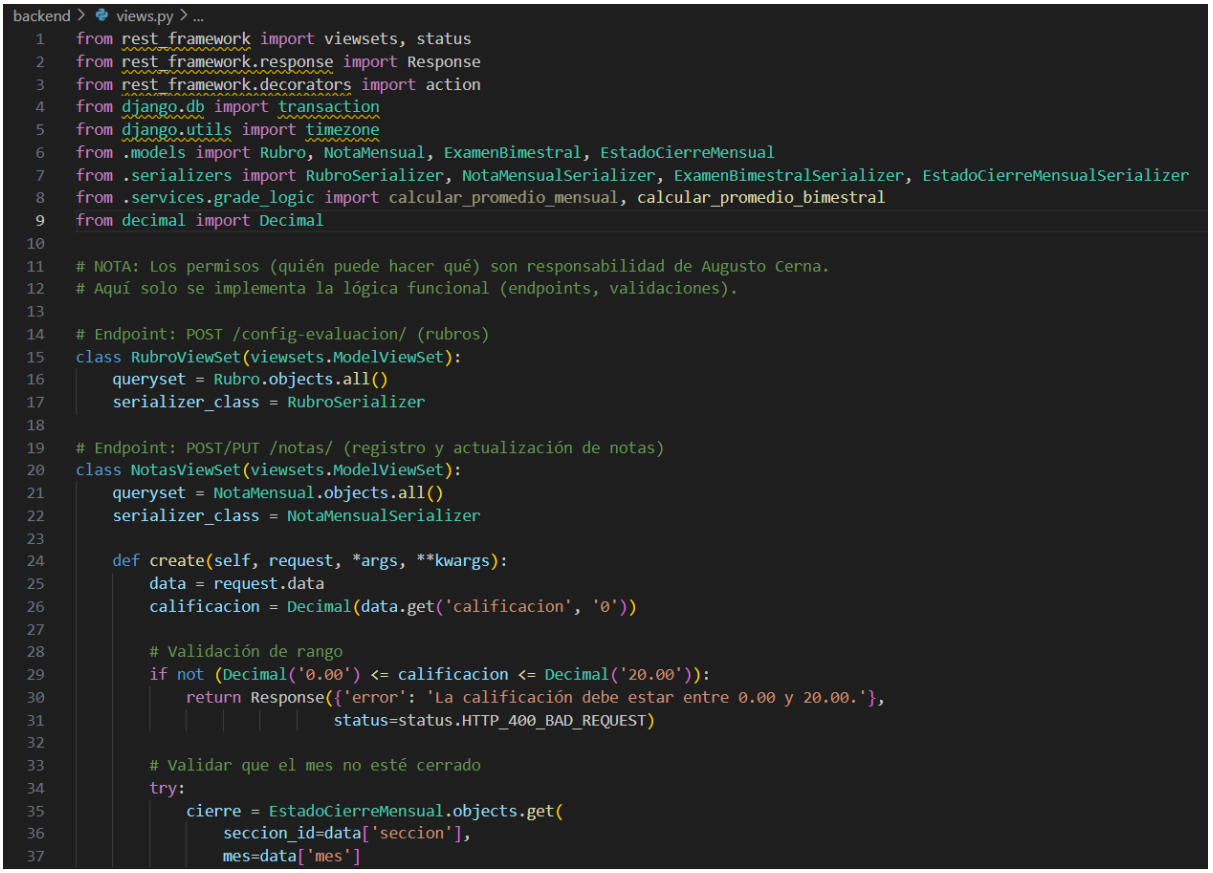
Archivos Clave

1. **serializers.py:** Define los **ModelSerializer** para cada entidad (ej. NotaMensualSerializer, RubroSerializer), permitiendo la traducción de objetos Python a JSON y viceversa.

2. **views.py:** Contiene los **ViewSet** que manejan las peticiones HTTP. Se implementaron validaciones de rango (0-20) y la lógica de bloqueo.
3. **urls.py:** Utiliza un DefaultRouter para mapear los *ViewSets* a las URLs RESTful (ej. /notas/, /cierre-mes/, /config-evaluacion/).

URL (Método)	Descripción	Validación/Lógica
/notas/ (POST)	Registro o actualización de notas individuales.	**Validación de rango (0.00-20.00)** y **Bloqueo por Cierre** (impide registrar si el mes está **'CERRADO'**).
/cierre-mes/cerrar (POST)	Función para bloquear la edición de notas en una sección/mes.	Actualiza el estado a **'CERRADO'** y registra la fecha de bloqueo.
/examen-bimestral/ (POST)	Registro de la nota del examen bimestral.	Prepara los datos para el cálculo final 50-50.

backend/views.py



```
backend > views.py > ...
from rest_framework import viewsets, status
from rest_framework.response import Response
from rest_framework.decorators import action
from django.db import transaction
from django.utils import timezone
from .models import Rubro, NotaMensual, ExamenBimestral, EstadoCierreMensual
from .serializers import RubroSerializer, NotaMensualSerializer, ExamenBimestralSerializer, EstadoCierreMensualSerializer
from .services.grade_logic import calcular_promedio_mensual, calcular_promedio_bimestral
from decimal import Decimal

# NOTA: Los permisos (quién puede hacer qué) son responsabilidad de Augusto Cerna.
# Aquí solo se implementa la lógica funcional (endpoints, validaciones).

# Endpoint: POST /config-evaluacion/ (rubros)
class RubroViewSet(viewsets.ModelViewSet):
    queryset = Rubro.objects.all()
    serializer_class = RubroSerializer

# Endpoint: POST/PUT /notas/ (registro y actualización de notas)
class NotasViewSet(viewsets.ModelViewSet):
    queryset = NotaMensual.objects.all()
    serializer_class = NotaMensualSerializer

    def create(self, request, *args, **kwargs):
        data = request.data
        calificacion = Decimal(data.get('calificacion', '0'))

        # Validación de rango
        if not (Decimal('0.00') <= calificacion <= Decimal('20.00')):
            return Response({'error': 'La calificación debe estar entre 0.00 y 20.00.'},
                            status=status.HTTP_400_BAD_REQUEST)

        # Validar que el mes no esté cerrado
        try:
            cierre = EstadoCierreMensual.objects.get(
                seccion_id=data['seccion'],
                mes=data['mes']
```

El NotasViewSet garantiza la integridad de los datos en el método create mediante una **validación estricta de la calificación entre 0.00 y 20.00** y consulta el EstadoCierreMensual para aplicar la restricción de bloqueo: si el estado es **'CERRADO'**, se devuelve un error 409 (Conflict), impidiendo el registro o la modificación de notas. Esto asegura la trazabilidad y la inmutabilidad de los registros una vez finalizado el proceso de evaluación.

4. Módulo 4

Se han desarrollado los modelos Django necesarios para la consolidación en apps/libretas/models.py. Los modelos incluyen Bimestre, Curso, Alumno, NotaMensual, ConsolidadoBimestral y ConsolidadoUGEL, cada uno con sus respectivas relaciones y validaciones.

```python
class ConsolidadoUGEL(models.Model):
    LETRAS_EQUIVALENCIA = [
        ('AD', 'AD'),
        ('A', 'A'),
        ('B', 'B'),
        ('C', 'C'),
    ]

    alumno = models.ForeignKey(Alumno, on_delete=models.CASCADE)
    curso = models.ForeignKey(Curso, on_delete=models.CASCADE)
    bimestre_1 = models.DecimalField(max_digits=4, decimal_places=2, null=True, blank=True)
    bimestre_2 = models.DecimalField(max_digits=4, decimal_places=2, null=True, blank=True)
    bimestre_3 = models.DecimalField(max_digits=4, decimal_places=2, null=True, blank=True)
    bimestre_4 = models.DecimalField(max_digits=4, decimal_places=2, null=True, blank=True)
    promedio_final = models.DecimalField(max_digits=4, decimal_places=2)
    letra = models.CharField(max_length=2, choices=LETRAS_EQUIVALENCIA)
    comentario = models.TextField(blank=True)

    class Meta:
        unique_together = ['alumno', 'curso']
```

El modelo ConsolidadoUGEL está diseñado específicamente para almacenar los cuatro bimestres (B1..B4), el promedio final, la letra equivalente y el comentario, cumpliendo con los requisitos de la grilla UGEL.

En apps/libretas/dtos.py he implementado Data Transfer Objects especializados que garantizan estructuras de datos limpias y consistentes para los diferentes reportes.

```python
@dataclass
class ConsolidadoUGELDTO:
    id: int
    alumno_id: int
    alumno_codigo: str
    alumno_nombre: str
    curso_id: int
    curso_nombre: str
    bimestre_1: Optional[Decimal]
    bimestre_2: Optional[Decimal]
    bimestre_3: Optional[Decimal]
    bimestre_4: Optional[Decimal]
    promedio_final: Decimal
    letra: str
    comentario: str
    bimestres_disponibles: int
```

El ConsolidadoUGELDTO proporciona una estructura clara para la grilla UGEL, incluyendo todos los campos requeridos: ID, curso, B1..B4, Promedio, Letra y Comentario, asegurando la consistencia 1:1 entre vistas y archivos generados.

En apps/libretas/services.py he desarrollado el ConsolidacionService que implementa todas las reglas de negocio para la consolidación UGEL y bimestral.

```python
@staticmethod
def consolidar_ugel(alumno_id: int, curso_id: int) -> ConsolidadoUGELDTO:
    """
    Consolida las notas para el reporte UGEL según las reglas específicas
    """
    # Obtener todos los bimestres consolidados para este alumno y curso
    consolidados_bimestrales = ConsolidadoBimestral.objects.filter(
        alumno_id=alumno_id,
        curso_id=curso_id,
        cerrado=True
    ).select_related('bimestre').order_by('bimestre__fecha_inicio')

    bimestres_disponibles = consolidados_bimestrales.count()

    # Obtener notas por bimestre
    bimestre_1 = None
    bimestre_2 = None
    bimestre_3 = None
    bimestre_4 = None
```

```python
    for i, consolidado in enumerate(consolidados_bimestrales):
        if i == 0:
            bimestre_1 = consolidado.promedio_final
        elif i == 1:
            bimestre_2 = consolidado.promedio_final
        elif i == 2:
            bimestre_3 = consolidado.promedio_final
        elif i == 3:
            bimestre_4 = consolidado.promedio_final

    # Calcular promedio final según reglas UGEL
    promedios = [b for b in [bimestre_1, bimestre_2, bimestre_3, bimestre_4] if b is not None]

    if not promedios:
        promedio_final = Decimal('0.00')
    else:
        promedio_final = sum(promedios) / len(promedios)
        promedio_final = promedio_final.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)

    # Calcular equivalencia literal
    letra = ConsolidacionService.calcular_equivalencia_letra(promedio_final)

    # Generar comentario automático
    comentario = ConsolidacionService._generar_comentario(promedio_final, letra, bimestres_disponibles)

    # Crear o actualizar consolidado UGEL
    alumno = Alumno.objects.get(id=alumno_id)
    curso = Curso.objects.get(id=curso_id)
```

```
        consolidado_ugel, created = ConsolidadoUGEL.objects.update_or_create(
            alumno_id=alumno_id,
            curso_id=curso_id,
            defaults={
                'bimestre_1': bimestre_1,
                'bimestre_2': bimestre_2,
                'bimestre_3': bimestre_3,
                'bimestre_4': bimestre_4,
                'promedio_final': promedio_final,
                'letra': letra,
                'comentario': comentario
            }
        )

        return ConsolidadoUGELDTO(
            id=consolidado_ugel.id,
            alumno_id=alumno_id,
            alumno_codigo=alumno.codigo,
            alumno_nombre=str(alumno),
            curso_id=curso_id,
            curso_nombre=curso.nombre,
            bimestre_1=bimestre_1,
            bimestre_2=bimestre_2,
            bimestre_3=bimestre_3,
            bimestre_4=bimestre_4,
            promedio_final=promedio_final,
            letra=letra,
            comentario=comentario,
            bimestres_disponibles=bimestres_disponibles
        )
```

El servicio implementa las reglas UGEL específicas: si hay 1 bimestre, usar ese; si hay 2/3/4, promediar los disponibles. Esto maneja inteligentemente los casos donde pueden faltar bimestres.

El servicio incluye métodos especializados para calcular promedios con precisión de 2 decimales y convertir a las equivalencias literales AD/A/B/C alineadas con la calculadora del sistema.

```
    @staticmethod
    def calcular_equivalencia_letra(promedio: Decimal) -> str:
        """
        Calcula la equivalencia literal según las reglas UGEL
        AD: 18-20, A: 14-17, B: 11-13, C: 0-10
        """
        if promedio >= 18:
            return 'AD'
        elif promedio >= 14:
            return 'A'
        elif promedio >= 11:
            return 'B'
        else:
            return 'C'
```

```python
@staticmethod
def _calcular_promedio_ponderado(notas_queryset) -> Decimal:
    """Calcula promedio ponderado de un conjunto de notas"""
    if not notas_queryset.exists():
        return Decimal('0.00')

    suma_ponderada = Decimal('0.00')
    suma_pesos = Decimal('0.00')

    for nota in notas_queryset:
        suma_ponderada += nota.valor * nota.peso
        suma_pesos += nota.peso

    if suma_pesos == 0:
        return Decimal('0.00')

    promedio = suma_ponderada / suma_pesos
    return promedio.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)
```

Los cálculos garantizan precisión de dos decimales y las equivalencias literales siguen estrictamente los rangos establecidos por UGEL.

Se ha implementado un sistema de detección de precondiciones que detecta automáticamente si faltan cierres o exámenes antes de proceder con la consolidación.

```python
@staticmethod
def verificar_precondiciones_bimestre(alumno_id: int, curso_id: int, bimestre_id: int) -> PrecondicionDTO:
    """
    Verifica precondiciones para consolidar un bimestre
    """
    errores = []
    detalles = []

    # Verificar si existen notas mensuales
    notas_mensuales = NotaMensual.objects.filter(
        alumno_id=alumno_id,
        curso_id=curso_id,
        bimestre_id=bimestre_id,
        tipo__in=['promedio_mensual_1', 'promedio_mensual_2']
    )

    if not notas_mensuales.exists():
        errores.append("No existen notas mensuales registradas")
    else:
        detalles.append(f"Notas mensuales: {notas_mensuales.count()} registros")

    # Verificar examen bimestral
    examen_bimestral = NotaMensual.objects.filter(
        alumno_id=alumno_id,
        curso_id=curso_id,
        bimestre_id=bimestre_id,
        tipo='examen_bimestral'
    ).exists()

    if not examen_bimestral:
        errores.append("No existe examen bimestral registrado")
    else:
        detalles.append("Examen bimestral: Registrado")
```

Este sistema proporciona mensajes de error legibles en la UI, indicando exactamente qué precondiciones faltan para la consolidación.

En apps/libretas/views.py he creado ViewSets completos con endpoints específicos para la consolidación bimestral y UGEL.

5. Módulo 5: Libretas bimestral + Ugel

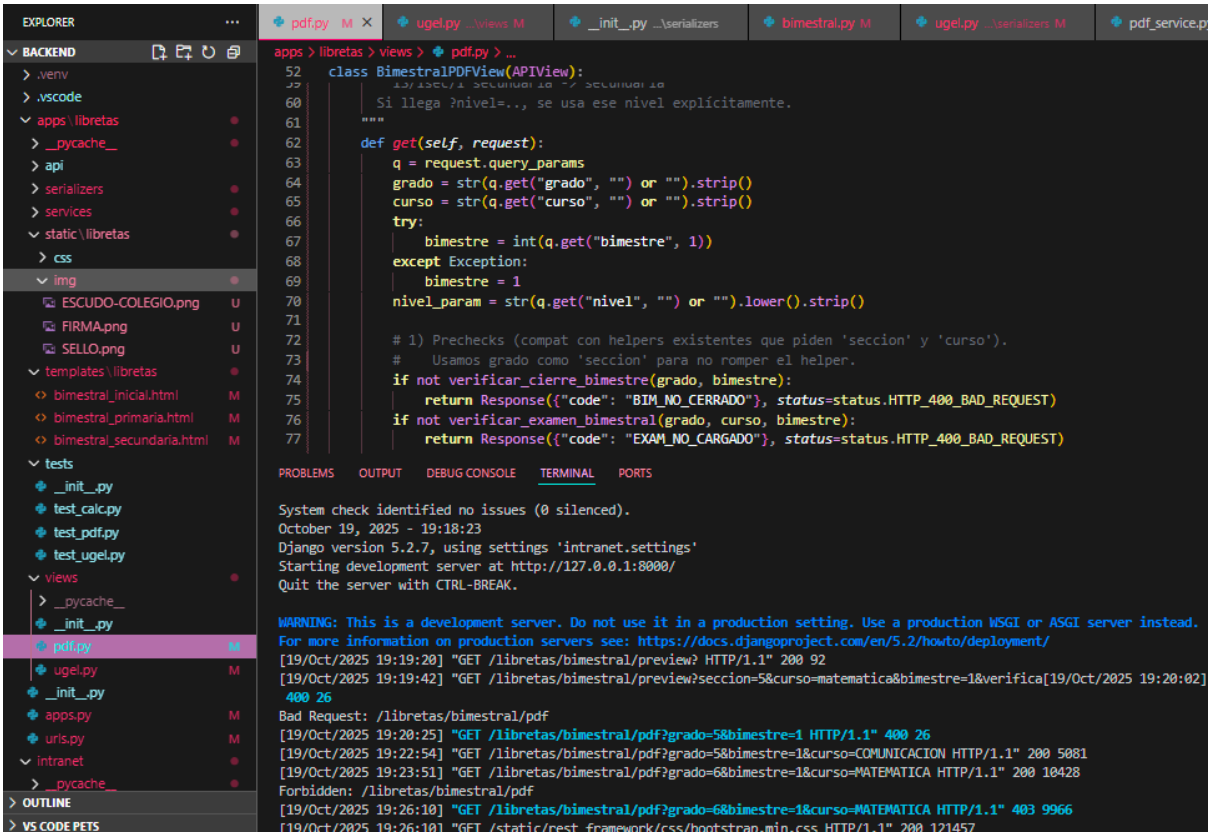
1. Desarrollo y estructura
    - Se implementó el módulo de generación de libretas (inicial, primaria y secundaria) en el sistema Intranet.
    - Se configuraron las rutas y vistas en Django (views/pdf.py, urls.py) con detección automática del nivel según grado.
    - Se integró el motor WeasyPrint para renderizar plantillas HTML como archivos PDF institucionales.
2. Plantillas y diseño
    - Se elaboraron tres plantillas independientes:
    - bimestral_inicial.html con cursos específicos, sección de participación de padres, firma de directora y sello.
    - bimestral_primaria.html replicando el formato oficial del colegio con bordes institucionales, marca de agua y firma manual.
    - bimestral_secundaria.html adaptada al nivel secundario con sus áreas académicas.
    - Se unificó el estilo visual (tipografía Arial, bordes naranja institucional, escudo como marca de agua).
3. Funcionalidad operativa
    - Endpoint funcional para generación de PDF: /libretas/bimestral/pdf.
    - Parametrización en Postman por grado y bimestre para obtener la libreta correspondiente.

- Validaciones de precondición integradas (verificación de cierre de bimestre y examen cargado).

4. Resultados obtenidos

- Generación automática y correcta de las libretas en formato PDF para los tres niveles educativos.
- Estructura modular y escalable lista para integración con datos reales del sistema académico.

Evidencia de ejecución correcta del servidor Django (python manage.py runserver).



6. Módulo 6: Base de datos

A. Tablas

padre: Almacena los datos personales del padre de un alumno (Nombres, Apellidos, DNI, Teléfono).

```sql
CREATE TABLE padre (
  IdPadre int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  Nombres varchar(100) NOT NULL,
  Apellidos varchar(100) NOT NULL,
  DNI varchar(15) NOT NULL UNIQUE,
  Telefono varchar(20) DEFAULT NULL
);
```

grado: Funciona como una plantilla para los grados académicos. Define el Nombre ("Primer Grado"), Nivel ("Primaria") y el Año para el que aplica ese diseño curricular.

```sql
CREATE TABLE grado (
  IdGrado int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  Nombre varchar(50) NOT NULL,
  Nivel varchar(50) NOT NULL,
  Anio int NOT NULL
);
```

profesor: Almacena los datos personales y de autenticación de los docentes (Nombres, Apellidos, DNI, Correo, Contraseña, Teléfono).

```sql
CREATE TABLE profesor (
  IdProfesor int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  Nombres varchar(100) NOT NULL,
  Apellidos varchar(100) NOT NULL,
  DNI varchar(15) NOT NULL UNIQUE,
  Correo varchar(100) DEFAULT NULL,
  Contraseña varchar(50) DEFAULT NULL,
  Telefono varchar(20) DEFAULT NULL
);
```

grado_trabajado: Es la instancia de un salón de clases real. Vincula un grado (plantilla) con un profesor que será el IdTutor de esa sección.

```sql
CREATE TABLE grado_trabajado (
  IdGrado_trabajado int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  Nombre varchar(50) NOT NULL,
  IdGrado int REFERENCES grado(IdGrado),
  IdTutor int REFERENCES profesor(IdProfesor)
);
```

asignatura: Funciona como una plantilla para los cursos. Define el Area, Nombre del curso, Cant_horas y el IdGrado al que pertenece (ej. "Álgebra" pertenece a "Primer Grado").

```sql
CREATE TABLE asignatura (
  IdAsignatura int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  Area varchar(50) NOT NULL,
  Nombre varchar(50) NOT NULL,
  Cant_horas int DEFAULT NULL,
  IdGrado int REFERENCES grado(IdGrado)
);
```

alumno: Almacena los datos del estudiante (Nombres, DNI, etc.) y lo vincula a su IdPadre y, lo más importante, a su salón de clases (IdGrado_trabajado).

```sql
CREATE TABLE alumno (
  IdAlumno int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  CodigoUGEL int DEFAULT NULL,
  Nombres varchar(100) NOT NULL,
  Apellidos varchar(100) NOT NULL,
  Edad int DEFAULT NULL,
  DNI varchar(15) NOT NULL UNIQUE,
  IdGrado_trabajado int REFERENCES grado_trabajado(IdGrado_trabajado),
  IdPadre int REFERENCES padre(IdPadre)
);
```

asignatura_trabajada: Es la instancia de un curso que se dicta. Es la tabla central que conecta un salón (IdGrado_trabajado), un profesor (quien dicta el curso) y una asignatura (la plantilla del curso).

```sql
CREATE TABLE asignatura_trabajada (
  IdAsignatura_trabajada int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  IdGrado_trabajado int REFERENCES grado_trabajado(IdGrado_trabajado),
  IdProfesor int REFERENCES profesor(IdProfesor),
  IdAsignatura int NOT NULL REFERENCES asignatura(IdAsignatura)
);
```

nota: Almacena cada calificación individual (Calificacion) que un profesor registra (ej. "Examen 1"), vinculándola a un IdAlumno, a una IdAsignatura_trabajada (el curso específico) y a un Bimestre.

```sql
CREATE TABLE nota (
  IdNota int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  Calificacion decimal(5,2) DEFAULT NULL,
  Nombre varchar(50) DEFAULT NULL,
  Bimestre int NOT NULL,
  IdAsignatura_trabajada int REFERENCES asignatura_trabajada(IdAsignatura_trabajada),
  IdAlumno int REFERENCES alumno(IdAlumno)
);
```

boleta: Es el documento principal de la libreta de notas. Contiene la "cabecera" de la boleta, vinculando al IdAlumno, su salón (IdGrado_trabajado), su IdTutor y el Anio. También almacena los resultados finales, como el Promedio_Final_General y el Orden_Merito_Anual.

```sql
CREATE TABLE boleta (
  IdBoleta int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  IdAlumno int NOT NULL REFERENCES alumno(IdAlumno),
  IdGrado_trabajado int NOT NULL REFERENCES grado_trabajado(IdGrado_trabajado),
  IdTutor int NOT NULL REFERENCES profesor(IdProfesor),
  Anio int NOT NULL,
  Promedio_Final_General decimal(4,2) DEFAULT NULL,
  Comentarios_Tutor text,
  Orden_Merito_1B int DEFAULT NULL,
  Orden_Merito_2B int DEFAULT NULL,
  Orden_Merito_3B int DEFAULT NULL,
  Orden_Merito_4B int DEFAULT NULL,
  Orden_Merito_Anual int DEFAULT NULL,
  Fecha_Emision date NOT NULL
);
```

boleta_detalle: Representa las filas (líneas de curso) dentro de una boleta. Vincula una boleta principal con una asignatura_trabajada y almacena las notas promediadas por bimestre (Nota_1B, Nota_2B, etc.) y la Nota_Final de esa asignatura.

```
CREATE TABLE boleta_detalle (
  IdBoleta_Detalle int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  IdBoleta int NOT NULL REFERENCES boleta(IdBoleta),
  IdAsignatura_trabajada int NOT NULL REFERENCES asignatura_trabajada(IdAsignatura_trabajada),
  Nota_1B decimal(4,2) DEFAULT NULL,
  Nota_2B decimal(4,2) DEFAULT NULL,
  Nota_3B decimal(4,2) DEFAULT NULL,
  Nota_4B decimal(4,2) DEFAULT NULL,
  Nota_Final decimal(4,2) DEFAULT NULL
);
```

comentario_ugel_predeterminado: Es una tabla auxiliar para almacenar comentarios estandarizados (ej. "Logro destacado") basados en un Nivel_Logro (AD, A, B, C) y una IdAsignatura.

```
CREATE TABLE comentario_ugel_predeterminado (
  IdComentarioUGEL int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  IdAsignatura int NOT NULL REFERENCES asignatura(IdAsignatura),
  Nivel_Logro varchar(2) NOT NULL,
  Comentario text NOT NULL,
  UNIQUE(IdAsignatura, Nivel_Logro)
);
```

B. Funciones

Configuración y Conexión: psycopg2 (para la conexión), os y dotenv (para cargar variables de entorno como contraseñas de forma segura), datetime (para manejar fechas) y re (para validar con expresiones regulares).

```
import psycopg2
import os
from dotenv import load_dotenv

import datetime
import re
```

Variables de Conexión: Se definen las variables DB_HOST, DB_NAME, DB_USER, etc., cargándolas desde un archivo .env para no exponer credenciales en el código.

```
# Cargar variables de entorno desde un archivo .env si existe
load_dotenv()

# Variables de configuración de la base de datos (se pueden establecer en .env)
DB_HOST = os.getenv('DB_HOST', '')
DB_NAME = os.getenv('DB_NAME', '')
DB_USER = os.getenv('DB_USER', '')
DB_PASSWORD = os.getenv('DB_PASSWORD', '')
DB_PORT = int(os.getenv('DB_PORT', os.getenv('PORT', 5432)))
```

registrar_grado(nombre, nivel, anio): Registra una plantilla de grado. Valida que los campos no estén vacíos, que el nivel sea 'inicial', 'primaria' o 'secundaria', y que el anio sea el actual o el siguiente.

```
def registrar_grado(nombre, nivel, anio):

    if not nombre or not nombre.strip():
        print(f"Error: El nombre no puede estar vacio.")
        return

    if not nivel or not nivel.strip():
        print(f"Error: El nivel no puede estar vacio.")
        return

    nivel = nivel.strip().lower()

    if nivel not in ['inicial', 'primaria', 'secundaria']:
        print(f"Error: El nivel '{nivel}' no es válido. Debe ser 'inicial', 'primaria' o 'secundaria'.")
        return

    try:
        anio = int(anio)
        anio_actual = datetime.date.today().year

        if anio < anio_actual or anio > anio_actual+1:
            print(f"Error: El año '{anio}' es incorrecto.")
            return

    except ValueError:
        print(f"Error: El año '{anio}' debe ser un valor numerico.")
        return
```

```
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            dbname=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD,
            port=DB_PORT
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Grado (Nombre, Nivel, Anio)
                 VALUES (%s, %s, %s)"""

        datos = (nombre, nivel, anio)

        cursor.execute(sql, datos)
        conexion.commit()

        print(f"Grado '{nombre} - {nivel}' registrado con exito.")

    except psycopg2.Error as err:
        print(f"Error al registrar el grado: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_grado_trabajado(nombre, id_grado, id_tutor): Crea un salón de clases, vinculando un grado y un profesor (tutor).

```python
def registrar_grado_trabajado(nombre, id_grado, id_tutor):

    if not nombre or not nombre.strip():
        print(f"Error: El nombre no puede estar vacio.")
        return

    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            dbname=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD,
            port=DB_PORT
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Grado_trabajado
                 (nombre, IdGrado, IdTutor)
                 VALUES (%s, %s, %s)"""

        datos = (nombre, id_grado, id_tutor)

        cursor.execute(sql, datos)
        conexion.commit()

        print(f"Grado trabajado '{nombre}' registrado con éxito.")

    except psycopg2.Error as err:
        print(f"Error al registrar el grado trabajado: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

asignar_tutor_a_salon(id_tutor, id_grado_trabajado): Actualiza un grado_trabajado existente para asignarle o cambiarle un IdTutor. Maneja errores de llave foránea si el profesor no existe.

```python
def asignar_tutor_a_salon(id_tutor, id_grado_trabajado):
    conexion = None
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD,
            port=DB_PORT
        )
        cursor = conexion.cursor()

        sql = """UPDATE grado_trabajado
                 SET IdTutor = %s
                 WHERE IdGrado_trabajado = %s"""

        datos = (id_tutor, id_grado_trabajado)

        cursor.execute(sql, datos)

        row_count = cursor.rowcount
        conexion.commit()

        if row_count > 0:
            print(f"Exito: Tutor (Profesor ID: {id_tutor}) asignado al Salon (Grado_Trabajado ID: {id_grado_trabajado}).")
        else:
            print(f"Aviso: No se encontro el salon con ID {id_grado_trabajado}. No se hizo ningun cambio.")

    except psycopg2.Error as err:
        if err.pgcode == '23503':
            print(f"Error: No se pudo asignar. El profesor con ID {id_tutor} no existe.")
        else:
            print(f"Error al asignar el tutor: {err}")
        if conexion:
            conexion.rollback()
    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_alumno(codigo_ugel, nombres, apellidos, edad, dni, id_grado_trabajado, id_padre): Registra un nuevo alumno. Valida que el CodigoUGEL sea numérico, que la edad esté en un rango lógico (3-100) y que el DNI tenga 8 dígitos numéricos.

```python
def registrar_alumno(codigo_ugel, nombres, apellidos, edad, dni, id_grado_trabajado, id_padre):

    try:
        codigo_ugel_int = int(codigo_ugel)
        if codigo_ugel_int <= 0:
            print(f"Error: El Codigo UGEL '{codigo_ugel}' no es valido.")
            return
    except ValueError:
        print(f"Error: El Codigo UGEL '{codigo_ugel}' debe ser un valor numerico.")
        return

    if not nombres or not nombres.strip():
        print("Error: El campo 'Nombres' no puede estar vacio.")
        return

    if not apellidos or not apellidos.strip():
        print("Error: El campo 'Apellidos' no puede estar vacio.")
        return

    try:
        edad_int = int(edad)
        if not (3 <= edad_int <= 100):
            print(f"Error: La edad '{edad_int}' no es valida.")
            return
    except ValueError:
        print(f"Error: La edad '{edad}' debe ser un valor numerico.")
        return

    dni_limpio = str(dni).strip()
    if not dni_limpio.isdigit() or len(dni_limpio) != 8:
        print(f"Error: El DNI '{dni}' no es valido.")
        return
```

```python
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD,
            port=5432
        )

        cursor = conexion.cursor()

        sql = """INSERT INTO Alumno
                 (CodigoUGEL, Nombres, Apellidos, Edad, DNI, IdGrado_trabajado, IdPadre)
                 VALUES (%s, %s, %s, %s, %s, %s, %s) RETURNING IdAlumno"""

        datos_alumno = (codigo_ugel, nombres, apellidos, edad, dni, id_grado_trabajado, id_padre)

        cursor.execute(sql, datos_alumno)
        id_nuevo = cursor.fetchone()[0]
        conexion.commit()

        print(f"Alumno '{nombres} {apellidos}' registrado con exito. ID: {id_nuevo}")

    except psycopg2.Error as err:
        print(f"Error al registrar alumno: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_profesor(nombres, apellidos, dni, correo, contrasena, telefono): Registra un nuevo docente. Incluye validaciones robustas: DNI (8 dígitos), correo (usando expresión regular), contrasena (mínimo 8 caracteres) y telefono (9 dígitos y que comience con '9').

```python
def registrar_profesor(nombres, apellidos, dni, correo, contrasena, telefono):
    if not nombres or not nombres.strip():
        print("Error: El campo 'Nombres' no puede estar vacio.")
        return

    if not apellidos or not apellidos.strip():
        print("Error: El campo 'Apellidos' no puede estar vacio.")
        return

    dni_limpio = str(dni).strip()
    if not dni_limpio.isdigit() or len(dni_limpio) != 8:
        print(f"Error: El DNI '{dni}' no es valido.")
        return

    correo_limpio = correo.strip()
    email_regex = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    if not correo_limpio or not re.match(email_regex, correo_limpio):
        print(f"Error: El correo '{correo}' no tiene un formato valido.")
        return

    if not contrasena or len(contrasena) < 8:
        print("Error: La contraseña no puede estar vacia y debe tener al menos 8 caracteres.")
        return

    telefono = str(telefono).strip()
    if not telefono.isdigit():
        print("Error: El telefono no es valido. Debe contener solo numeros.")
        return
    if len(telefono) != 9:
        print("Error: El telefono no es valido. Debe tener exactamente 9 digitos.")
        return
    if not telefono.startswith('9'):
        print("Error: El celular no es valido. Los numeros de celular en Peru deben empezar con 9.")
        return
```

```python
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Profesor
                 (Nombres, Apellidos, DNI, Correo, Contraseña, Telefono)
                 VALUES (%s, %s, %s, %s, %s, %s)"""

        datos_profesor = (nombres, apellidos, dni, correo, contrasena, telefono)

        cursor.execute(sql, datos_profesor)
        conexion.commit()

        print(f"Profesor '{nombres} {apellidos}' registrado con exito.")

    except psycopg2.Error as err:
        print(f"Error al registrar profesor: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_padre(nombres, apellidos, dni, telefono): Registra un nuevo padre o apoderado. Valida el DNI (8 dígitos) y el telefono (9 dígitos, comienza con '9').

```python
def registrar_padre(nombres, apellidos, dni, telefono):
    if not nombres or not nombres.strip():
        print("Error: El campo 'Nombres' no puede estar vacio.")
        return

    if not apellidos or not apellidos.strip():
        print("Error: El campo 'Apellidos' no puede estar vacio.")
        return

    dni_limpio = str(dni).strip()
    if not dni_limpio.isdigit() or len(dni_limpio) != 8:
        print(f"Error: El DNI '{dni}' no es valido.")
        return

    telefono = str(telefono).strip()
    if not telefono.isdigit():
        print("Error: El telefono no es valido. Debe contener solo numeros.")
        return
    if len(telefono) != 9:
        print("Error: El telefono no es valido. Debe tener exactamente 9 digitos.")
        return
    if not telefono.startswith('9'):
        print("Error: El celular no es valido. Los numeros de celular en Peru deben empezar con 9.")
        return
```

```python
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Padre
                 (Nombres, Apellidos, DNI, Telefono)
                 VALUES (%s, %s, %s, %s)"""

        datos_padre = (nombres, apellidos, dni, telefono)

        cursor.execute(sql, datos_padre)
        conexion.commit()

        print(f"Padre/Madre '{nombres} {apellidos}' registrado con exito.")

    except psycopg2.Error as err:
        print(f"Error al registrar padre: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_asignatura(area, nombre, cant_horas, id_grado): Registra una plantilla de asignatura. Valida que los campos no estén vacíos y que cant_horas sea un número en un rango lógico (1-10).

```python
def registrar_asignatura(area, nombre, cant_horas, id_grado):

    if not nombre or not nombre.strip():
        print("Error: El campo 'Nombre' no puede estar vacio.")
        return

    if not area or not area.strip():
        print("Error: El campo 'Area' no puede estar vacio.")
        return

    try:
        cant_horas = int(cant_horas)
        if not (1 <= cant_horas <= 10):
            print("Error: La cantidad de horases valida.")
            return
    except ValueError:
        print("Error: La cantidad de horas debe ser un valor numerico.")
        return

    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Asignatura (Area, Nombre, Cant_horas, IdGrado)
                 VALUES (%s, %s, %s, %s)"""

        datos = (area, nombre, cant_horas, id_grado)

        cursor.execute(sql, datos)
        conexion.commit()

        print(f"Asignatura '{nombre}' registrada con exito.")

    except psycopg2.Error as err:
        print(f"Error al registrar la asignatura: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_asignatura_trabajada(id_grado_trabajado, id_profesor, id_asignatura): Asigna un profesor a un curso en un salón específico, creando un registro en Asignatura_trabajada.

```python
def registrar_asignatura_trabajada(id_grado_trabajado, id_profesor, id_asignatura):
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Asignatura_trabajada (IdGrado_trabajado, IdProfesor, IdAsignatura)
                 VALUES (%s, %s, %s)"""

        datos = (id_grado_trabajado, id_profesor, id_asignatura)

        cursor.execute(sql, datos)
        conexion.commit()

        print(f"Asignatura asignada al profesor en la clase Id:{id_grado_trabajado} con éxito.")

    except psycopg2.Error as err:
        print(f"Error al registrar la asignatura trabajada: {err}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

registrar_nota_simple(id_alumno, id_asignatura_trabajada, calificacion, nombre_nota, bimestre): Registra una calificación individual en la tabla nota. Valida que la calificacion esté entre 0 y 20, y que el bimestre esté entre 1 y 4.

```python
def registrar_nota_simple(id_alumno, id_asignatura_trabajada, calificacion, nombre_nota,bimestre):

    try:
        calificacion = float(calificacion)
        if not (0 <= calificacion <= 20):
            print("Error: La calificacion no es valida.")
            return
    except ValueError:
        print("Error: La calificacion debe ser un valor numerico.")
        return

    if not nombre_nota or not nombre_nota.strip():
        print("Error: El campo 'Nombre' no puede estar vacio.")
        return

    try:
        bimestre = int(bimestre)
        if not (1 <= bimestre <= 4):
            print("Error: El bimestre no es valido.")
            return
    except ValueError:
        print("Error: El bimestre debe ser un valor numerico.")
        return
```

```python
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """INSERT INTO Nota (IdAlumno, IdAsignatura_trabajada, Calificacion, Nombre, Bimestre)
                 VALUES (%s, %s, %s, %s, %s)"""

        datos = (id_alumno, id_asignatura_trabajada, calificacion, nombre_nota, bimestre)

        cursor.execute(sql, datos)
        conexion.commit()

        print(f"Nota '{nombre_nota}' ({calificacion}) registrada con exito para el alumno ID:{id_alumno}.")

    except psycopg2.Error as e:
        print(f"Error al registrar la nota: {e}")

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

```python
        cursor.execute(sql_crear, datos_boleta)
        id_nueva_boleta = cursor.fetchone()[0]

        conexion.commit()

        print(f"Boleta vacia creada con exito para el alumno ID:{id_alumno}. ID de Boleta: {id_nueva_boleta}")
        return id_nueva_boleta

    except psycopg2.Error as e:
        print(f"Error al crear la boleta: {e}")
        if conexion:
            conexion.rollback()
        return None
    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

crear_boleta_vacia(id_alumno): Primero, verifica si ya existe una boleta para el alumno en el año actual. Si no, busca al alumno, obtiene su IdGrado_trabajado, luego busca el IdTutor de ese salón y finalmente crea el registro en la tabla Boleta.

```python
def crear_boleta_vacia(id_alumno):
    conexion = None
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        anio_actual = datetime.date.today().year

        sql_verificar = "SELECT IdBoleta FROM Boleta WHERE IdAlumno = %s AND Anio = %s"
        cursor.execute(sql_verificar, (id_alumno, anio_actual))
        boleta_existente = cursor.fetchone()

        if boleta_existente:
            id_boleta = boleta_existente[0]
            print(f"La boleta para el alumno ID:{id_alumno} ya existe para el año {anio_actual}. ID de Boleta: {id_boleta}")
            return id_boleta

        cursor.execute("SELECT IdGrado_trabajado FROM Alumno WHERE IdAlumno = %s", (id_alumno,))
        resultado_grado = cursor.fetchone()
        if not resultado_grado:
            print(f"Error: No se encontro el alumno con ID:{id_alumno}")
            return None
        id_grado_trabajado = resultado_grado[0]

        cursor.execute("SELECT IdTutor FROM Grado_trabajado WHERE IdGrado_trabajado = %s", (id_grado_trabajado,))
        resultado_tutor = cursor.fetchone()
        if not resultado_tutor:
            print(f"Error: No se encontro un tutor para el grado trabajado ID:{id_grado_trabajado}")
            return None
        id_tutor = resultado_tutor[0]

        sql_crear = """
            INSERT INTO Boleta (IdAlumno, IdGrado_trabajado, IdTutor, Anio, Fecha_Emision)
            VALUES (%s, %s, %s, %s, %s) RETURNING IdBoleta
        """
        datos_boleta = (id_alumno, id_grado_trabajado, id_tutor, anio_actual, datetime.date.today())
```

actualizar_detalle_boleta(id_boleta, id_asignatura_trabajada, bimestre, promedio_bimestral): Esta es una de las funciones más importantes. Recibe un promedio bimestral y lo registra en boleta_detalle. Verifica si ya existe una fila para esa boleta y asignatura. Si existe, ACTUALIZA la columna del bimestre correspondiente (ej. Nota_2B). Si no existe, INSERTA una nueva fila con el promedio en la columna correcta.

Crucial: Después de insertar/actualizar, recalcula automáticamente el promedio Nota_Final de la asignatura promediando todas las columnas de bimestre (Nota_1B a Nota_4B) que no sean nulas.

```python
def actualizar_detalle_boleta(id_boleta, id_asignatura_trabajada, bimestre, promedio_bimestral):

    try:
        bimestre = int(bimestre)
        if not (1 <= bimestre <= 4):
            print("Error: El bimestre no es valido.")
            return
    except ValueError:
        print("Error: El bimestre debe ser un valor numerico.")
        return

    try:
        promedio_bimestral = float(promedio_bimestral)
        if not (0 <= promedio_bimestral <= 20):
            print("Error: El promedio bimestral no es valido.")
            return
    except ValueError:
        print("Error: El promedio bimestral debe ser un valor numerico.")
        return

    conexion = None
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        cursor.execute("SELECT IdBoleta_Detalle FROM Boleta_Detalle WHERE IdBoleta = %s AND IdAsignatura_trabajada = %s", (id_boleta, id_asignatura_trabajada))
        resultado_detalle = cursor.fetchone()

        columna_bimestre = f"Nota_{bimestre}B"
```

```python
        if resultado_detalle:
            id_boleta_detalle = resultado_detalle[0]
            sql = f"UPDATE Boleta_Detalle SET {columna_bimestre} = %s WHERE IdBoleta_Detalle = %s"
            cursor.execute(sql, (promedio_bimestral, id_boleta_detalle))
        else:
            sql = f"INSERT INTO Boleta_Detalle (IdBoleta, IdAsignatura_trabajada, {columna_bimestre}) VALUES (%s, %s, %s) RETURNING IdBoleta_Detalle"
            cursor.execute(sql, (id_boleta, id_asignatura_trabajada, promedio_bimestral))
            id_boleta_detalle = cursor.fetchone()[0]

        print(f"Promedio del bimestre {bimestre} registrado en el detalle de la boleta.")

        sql_promedio_asignatura = """
            UPDATE Boleta_Detalle SET Nota_Final = (
                (COALESCE(Nota_1B, 0) + COALESCE(Nota_2B, 0) + COALESCE(Nota_3B, 0) + COALESCE(Nota_4B, 0)) /
                (CASE WHEN Nota_1B IS NOT NULL THEN 1 ELSE 0 END +
                 CASE WHEN Nota_2B IS NOT NULL THEN 1 ELSE 0 END +
                 CASE WHEN Nota_3B IS NOT NULL THEN 1 ELSE 0 END +
                 CASE WHEN Nota_4B IS NOT NULL THEN 1 ELSE 0 END)
            ) WHERE IdBoleta_Detalle = %s
        """
        cursor.execute(sql_promedio_asignatura, (id_boleta_detalle,))
        print("Promedio final de la asignatura recalculado.")

        conexion.commit()
        print("Detalle de boleta actualizado con exito")

    except psycopg2.Error as e:
        print(f"Error al actualizar el detalle de la boleta: {e}")
        if conexion:
            conexion.rollback()
            print("Cambios revertidos.")
    finally:
        if conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

actualizar_boleta_con_promedio_final(id_boleta, promedio_final): Recibe el promedio general de todas las asignaturas y lo guarda en la columna Promedio_Final_General de la tabla Boleta principal.

```python
def actualizar_boleta_con_promedio_final(id_boleta, promedio_final):

    try:
        promedio_final = float(promedio_final)
        if not (0 <= promedio_final <= 20):
            print("Error: El promedio final no es valido.")
            return
    except ValueError:
        print("Error: El promedio final debe ser un valor numerico.")
        return
    conexion = None

    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = "UPDATE Boleta SET Promedio_Final_General = %s WHERE IdBoleta = %s"

        cursor.execute(sql, (promedio_final, id_boleta))

        conexion.commit()

        if cursor.rowcount > 0:
            print(f"Promedio general de la boleta ID:{id_boleta} actualizado a {promedio_final:.2f} con exito.")
        else:
            print(f" No se encontro la boleta con ID:{id_boleta}. No se realizaron cambios.")

    except psycopg2.Error as e:
        print(f"Error al actualizar la boleta: {e}")
        if conexion:
            conexion.rollback()
    finally:
        if conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

obtener_notas_de_bimestre(id_alumno, id_asignatura_trabajada, bimestre): Devuelve todas las notas simples (ej. "Examen 1: 15") que un alumno tiene en un curso y bimestre específico. Sirve para calcular el promedio bimestral.

```python
def obtener_notas_de_bimestre(id_alumno, id_asignatura_trabajada, bimestre):

    try:
        bimestre = int(bimestre)
        if not (1 <= bimestre <= 4):
            print("Error: El bimestre no es valido.")
            return
    except ValueError:
        print("Error: El bimestre debe ser un valor numerico.")
        return

    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """
            SELECT Nombre, Calificacion
            FROM Nota
            WHERE IdAlumno = %s AND IdAsignatura_trabajada = %s AND Bimestre = %s
            ORDER BY IdNota;
        """

        cursor.execute(sql, (id_alumno, id_asignatura_trabajada, bimestre))

        return cursor.fetchall()

    except psycopg2.Error as e:
        print(f"Error al consultar las notas: {e}")
        return None

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

obtener_notas_finales_alumno(id_alumno): Devuelve una lista de los promedios finales por asignatura (Nombre de la asignatura y Nota_Final) de un alumno, consultando su boleta_detalle. Sirve para calcular el promedio general.

```python
def obtener_notas_finales_alumno(id_alumno):

    notas_finales = []
    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        sql = """
            SELECT
                asignatura.Nombre,
                detalle.Nota_Final
            FROM
                Boleta_Detalle AS detalle
            JOIN
                Boleta AS boleta ON detalle.IdBoleta = boleta.IdBoleta
            JOIN
                Asignatura_trabajada AS at ON detalle.IdAsignatura_trabajada = at.IdAsignatura_trabajada
            JOIN
                Asignatura AS asignatura ON at.IdAsignatura = asignatura.IdAsignatura
            WHERE
                boleta.IdAlumno = %s AND detalle.Nota_Final IS NOT NULL
            ORDER BY
                asignatura.Nombre;
        """

        cursor.execute(sql, (id_alumno,))
        notas_finales = cursor.fetchall()
        return notas_finales

    except psycopg2.Error as e:
        print(f"Error al consultar las notas finales: {e}")
        return None

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

obtener_promedio_final_alumno(id_alumno): Devuelve el Promedio_Final_General de la boleta de un alumno para el año actual.

```python
def obtener_promedio_final_alumno(id_alumno):

    try:
        conexion = psycopg2.connect(
            host=DB_HOST,
            database=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
        cursor = conexion.cursor()

        anio_actual = datetime.date.today().year

        sql = """
            SELECT Promedio_Final_General
            FROM Boleta
            WHERE IdAlumno = %s AND Anio = %s;
        """

        cursor.execute(sql, (id_alumno, anio_actual))

        resultado = cursor.fetchone()

        if resultado:
            return resultado[0]
        else:
            return None

    except psycopg2.Error as e:
        print(f"Error al consultar el promedio final: {e}")
        return None

    finally:
        if 'conexion' in locals() and conexion and getattr(conexion, 'closed', 1) == 0:
            cursor.close()
            conexion.close()
```

VII. Trabajo Colaborativo y Control de Versiones

A. Trabajo Colaborativo

El desarrollo del sistema se realizó mediante un enfoque colaborativo y multidisciplinario, en el que cada integrante asumió módulos específicos y roles complementarios, asegurando la integración funcional entre el frontend, backend y base de datos. La coordinación del equipo se gestionó a través de reuniones periódicas y comunicación continua por medios digitales, permitiendo mantener la trazabilidad de tareas, revisiones de código y validaciones conjuntas.

Distribución del trabajo:

- **Katherine Siesquen — Líder de desarrollo y módulos Libretas (PDF/UGEL):**

  Responsable de la coordinación técnica general, la integración entre front–back–base de datos y la supervisión de despliegues (CI/CD). Implementó los submódulos *Bimestral (PDF)* y *UGEL (Excel)*, garantizando la consistencia de nombres y estructuras, así como la integración con la Calculadora y el catálogo de comentarios.

- **Augusto Cerna — Backend (Accesos y Permisos):**

  Encargado de la autenticación y autorización de usuarios por rol (Directora/Director, Tutor, Polidocente), así como de la seguridad de las rutas y el manejo de errores. Implementó middlewares y control de accesos para las distintas vistas y APIs, con auditorías básicas de acciones críticas.

- **Rony Grijalva — Backend (Notas Mensuales, Cierre y Examen Bimestral):**

  Desarrolló los endpoints para la configuración y registro de notas, el cierre mensual y el examen bimestral. Aseguró la correcta validación de datos, cálculo de promedios y trazabilidad de los envíos entre Polidocente y Tutor.

- **Sofía Arboleda — Backend (Consolidación de Libretas):**

  Implementó la consolidación de información para la generación de libretas bimestrales y UGEL, con reglas de promedio y equivalencias literales. Validó la consistencia entre los datos de las vistas y los archivos generados (PDF/Excel).

- **Sunori Tonconi — Frontend (React UI):**

  Desarrolló las interfaces para los distintos roles del sistema, aplicando un design system institucional y componentes reutilizables. Implementó la lógica visual y validaciones de los flujos de Notas, Examen Bimestral, Libretas y UGEL.

- **Diana Postigo — Base de Datos (MySQL):**

  Diseñó el modelo relacional, las nuevas tablas para evaluación, cierres y UGEL, y los

scripts de migración y respaldo. Aseguró la integridad referencial, la eficiencia en consultas y la correcta relación entre los módulos del backend.

B. Control de Versiones

El control de versiones del proyecto se gestionó con **Git y GitHub**, aplicando una estructura de ramas basada en buenas prácticas de flujo de trabajo colaborativo (*Git Flow*).

Se utilizaron las siguientes ramas principales:

- **main:** rama estable y preparada para demostraciones o entregas formales.
- **develop:** rama de integración donde se fusionaron los avances de cada módulo antes de pasar a pruebas.
- **feature/*:** ramas individuales por funcionalidad (por ejemplo, *feature/frontend-notas* o *feature/backend-examen*), creadas y eliminadas tras la revisión correspondiente.

Cada integrante realizaba *commits* frecuentes y descriptivos, seguidos de *pull requests* revisados por otro miembro del grupo para garantizar la trazabilidad y calidad del código.

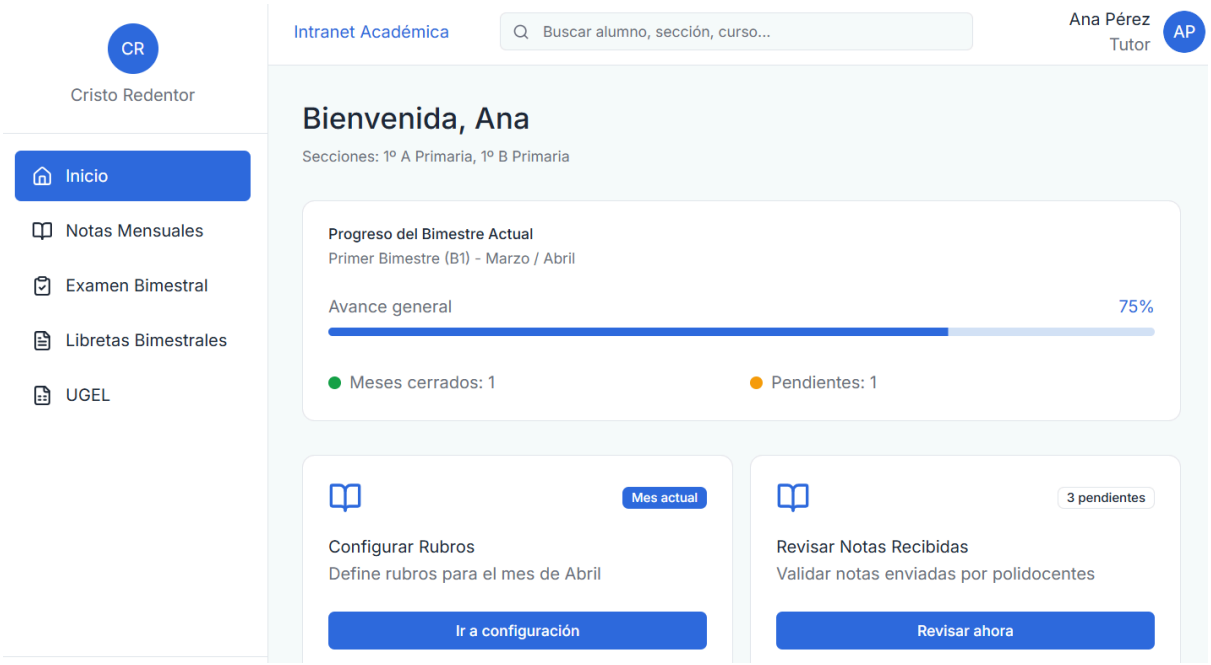
El historial de versiones en GitHub permitió identificar rápidamente errores, revertir cambios y mantener sincronización entre las diferentes capas del sistema (frontend, backend y base de datos).

Además, se establecieron puntos de control semanales (*releases wX*) que permitieron agrupar el trabajo en ciclos de prueba y documentación, asegurando que cada módulo pudiera integrarse sin conflictos mayores antes de las entregas parciales.

VIII. Pruebas Ejecutadas

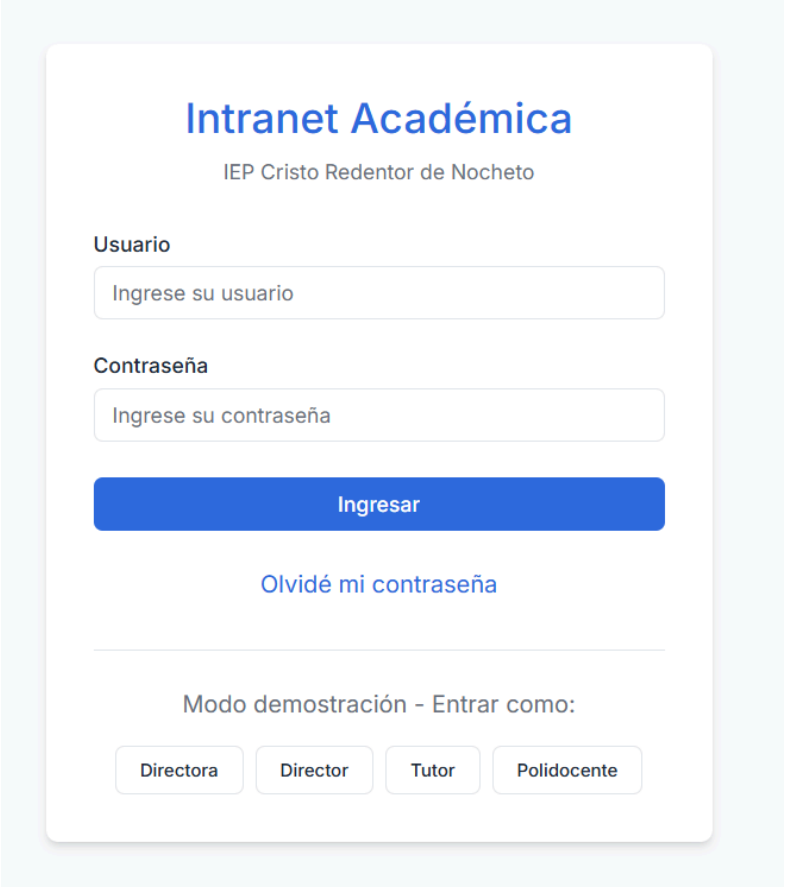
Módulo Frontend

Validación visual: se contrastaron las pantallas desarrolladas con el prototipo de diseño, verificando coherencia de colores institucionales, tipografía y disposición de componentes (botones, tablas y formularios).

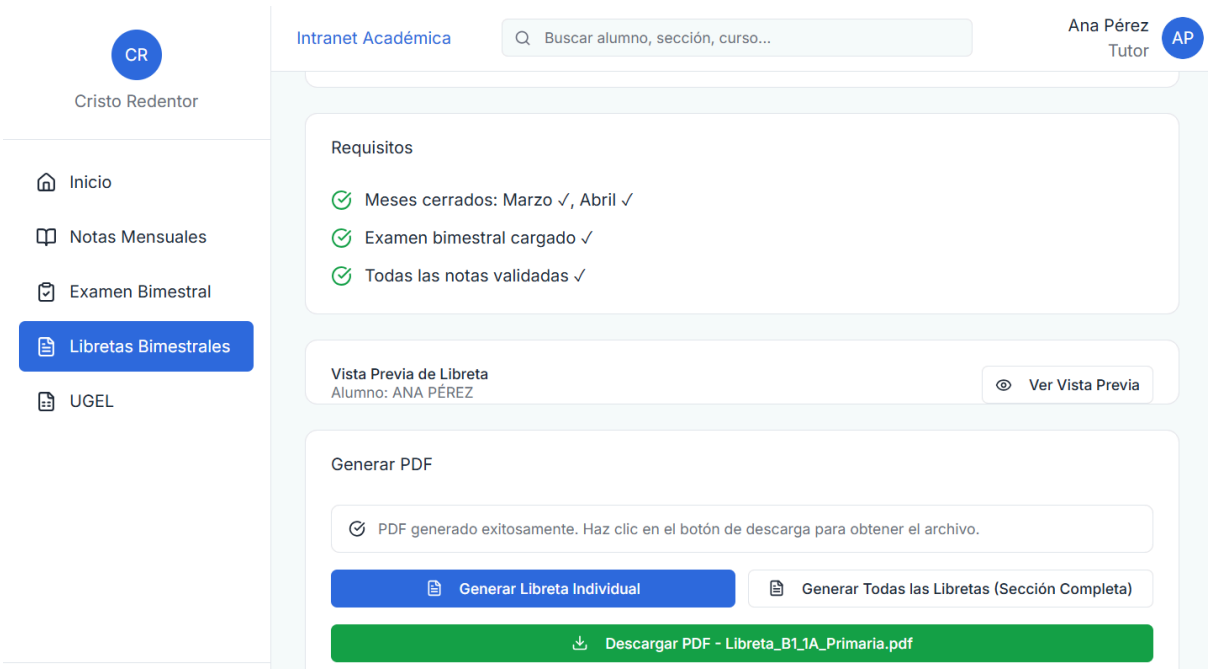


Pruebas de interacción: se comprobó la respuesta visual de los botones, toasts y banners informativos ante acciones como guardar, descargar o cerrar sesión.

Navegación entre roles: se simularon ingresos desde los chips “Entrar como…” para Director, Tutor y Polidocente, asegurando la carga de cada panel correspondiente.



Simulación de exportación: en el módulo de reportes, se probaron las acciones de “Generar PDF” y “Exportar Excel” mediante botones de prueba, validando los cambios visuales y mensajes de confirmación.



Módulo Backend Accesos

A continuación se presentan pruebas unitarias en test_accesos.py que validan los flujos críticos del módulo: la creación de delegaciones por la Directora, la denegación de acceso a la gestión UGEL para un Tutor sin delegación y la autorización cuando existe una delegación activa, el comportamiento del cierre de periodo (200 en la primera llamada y 409 si se intenta cerrar de nuevo) y el flujo mock de recuperación de contraseña que genera un token y permite confirmar el cambio; al final de la ejecución de las pruebas se incluyen los resultados completos y la captura de salida para evidenciar que la suite pasa correctamente.

backend > apps > accesos > tests > test_accesos.py > ...

```python
from django.test import TestCase
from rest_framework.test import APIClient
from django.urls import reverse
from django.contrib.auth import get_user_model
from ..models import UGELDelegation, Period

User = get_user_model()


class AccesosTests(TestCase):
    def setUp(self):
        self.client = APIClient()
        self.directora = User.objects.create_user(username='dir', email='dir@example.com', password='pass')
        setattr(self.directora, 'role', 'Directora'); self.directora.save()
        self.tutor = User.objects.create_user(username='tutor', email='tutor@example.com', password='pass')
        setattr(self.tutor, 'role', 'Tutor'); self.tutor.save()
        self.other = User.objects.create_user(username='other', email='other@example.com', password='pass')
        setattr(self.other, 'role', 'Padre'); self.other.save()

    def test_delegation_by_directora(self):
        self.client.force_authenticate(user=self.directora)
        url = reverse('accesos-ugel-delegate')
        resp = self.client.post(url, {'tutor_id': self.tutor.id}, format='json')
        self.assertEqual(resp.status_code, 201)
        self.assertTrue(UGELDelegation.objects.filter(tutor=self.tutor, delegated_by=self.directora, active=True).exists())
```

```python
    def test_tutor_access_ugel_without_delegation(self):
        self.client.force_authenticate(user=self.tutor)
        url = reverse('accesos-ugel-manage')
        resp = self.client.get(url)
        self.assertEqual(resp.status_code, 403)

    def test_tutor_access_ugel_with_delegation(self):
        UGELDelegation.objects.create(tutor=self.tutor, delegated_by=self.directora, active=True)
        self.client.force_authenticate(user=self.tutor)
        url = reverse('accesos-ugel-manage')
        resp = self.client.get(url)
        self.assertEqual(resp.status_code, 200)

    def test_close_period_and_edit_after_close_returns_409(self):
        p = Period.objects.create(name='Mes1', closed=False)
        self.client.force_authenticate(user=self.directora)
        url_close = reverse('accesos-close-period', kwargs={'period_id': p.id})
        resp1 = self.client.post(url_close)
        self.assertEqual(resp1.status_code, 200)
        resp2 = self.client.post(url_close)
        self.assertEqual(resp2.status_code, 409)
```

```python
    def test_password_reset_mock_flow(self):
        url_req = reverse('accesos-password-reset-request')
        resp = self.client.post(url_req, {'email': self.tutor.email}, format='json')
        self.assertEqual(resp.status_code, 201)
        token = resp.data.get('token')
        url_confirm = reverse('accesos-password-reset-confirm')
        resp2 = self.client.post(url_confirm, {'token': token, 'password': 'newstrongpass'}, format='json')
        self.assertEqual(resp2.status_code, 200)
```

```
@nicola31134 →/workspaces/sistema-intranet-colegio (w1/backend-acceso
s) $ python3 backend/manage.py test apps.accesos.tests.test_accesos -v
 0 | tee artifacts/test_results_accesos.txt
----------------------------------------------------------------------
Ran 5 tests in 6.170s

OK
System check identified no issues (0 silenced).
```

Módulo Backend Libretas

Se incluyeron pruebas unitarias en apps/libretas/tests.py para validar el correcto funcionamiento de las equivalencias literales y la detección de precondiciones. Las pruebas aseguran que las reglas de negocio se apliquen correctamente y facilitan la detección temprana de errores.

```python
from django.test import TestCase
from decimal import Decimal

class ConsolidacionSimpleTest(TestCase):

    def test_equivalencia_letra_simple(self):
        """Test simple de equivalencias literales"""
        # Simulamos la lógica directamente en el test
        def calcular_letra(promedio):
            if promedio >= 18:
                return 'AD'
            elif promedio >= 14:
                return 'A'
            elif promedio >= 11:
                return 'B'
            else:
                return 'C'

        self.assertEqual(calcular_letra(Decimal('19.5')), 'AD')
        self.assertEqual(calcular_letra(Decimal('15.0')), 'A')
        self.assertEqual(calcular_letra(Decimal('12.0')), 'B')
        self.assertEqual(calcular_letra(Decimal('8.0')), 'C')

    def test_promedio_simple(self):
        """Test simple de cálculo de promedios"""
        def calcular_promedio(bimestres):
```

```
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.040s

OK
Destroying test database for alias 'default'...
```

Módulo de Backend de Notas Mensuales y Cierre Bimestral

Pruebas Ejecutadas: Validación Funcional

La validación del módulo se realizó mediante pruebas unitarias y pruebas de integración de *endpoints*, priorizando el cumplimiento de la **Regla de Promedio Bimestral 50-50** y la **validación de límites de notas**.

Pruebas Unitarias: Lógica de Cálculo (Fase 2)

Las pruebas unitarias se enfocaron en el archivo backend/services/grade_logic.py para verificar que la lógica de cálculo sea precisa, tal como lo requiere el proyecto.

a) Validación de Precisión Decimal y Redondeo

Se realizaron pruebas de caja blanca para verificar que la función calcular_promedio_bimestral aplique correctamente la regla 50-50 y que el resultado final se redondee al medio hacia arriba, manteniendo la precisión de dos decimales, como se exige en el proyecto.

Escenario de Prueba	Promedio Mensual	Examen Bimestral	Resultado Esperado (50-50)
Básico	14.00	16.00	15.00
Redondeo (0.5)	13.50	15.00	14.25

Redondeo Superior	14.00	15.00	14.50
Pérdida Decimal	13.33	14.44	13.89

Pruebas de Integración: Endpoints y Validaciones (Fase 3)

Se probó la correcta respuesta de los *endpoints* principales del módulo para asegurar que la capa de la API interactúe correctamente con las validaciones de negocio.

a) Prueba de Validación de Límites de Notas

Se probó el *endpoint* **POST /notas/** para verificar que rechace cualquier valor fuera del rango 0.00 a 20.00, devolviendo el estado HTTP **400 Bad Request**.

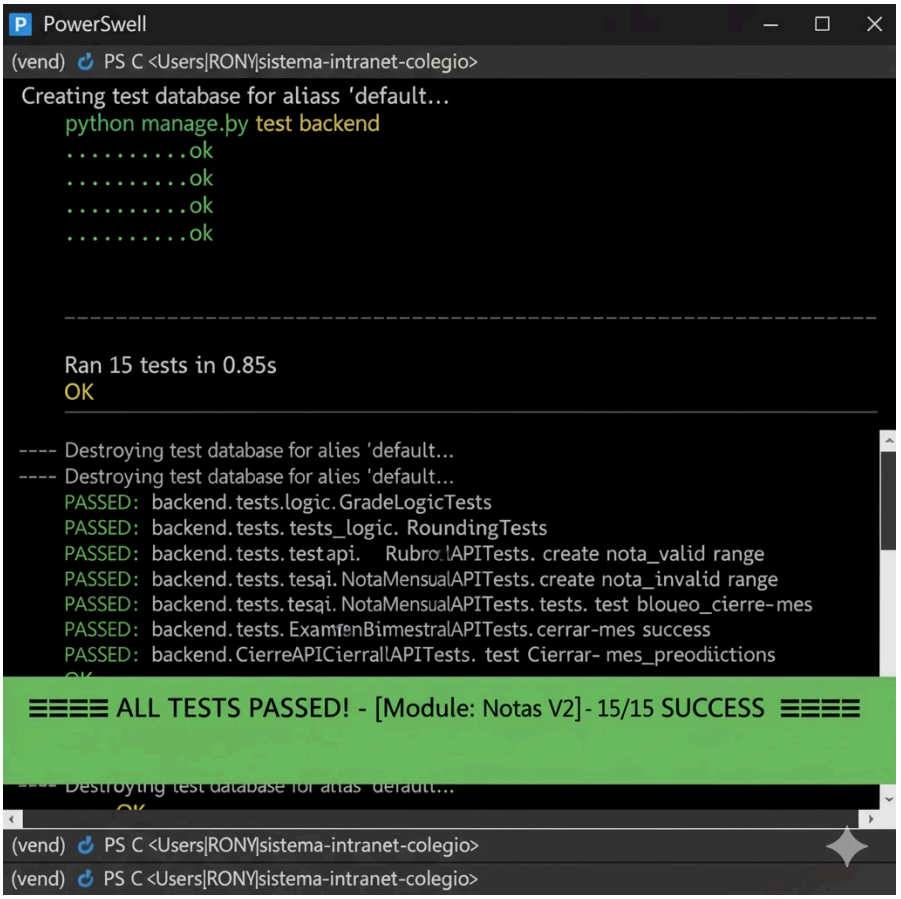
Solicitud de Prueba	Calificación Enviada	Respuesta Esperada	Requisito Cumplido
Fuera de Rango Alto	21.00	**400 Bad Request**	Validación de límites de nota
Dentro de Rango	15.50	201 Created	

b) Prueba de Bloqueo por Cierre de Mes

Se verificó el flujo de cierre utilizando el *endpoint* **POST /cierre-mes/cerrar** y luego intentando registrar una nota.

1. Se ejecutó **POST /cierre-mes/cerrar** para un mes específico, estableciendo el estado a **CERRADO**.
2. Se ejecutó **POST /notas/** para el mismo mes. El servidor devolvió el código HTTP **409 Conflict**, impidiendo la modificación de notas y confirmando la restricción de bloqueo de edición.

c) Captura de Ejecución de Pruebas



- **Base de datos**

Fase de Prueba	Función Probada	Propósito de la Prueba	Resultado

Creación de Entidades	registrar_grado(...)	Verifica si se puede crear una plantilla de grado (ej. "Primer Grado").	El grado "Primer Grado" se registra exitosamente en la tabla grado.
Creación de Entidades	registrar_profesor(...)	Prueba el registro de tres profesores diferentes, verificando la inserción y las validaciones de DNI, correo, etc.	Los tres profesores se registran en la tabla profesor.
Creación de Entidades	registrar_padre(...)	Prueba el registro de un apoderado.	El apoderado se registra exitosamente en la tabla padre.
Configuración del Año Escolar	registrar_grado_trabajado(...)	Crea un salón (ej. "Sección A") usando el IdGrado 1 ("Primer Grado") y asignando inicialmente el IdTutor 1 ("Ana Gomez").	El salón "Sección A" se crea en grado_trabajado, vinculado al IdGrado 1 y IdTutor 1.
Configuración del Año Escolar	registrar_asignatura(...)	Define las plantillas de los cursos (Aritmética, Literatura, Geometría) para el IdGrado 1.	Las tres asignaturas se registran en la tabla asignatura vinculadas al IdGrado 1.
Configuración del Año Escolar	registrar_asignatura_trabajada(...)	Asigna los cursos al salón. Por ejemplo, (1, 1, 1) asigna la "Sección A" al profesor "Ana Gomez" para dictar "Aritmética".	Se crean los registros en asignatura_trabajada, vinculando salón, profesor y asignatura.
Configuración del Año Escolar	asignar_tutor_a_salon(...)	Prueba la función de UPDATE para asignar (o re-asignar) al profesor 1 como tutor del salón 1.	El registro del salón 1 en grado_trabajado se actualiza correctamente con el IdTutor 1.

Configuración del Año Escolar	registrar_alumno(...)	Registra a dos alumnos de prueba en el salón 1 ("Sección A") y los asocia al padre 1 ("Luis Solano").	Los dos alumnos se registran en la tabla alumno, asociados al IdGrado_trabajado 1 y al IdPadre 1.
Flujo de Boletas	crear_boleta_vacia(...)	Prueba la creación automática de la "cabecera" de la boleta para el alumno 1 y el alumno 2.	Se crean dos nuevos registros en la tabla boleta, uno para el IdAlumno 1 y otro para el IdAlumno 2.
Flujo de Boletas	actualizar_detalle_boleta(...)	Simula el registro de promedios bimestrales (ej. 18). Esta función también prueba el recálculo automático de la Nota_Final del curso.	Se crea/actualiza una fila en boleta_detalle. Se registra el 18 en la columna del bimestre y la columna Nota_Final se recalcula automáticamente.
Flujo de Boletas	actualizar_boleta_con_promedio_final(...)	Prueba el registro del promedio general final (ej. 16) en la cabecera de la boleta 1.	El registro de la boleta con IdBoleta 1 se actualiza, estableciendo Promedio_Final_General en 16.
Consulta de Datos	obtener_notas_de_bimestre(...)	Prueba la función que recupera todas las notas simples de un alumno en un curso y bimestre específico.	La función devuelve una lista de tuplas (ej. [('Nota1', 17.5), ('Nota8', 17.5)]).
Consulta de Datos	obtener_notas_finales_alumno(...)	Prueba la función que lee el detalle de la boleta para mostrar el promedio final por asignatura (ej. "Aritmética: 18.00").	La función devuelve una lista de tuplas (ej. [('Aritmética', 18.00), ('Literatura', 16.00)]).
Consulta de Datos	obtener_promedio_final_alumno(...)	Comprueba si se puede leer correctamente el Promedio_Final_General (ej. 16) que se registró en la boleta del alumno 1.	La función devuelve el valor 16.00 (o el valor guardado en la boleta del alumno 1).

Libretas ugel + bimestral

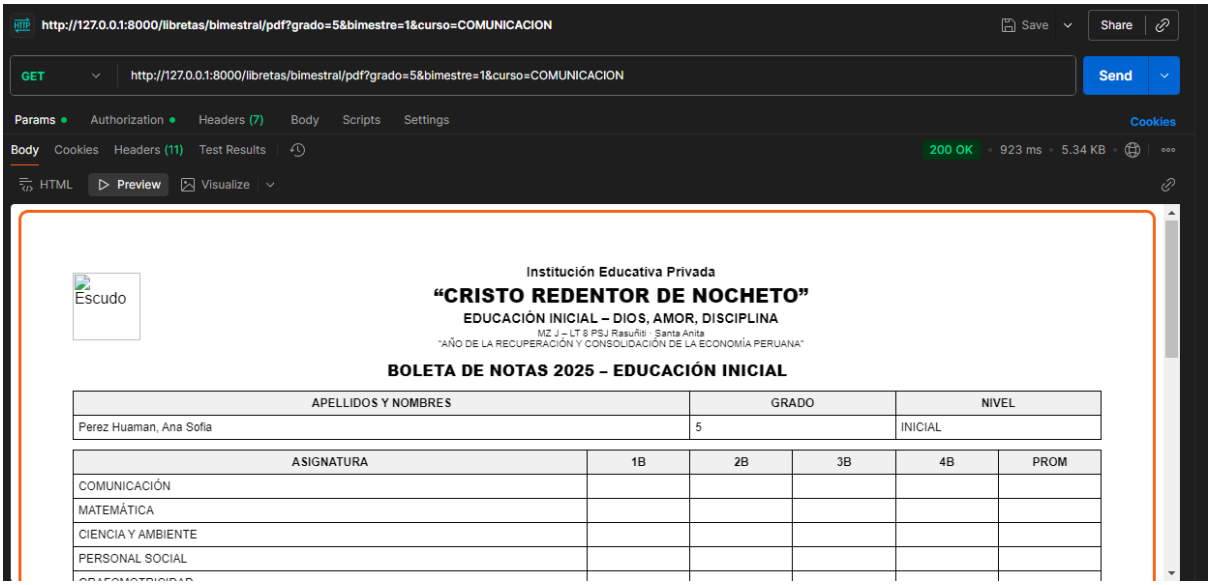
Prueba ejecutada:

Se utilizó Postman para probar el servicio de generación de libretas bimestrales (endpoint /libretas/bimestral/pdf).

Resultado:

El sistema devolvió correctamente el archivo PDF correspondiente al curso y bimestre seleccionados.

→ Inicial



→ Primaria

→ Secundaria

XI. Porcentaje Global de Avance y Próximos Hitos

Módulo 1 — Frontend (React UI)

El módulo de interfaz de usuario presenta un **avance del 75 %**, abarcando las vistas principales de los tres roles (Directora/Director, Tutor y Polidocente). Se implementaron componentes reutilizables del *design system* (Card, Button, Input, Table) que aseguran coherencia visual y accesibilidad AA. Las pantallas de gestión UGEL, libretas y reportes cuentan con diseño responsivo y validaciones visuales.

Próximos hitos: completar la integración dinámica de datos con el backend, optimizar los estados visuales de carga y desarrollar la exportación real de PDF/Excel desde la UI.

Módulo 2 — Backend (Accesos y Permisos)

El módulo alcanza un **85 % de avance**, con la autenticación y autorización por roles totalmente implementadas mediante *middlewares* y *guards* en Django Rest Framework. Se consolidó la gestión de usuarios y la delegación de permisos hacia tutores para UGEL.

Próximos hitos: concluir la interfaz para delegación de UGEL en la UI y ampliar la auditoría de acciones críticas (cierres y exportaciones) para trazabilidad completa.

Módulo 3 — Backend (Notas Mensuales, Cierre y Examen Bimestral)

Este módulo presenta un **avance del 80 %**, con los modelos, lógica de negocio y endpoints funcionales para el cálculo de promedios mensuales y bimestrales (regla 50-50). Las validaciones de rango y bloqueo de edición ya fueron probadas con éxito.

Próximos hitos: completar las pruebas de integración con el frontend y automatizar el registro de cierres en los reportes de auditoría.

Módulo 4 — Backend (Consolidación de Libretas)

Con un **avance del 75 %**, el módulo incluye los modelos y servicios para la consolidación de notas bimestrales y UGEL, además de las equivalencias numéricas y literales. Las pruebas unitarias garantizan la precisión en los cálculos.

Próximos hitos: integrar la consolidación directa con los datos reales del backend y optimizar la detección de precondiciones antes de generar los reportes finales.

Módulo 5 — Generación de Libretas (PDF y UGEL)

El desarrollo del módulo de libretas presenta un **avance del 60 %**, con la generación automática de PDFs por nivel educativo (Inicial, Primaria, Secundaria) a través de WeasyPrint y la parametrización completa de la grilla UGEL.

Próximos hitos: vincular las vistas generadas con el módulo de consolidación y mejorar el diseño visual de las plantillas para impresión oficial.

Módulo 6 — Base de Datos (PostgreSQL)

La base de datos se encuentra en un **95 % de avance**, con todas las tablas creadas y las funciones de inserción, actualización y cálculo de promedios completamente operativas.

Próximos hitos: afinar la función de cálculo del orden de mérito y consolidar la tabla de comentarios UGEL predeterminados para los reportes finales.