Universidad de la Sierra Sur

Licenciatura en Informática

Primer Parcial

Sistema para la Planificación de Exámenes

Tecnologías Web II

Grupo: 706
Profesor: Irving Ulises Hernández Miguel

Integrantes del equipo:

- Jorge López López
- Yhosmar Beltrán Martínez Hernández
- Karen Citlali Pacheco Rodríguez
- Reyder Javier Tineo Tineo

Octubre de 2025, Miahuatlán de Porfirio Díaz, Oaxaca

Índice

1. Introducción

El presente informe documenta el avance significativo alcanzado en el desarrollo del Sistema de Gestión de Horarios de Exámenes, un proyecto diseñado para optimizar y automatizar la programación de evaluaciones académicas en instituciones educativas. Hasta este punto, se ha establecido una base tecnológica sólida que garantiza la escalabilidad, mantenibilidad y robustez del sistema.

El proyecto adopta una arquitectura cliente-servidor moderna, implementando separación de responsabilidades entre las diferentes capas: un backend desarrollado con FastAPI (Python 3.13) que expone servicios RESTful, un frontend construido con React y TypeScript que proporciona una interfaz de usuario intuitiva, y una base de datos PostgreSQL 16 que asegura la persistencia confiable de la información académica. La containerización mediante Docker garantiza consistencia entre entornos de desarrollo, pruebas y producción, facilitando el despliegue y la colaboración del equipo.

La implementación actual se enfoca en establecer los fundamentos arquitectónicos críticos, incluyendo la definición del modelo de datos, la configuración de la infraestructura y la preparación del entorno de desarrollo, sentando las bases para el desarrollo iterativo de funcionalidades específicas en fases posteriores.

2.- Stack Tecnológico

El proyecto utiliza un conjunto moderno de tecnologías que garantizan rendimiento, escalabilidad y facilidad de mantenimiento a largo plazo.
El desarrollo se basa en una arquitectura cliente-servidor, donde FastAPI (Python) implementa el backend y React gestiona la interfaz de usuario en el frontend, ambos ejecutados dentro de contenedores Docker para asegurar portabilidad y consistencia entre entornos.
Internamente, el backend sigue una arquitectura por capas, que separa la aplicación en niveles de presentación (API), lógica de negocio y acceso a datos, facilitando la organización del código, las pruebas unitarias y la evolución del sistema sin afectar las demás partes.

2.1 Lenguaje y Entorno

- **Lenguaje principal:** Python 3.11+
  Elegido por su soporte asincrónico, estabilidad y compatibilidad directa con FastAPI.
- **Gestor de dependencias:** *Poetry* (gestión avanzada de entornos virtuales y dependencias).

2.2 Framework Principal – FastAPI

FastAPI es el framework base del backend, orientado a APIs rápidas y seguras.
Características clave:

- Soporte nativo para asincronía (`async/await`).
- Documentación automática mediante **Swagger** y **OpenAPI**.
- Validación de datos robusta con **Pydantic**.
- Integración sencilla con ORMs, autenticación y middlewares.

Se utiliza para manejar toda la lógica del sistema, exponer endpoints REST y coordinar la comunicación con el frontend.

2.3 ORM y Conectividad con Base de Datos

- **ORM:** *SQLAlchemy 2.x*, encargado de mapear los modelos y facilitar las operaciones CRUD.

- **Conector:** controlador de PostgreSQL nativo (`psycopg2` o `asyncpg`).
- **Migraciones:** *Alembic* (considerado para etapas posteriores).

2.4 Autenticación

El sistema implementa autenticación basada en **JWT (JSON Web Token)** para asegurar las rutas protegidas.
Librerías propuestas:

- `python-jose` o `PyJWT` para creación y validación de tokens.
- `passlib` para encriptar contraseñas.
- Middleware personalizado para verificar tokens y roles de usuario.

Este enfoque permite sesiones sin estado y facilita la integración con clientes web o móviles.

2.5 Pruebas

Framework: *Pytest*
Objetivo: asegurar la estabilidad de los endpoints y modelos principales antes de integrar funcionalidades avanzadas.
Se usarán bases de datos temporales (SQLite o PostgreSQL de prueba) para evitar alterar datos reales durante el testing.

2.6 Despliegue y Contenedores

- **Docker:** encapsula tanto el backend como la base de datos.
- **Docker Compose:** coordina la ejecución de servicios y redes internas.
- **Servidor de aplicación:** *Uvicorn* para desarrollo o *Gunicorn + UvicornWorker* en producción.
- **Proxy reverso:** *NGINX* (Propuesto, para despliegue productivo o acceso en red local).

2.7 Frontend – React Stack

2.7.1 Framework Base

React 18+, enfocado en la creación de una interfaz SPA (Single Page Application) que consume los servicios REST del backend.

2.7.2 Estado y Comunicación

- **TanStack Query (React Query):** para manejo de peticiones, cache y sincronización con la API.
- **Zustand o Redux Toolkit:** para el control de estado global (autenticación, configuración, datos compartidos).

2.7.3 Estilo / UI

- **TailwindCSS:** sistema de estilos rápido y eficiente.
- **Shadcn UI** o **Material UI (MUI):** biblioteca de componentes visuales modernos y personalizables.

2.7.4 Ruteo

- **React Router v6:** para navegación interna, vistas públicas y privadas, control de acceso y rutas dinámicas.

2.7.5 Integración con Backend

- Comunicación mediante **Axios** o **Fetch**, gestionada con TanStack Query.
- Configuración de variables de entorno (`VITE_API_URL`) para definir el endpoint de la API.
- Intercambio de datos en formato **JSON** mediante endpoints REST desarrollados con FastAPI.

3. Base de Datos

3.1. Tecnología y Especificaciones Técnicas

- **Motor de Base de Datos**: PostgreSQL 16
- **Tipo**: Base de datos relacional
- **Arquitectura**: Esquema normalizado con tablas de caché y tablas propias (Propuesta)

3.2. Esquema de Tablas de Caché (Datos Externos)

3.2.1. Tabla `periodos`

- **Función**: Almacenamiento de periodos académicos desde sistema externo
- **Fuentes**: `/api/periodo/lista`, `/api/periodo/actual`
- **Clave primaria**: `id_periodo` (autoincremental)
- **Restricciones**: `clave_periodo` única, no nula

3.2.2. Tabla `carreras`

- **Función**: Catálogo de carreras vigentes
- **Fuente**: `/api/carreras/vigentes`
- **Clave primaria**: `id_carrera` (autoincremental)
- **Campos críticos**: `clave_carrera` única, `vigente` booleano

3.2.3. Tabla `aulas`

- **Función**: Inventario de espacios físicos disponibles
- **Fuentes**: `/api/aulas`, `/api/aulas/capacidad/{cantidad}`
- **Clave primaria**: `id_aula` (autoincremental)
- **Tipos**: AULA, SALA, LABORATORIO (SALA = área de salud)

3.2.4. Tabla `grupos`

- **Función**: Grupos académicos por periodo y carrera
- **Fuentes**: `/api/grupos/periodo={periodo}`, `/api/grupos/lista-carrera`

- **Clave primaria**: `id_grupo` (autoincremental)
- **Índice único**: `(clave_grupo, id_periodo)`

3.2.5. Tabla `profesores`

- **Función**: Catálogo de profesores con información de licenciatura
- **Fuente**: Sistema externo + `/api/horarios/{periodo}/{idprofesor}`
- **Clave primaria**: `id_profesor` (autoincremental)

3.2.6. Tabla `materias_grupo`

- **Función**: Materias asociadas a cada grupo
- **Fuente**: `/api/horarios/{periodo}/grupo/{idGrupo}/materias`
- **Clave primaria**: `id_materia_grupo` (autoincremental)

3.3. Tablas Críticas del Sistema

3.3.1. Tabla `horarios_clases`

- **Función**: Almacenamiento de horarios regulares para detección de conflictos
- **Fuentes**: Múltiples endpoints de horarios por grupo, profesor y aula
- **Importancia crítica**: Detección de horas de inglés (`es_ingles`)
- **Índices**: Optimizados para búsquedas por profesor, aula y materia

3.4. Tablas Propias del Sistema

3.4.1. Tabla `tipos_examen`

- **Función**: Catálogo interno de tipos de examen
- **Tipos**: PARCIAL, ORDINARIO, EXTRAORDINARIO, ESPECIAL
- **Configuraciones**: Duración, requisito de sinodal, permisos de edición

3.4.2. Tabla `examenes` (PRINCIPAL)

- **Función**: Almacenamiento de horarios de exámenes generados

- **Estados**: BORRADOR, PROPUESTO, APROBADO_JEFE, APROBADO_SERVICIOS, RECHAZADO, REALIZADO
- **Relaciones**: Materia, aula, profesor aplicador, profesor sinodal
- **Índices**: Optimizados para validaciones de disponibilidad

3.4.3. Tabla `conflictos_horarios`

- **Función**: Registro automático de conflictos detectados
- **Tipos**: PROFESOR_OCUPADO, AULA_OCUPADA, GRUPO_CON_CLASE, AFECTA_INGLES, CAPACIDAD_INSUFICIENTE
- **Seguimiento**: Estado de resolución y auditoría

3.5. Tablas de Gestión y Seguridad

3.5.1. Tabla `usuarios`

- **Función**: Control de acceso y roles del sistema
- **Roles**: ADMIN, JEFE_CARRERA, SERVICIOS_ESCOLARES, SECRETARIA, COORDINADOR_ACADEMIAS
- **Seguridad**: Autenticación por email y password_hash

3.5.2. Tabla `disponibilidad_profesores`

- **Función**: Restricciones manuales de disponibilidad de profesores
- **Tipos**: NO_DISPONIBLE, PREFERENCIA, COMISION
- **Uso**: Complementa la validación automática de horarios de clases

3.5.3. Tabla `log_aprobaciones`

- **Función**: Trazabilidad completa de cambios y aprobaciones
- **Acciones**: CREAR, MODIFICAR, APROBAR, RECHAZAR, CANCELAR
- **Auditoría**: Registro de estados anteriores y nuevos

3.6. Tablas Auxiliares Especializadas

3.6.1. Tabla `exclusiones_ingles`

- **Función**: Identificación y exclusión de horarios de inglés
- **Origen**: Generada automáticamente desde `horarios_clases` (es_ingles=true)
- **Propósito**: Evitar programación de exámenes en horas de inglés

3.7. Relaciones Clave y Restricciones

3.7.1. Integridad Referencial

- **Relación examenes-profesores**: Validación de misma licenciatura y sin clases
- **Relación examenes-aulas**: Asignación desde API de aulas libres
- **Relación grupos-periodos**: Unicidad por periodo académico

3.7.2. Restricciones de Negocio

- **Profesor aplicador**: Misma carrera + sin clases en horario
- **Profesor sinodal**: Solo para exámenes extraordinarios/especiales
- **Horarios inglés**: Protegidos contra programación de exámenes

3.8. Estrategia de Sincronización

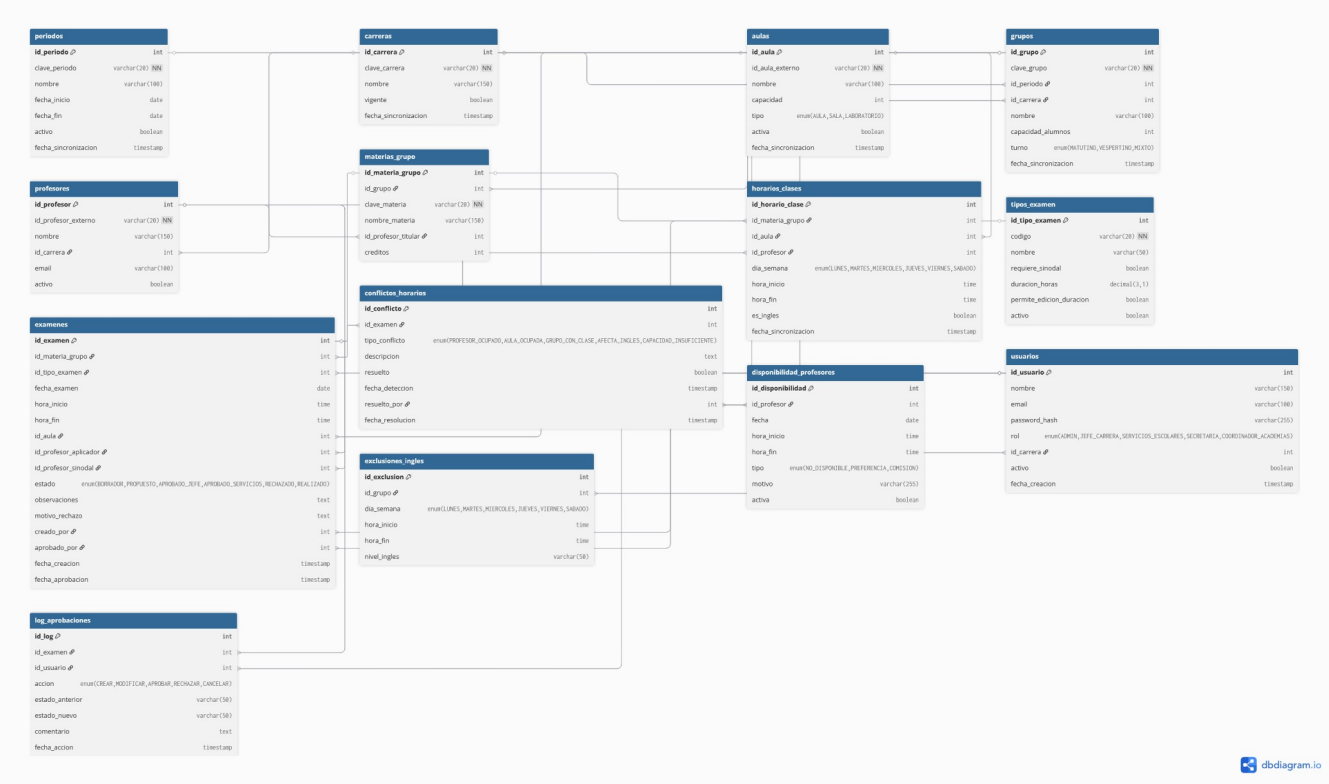
3.8.1. Mecanismos de Actualización

- **Campos de auditoría**: `fecha_sincronizacion` en tablas de caché
- **Control de vigencia**: Campos `activo` y `vigente`
- **Actualizaciones incrementales**: Basadas en `fecha_sincronizacion`

3.8.2. Manejo de Datos Externos

- **Claves externas**: `clave_periodo`, `clave_carrera`, `id_aula_externo`
- **Mapeo interno**: Conversión a IDs internos autoincrementales
- **Consistencia**: Validación de existencia antes de operaciones

3.8.3 Diagrama ER



4. Configuración de Docker

4.1. Archivo Docker Compose

```
version: '3.8'


services:
  # Base de datos PostgreSQL 16
  postgres:
    image: postgres:16-alpine
    container_name: horarios_postgres
    restart: unless-stopped
    environment:
      POSTGRES_USER: horarios_admin
      POSTGRES_PASSWORD: horarios_2025_secure
      POSTGRES_DB: horarios_examenes
      POSTGRES_INITDB_ARGS: "--encoding=UTF-8 --lc-collate=es_MX.UTF-8 --lc-ctype=es_MX.UTF-8"
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - horarios_network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U horarios_admin -d horarios_examenes"]
      interval: 10s
      timeout: 5s
      retries: 5


  # Backend FastAPI con Poetry
```

```
api:
  build:
    context: .
    dockerfile: dockerfile.dockerfile
  container_name: horarios_api
  restart: unless-stopped
  ports:
    - "8000:8000"
  env_file:
    - env.env
  environment:
    # Override para Docker internal networking
    DATABASE_URL: postgresql://horarios_admin:horarios_2025_secure@postgres:5432/horarios_examenes
  volumes:
    # Montar código fuente para desarrollo futuro
    - ./app:/app/app
    - ./alembic:/app/alembic
  depends_on:
    postgres:
      condition: service_healthy
  networks:
    - horarios_network
  # Comentado hasta que exista código - hasta que se cree app/main.py
  # command: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload

# Adminer - Gestor web de BD
adminer:
  image: adminer:latest
  container_name: horarios_adminer
```

```
    restart: unless-stopped
    ports:
      - "8080:8080"
    environment:
      ADMINER_DEFAULT_SERVER: postgres
      ADMINER_DESIGN: dracula
    networks:
      - horarios_network
    depends_on:
      - postgres

volumes:
  postgres_data:
    driver: local

networks:
  horarios_network:
    driver: bridge
```

4.2. Dockerfile (dockerfile.dockerfile)

```
FROM python:3.13-slim

# Metadatos
LABEL maintainer="isYeibby <sYeibbyS@outlook.com>"
LABEL description="Sistema de Horarios de Exámenes - Backend API"
LABEL version="0.1.0"

# Variables de entorno
ENV PYTHONUNBUFFERED=1 \
```

```
    PYTHONDONTWRITEBYTECODE=1

# Instalar dependencias del sistema
RUN apt-get update && apt-get install -y \
    gcc \
    postgresql-dev \
    curl \
    && rm -rf /var/lib/apt/lists/* \
    && apt-get clean

# Establecer directorio de trabajo
WORKDIR /app

# Copiar archivos de dependencias
COPY pyproject.toml ./

# Instalar Poetry
RUN pip install poetry

# Instalar dependencias del proyecto
RUN poetry install --no-root --no-dev

# Crear estructura básica de directorios
RUN mkdir -p /app/app /app/alembic /app/logs

# Crear usuario no-root para seguridad
RUN useradd -m -u 1000 appuser && chown -R appuser:appuser /app
USER appuser
```

```
# Exponer puerto
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=30s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1

# Comando por defecto (se sobreescribirá en desarrollo)
CMD ["echo", "Backend listo - Ejecuta: poetry run uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload"]
```

4.3. Archivo de Variables de Entorno (env.env)

```
# ============================================
# CONFIGURACIÓN BASE DE DATOS
# ============================================
DATABASE_URL=postgresql://horarios_admin:horarios_2025_secure@postgres:5432/horarios_examenes

# ============================================
# SEGURIDAD JWT
# ============================================
JWT_SECRET_KEY=clave_secreta_para_desarrollo_cambiar_en_produccion_2025
JWT_ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30

# ============================================
# ENTORNO
# ============================================
ENVIRONMENT=development
DEBUG=true
LOG_LEVEL=INFO
```

```
# ==========================================
# CORS - Orígenes permitidos
# ==========================================
ALLOWED_ORIGINS=http://localhost:5173,http://localhost:3000

# ==========================================
# APIs EXTERNAS (Por definir)
# ==========================================
EXTERNAL_API_BASE_URL=http://localhost:3000/api
EXTERNAL_API_TOKEN=token_por_definir
EXTERNAL_API_TIMEOUT=30

# ==========================================
# CONFIGURACIÓN APLICACIÓN
# ==========================================
APP_NAME=horarios-backend
APP_VERSION=0.1.0
```

4.4. Comandos de Verificación

```
# En la carpeta HORARIOS-BACKEND/

# 1. Levantar servicios
docker-compose up -d

# 2. Verificar que los contenedores estén corriendo
docker-compose ps

# 3. Verificar base de datos (via Adminer)
```

```
# Abrir navegador: http://localhost:8080
# Servidor: postgres
# Usuario: horarios_admin
# Contraseña: horarios_2025_secure
# Base de datos: horarios_examenes

# 4. Ver logs si es necesario
docker-compose logs postgres

# 5. Detener servicios
docker-compose down
```

5. Acceso en Red Local

5.1. Configuración de Red para Sala de Cómputo

5.1.1. IP de la PC: 132.18.53.78

docker-compose.override.yml (crear este archivo nuevo):

```
version: '3.8'


services:
  api:
    ports:
      - "8000:8000"
      - "132.18.53.78:8000:8000"  # Bind específico para IP de sala
    environment:
      ALLOWED_ORIGINS: http://localhost:5173,http://132.18.53.78:5173,http://192.168.1.100:5173


  postgres:
    ports:
      - "5432:5432"
      - "132.18.53.78:5432:5432"  # Exponer PostgreSQL en IP de sala


  adminer:
    ports:
      - "8080:8080"
      - "132.18.53.78:8080:8080"  # Exponer Adminer en IP de sala
```

Actualizar env.env (agregar estas líneas):

```
# ==========================================
# CONFIGURACIÓN RED LOCAL
# ==========================================
```

SERVER_HOST=132.18.53.78

SERVER_PORT=8000

CORS actualizado para red local

ALLOWED_ORIGINS=http://localhost:5173,http://132.18.53.78:5173,http://192.168.1.100:5173

5.2. Verificación de Conectividad

Desde la PC servidor (132.18.53.78):

Verificar que los servicios escuchan en la IP correcta

sudo netstat -tulpn | grep 132.18.53.78

o

ss -tulpn | grep 132.18.53.78

Verificar todos los puertos en escucha

sudo netstat -tulpn

o

ss -tulpn

Probar conectividad local a los puertos

nc -zv 132.18.53.78 8000

nc -zv 132.18.53.78 5432

nc -zv 132.18.53.78 8080

Verificar procesos Docker que están escuchando

docker ps --format "table {{.Names}}\t{{.Ports}}"

Desde otro equipo en la red:

Probar conectividad básica

ping 132.18.53.78

```
# Probar puertos específicos desde equipo remoto
nc -zv 132.18.53.78 8000
nc -zv 132.18.53.78 5432
nc -zv 132.18.53.78 8080

# Probar con curl (para servicios HTTP)
curl -I http://132.18.53.78:8080
curl -I http://132.18.53.78:8000
```

5.3. URLs de Acceso desde Red Local

5.3.1. Endpoints Disponibles

Servicio	URL de Acceso	Descripción	Estado
API Backend	`http://132.18.53.78:8000`	Servicio principal	No implementado aún
Documentación	`http://132.18.53.78:8000/docs`	Swagger UI	No disponible
Base de Datos	`132.18.53.78:5432`	PostgreSQL	No disponible
Adminer	`http://132.18.53.78:8080`	Gestor web de BD	No disponible

5.3.2. Configuración para Clientes Frontend

Para uso futuro en HORARIOS-FRONTEND/src/:

```
// src/config/api.ts
export const API_CONFIG = {
  BASE_URL: 'http://132.18.53.78:8000',
  TIMEOUT: 30000,
  ENDPOINTS: {
    AUTH: '/api/v1/auth',
    USUARIOS: '/api/v1/usuarios',
    PERIODOS: '/api/v1/periodos',
```

```
    CARRERAS: '/api/v1/carreras',
    AULAS: '/api/v1/aulas',
    GRUPOS: '/api/v1/grupos',
    PROFESORES: '/api/v1/profesores',
    HORARIOS: '/api/v1/horarios',
    EXAMENES: '/api/v1/examenes'
  }
};

// Ejemplo de cliente HTTP
import axios from 'axios';

export const apiClient = axios.create({
  baseURL: API_CONFIG.BASE_URL,
  timeout: API_CONFIG.TIMEOUT,
  headers: {
    'Content-Type': 'application/json',
  },
});
```

5.3.3. Verificación desde Navegador

Desde cualquier equipo en la red local:

1. **Adminer (Base de Datos):**
    - URL: `http://132.18.53.78:8080`
    - Sistema: **PostgreSQL**
    - Servidor: **postgres**
    - Usuario: **horarios_admin**
    - Contraseña: **horarios_2025_secure**

- Base de datos: **horarios_examenes**

2. **API Backend:**
    - URL: `http://132.18.53.78:8000` (No responderá hasta implementar código)
    - Documentación: `http://132.18.53.78:8000/docs` (No disponible aún)

5.4. Comandos de Implementación

Para activar la configuración de red:

```
# En la carpeta HORARIOS-BACKEND/


# 1. Iniciar servicios con override de red

docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d


# 2. Verificar binding de puertos

docker-compose ps


# 3. Probar desde otro equipo

# Abrir navegador y visitar: http://132.18.53.78:8080
```

Esta configuración permitirá que otros equipos en la red accedan a Adminer , cuando se implemente el código, de la API.

6.- Organización del Proyecto

El proyecto se gestionará mediante **dos repositorios separados**, uno para el backend y otro para el frontend, con el fin de mantener una separación clara de responsabilidades y facilitar la colaboración entre los integrantes del equipo.

- **Repositorio Backend:** HORARIOS-BACKEND
- **Repositorio Frontend:** HORARIOS-FRONTEND

6.1 Estructura de Carpetas

Cada repositorio seguirá una **estructura organizada por capas**, especialmente en el backend, para separar de manera clara la **lógica de presentación, la lógica de negocio y el acceso a datos**. Esto permitirá un desarrollo modular, pruebas unitarias más sencillas y escalabilidad futura.

Ejemplo de estructura propuesta para el backend (Propuesta sencilla):

```
/backend
 ├── api/            # Rutas y controladores FastAPI
 ├── core/           # Entidades, modelos de dominio
 ├── use_cases/      # Casos de uso / lógica de negocio
 ├── infrastructure/ # Persistencia, conexión a la base de datos, servicios externos
 └── tests/          # Pruebas unitarias y de integración
```

Para el frontend (Propuesta sencilla), se mantendrá la estructura típica de proyectos **React**, separando componentes, vistas, estados globales y servicios de comunicación con el backend:

```
/frontend
 ├── src/
 │   ├── components/  # Componentes reutilizables
 │   ├── pages/       # Vistas principales
 │   ├── store/       # Estado global (Zustand o Redux)
 │   ├── services/    # Comunicación con API (Axios / TanStack Query)
 │   └── styles/      # Archivos CSS / Tailwind
 └── public/
```

6.2 Aplicación de Clean Architecture

En el backend se busca **experimentar con Clean Architecture**, intentando que las capas internas (entidades y lógica de negocio) permanezcan independientes de frameworks y bases de datos.
Esto implica que:

- La lógica central del sistema no dependerá directamente de FastAPI ni de PostgreSQL.
- Las capas externas (API y persistencia) interactuarán con la lógica interna únicamente a través de **interfaces o adaptadores**, lo que permitirá cambios futuros en el framework, motor de base de datos o librerías sin afectar la funcionalidad central.

Con esta organización se asegura un desarrollo más **ordenado, mantenible y escalable**, capaz de adaptarse a futuras necesidades y crecimiento del proyecto.

7. Conclusión

El avance del proyecto demuestra una planificación técnica sólida y una ejecución metódica que ha permitido establecer los cimientos esenciales para el desarrollo exitoso del Sistema de Horarios de Exámenes. Se ha configurado exitosamente un entorno de desarrollo completo y reproducible mediante Docker, que incluye PostgreSQL 16 para la gestión de datos, FastAPI para los servicios backend y una estructura frontend preparada para React con TypeScript o solamente JS.

Entre los logros más significativos se destacan: el diseño detallado del esquema de base de datos que modela con precisión los requisitos del dominio académico, la implementación de una arquitectura por capas que facilita el mantenimiento y la escalabilidad, la configuración de redes locales para acceso multiplataforma, y la definición de flujos de trabajo con Poetry para la gestión de dependencias. Estas decisiones técnicas proporcionan una base robusta que soportará el desarrollo incremental de funcionalidades complejas como la generación automática de horarios, la detección de conflictos y los flujos de aprobación.

El proyecto se encuentra en una posición óptima para avanzar hacia la fase de implementación de funcionalidades específicas, con una infraestructura que promueve las mejores prácticas de desarrollo, facilita las pruebas automatizadas y asegura la calidad del código. La documentación técnica generada servirá como guía de referencia para el equipo de desarrollo y garantizará la continuidad del proyecto en sus siguientes iteraciones.