**INFORME DE PROYECTO: GENERADOR INTELIGENTE DE HORARIOS UNIVERSITARIO**

**Gabriel Alejandro Mantilla Clavijo**

**Melany Marcela Saez Acuña**

**Eddy Josue Lara Cermeno**

**Julio de Jesús Denubila Vergara**

**FUNCIONALIDADES**

Este proyecto está diseñado para ayudar a los estudiantes de la Universidad Tecnológica de Bolívar a crear horarios académicos óptimos de manera automática. Sus funcionalidades principales son:

* **Gestión de Materias:**
  + Búsqueda Inteligente: Permite buscar materias por nombre o código.
  + Selección de Materias: Los usuarios pueden agregar y quitar materias de una lista de selección.
  + Conteo de Créditos: El sistema calcula y muestra en tiempo real los créditos acumulados, advirtiendo si se excede el límite recomendado.
* **Generación de Horarios:**
  + Algoritmo de Combinaciones: Utiliza un algoritmo de backtracking para encontrar todas las combinaciones de clases posibles que no tengan conflictos de horario entre sí.
  + Fusión de NRCs: Agrupa horarios que son funcionalmente idénticos (misma materia, profesor y bloque horario) pero que tienen diferentes NRCs, simplificando la visualización.
* **Filtrado Avanzado y Optimización:**
  + Filtro por Profesor: Para cada materia, el usuario puede elegir incluir o excluir profesores específicos.
  + Filtro de Disponibilidad: Permite bloquear días y horas en los que el usuario no desea tener clases.
  + Optimización de Horarios: El usuario puede priorizar los resultados para:
  + Minimizar huecos: Encontrar horarios con el menor tiempo libre entre clases.
  + Maximizar días libres: Encontrar horarios que se concentren en la menor cantidad de días posible.
* **Visualización y Exportación:**
  + Vista de Cuadrícula: Muestra todos los horarios generados en una cuadrícula comparativa para una fácil revisión.
  + Contador de Resultados: Indica el número total de horarios válidos encontrados.
  + Vista Detallada: Ofrece una vista ampliada y detallada de un horario seleccionado, mostrando la distribución semanal y una lista de las clases con su información (profesor, NRC, etc.).
  + Exportación de Horarios: Permite descargar el horario seleccionado en formatos PDF y Excel (.xlsx).

**ARQUICTECTURA GENERAL**

El proyecto sigue una arquitectura cliente-servidor moderna de tres capas, un patrón robusto y escalable que separa claramente las responsabilidades, separando la lógica de la interfaz.

1. Capa de Presentación (Frontend): La interfaz con la que interactúa el usuario, desarrollada en Flutter.
2. Capa de Lógica de Negocio (Backend): El servidor API desarrollado en FastAPI (Python), que procesa todas las solicitudes y contiene la lógica central.
3. Capa de Datos (Base de Datos): El sistema de gestión de base de datos PostgreSQL, que almacena y sirve los datos brutos.

Componentes:

* **Backend (Servidor):**
  + Lenguaje: Python
  + Framework API: FastAPI, para crear una API RESTful rápida y eficiente.
  + Base de Datos: PostgreSQL, para almacenar toda la información de materias, cursos y profesores.
  + Conector de BD: psycopg para la comunicación entre Python y PostgreSQL.
  + Validación de Datos: Pydantic, para la validación y serialización de los datos de la API.
  + Entorno: Docker y Docker Compose, para la contenerización de la aplicación y la base de datos, asegurando un entorno de desarrollo y despliegue consistente.
* **Frontend (Cliente):**
  + Framework: Flutter, para crear una interfaz de usuario multiplataforma (web, Windows, etc.) desde una única base de código.
  + Lenguaje: Dart.
  + Gestión de Estado: StatefulWidget y setState para la gestión del estado de la interfaz.
  + Comunicación API: Paquete http para realizar las peticiones al backend.
  + Pruebas (Testing):
  + Framework: pytest y requests para realizar pruebas de integración y funcionales a los endpoints del backend.

**FLUJO DE DATOS**

1. Interacción del Usuario (Frontend):

* El usuario selecciona varias materias en el SubjectsPanel.
* Opcionalmente, abre el FilterWidget y establece filtros (ejemplo: excluir un profesor, bloquear horas).
* Finalmente, hace clic en el botón "Generar Horarios" en el MainActionsPanel.

1. Petición HTTP (Frontend -> Backend):

* El ApiService en Flutter recopila los códigos de las materias seleccionadas y los filtros en un objeto.
* Este objeto se convierte a formato JSON.
* Se envía una petición POST al endpoint /api/schedules/generate del backend, con el JSON en el cuerpo de la solicitud.

1. Procesamiento (Backend):

* Endpoint (FastAPI): Recibe la petición. Pydantic valida automáticamente que el JSON recibido coincida con el modelo GenerateScheduleRequest.
* Servicio (schedule\_generator): El endpoint invoca a la función principal del generador de horarios, pasándole los datos validados.
* Repositorio (repository.py): El servicio de generación necesita los datos de todas las clases disponibles para las materias seleccionadas. Llama a funciones del repositorio (ejemplo: get\_classes\_for\_subjects).
* Base de Datos (PostgreSQL): El repositorio traduce la llamada a una consulta SQL y la ejecuta contra la base de datos. La base de datos devuelve las filas correspondientes.
* Algoritmo: Con los datos de las clases, el algoritmo en schedule\_generator.py ejecuta su lógica de backtracking, aplicando los filtros para descartar combinaciones no válidas y realizando las optimizaciones solicitadas (minimizar huecos, etc.).
* El resultado es una lista de horarios válidos (una lista de listas de ClassOption).

1. Respuesta HTTP (Backend -> Frontend):

* El endpoint de FastAPI toma la lista de horarios generados, la convierte a formato JSON y la envía de vuelta al frontend como respuesta a la petición POST, típicamente con un código de estado 200 OK.

1. Visualización (Frontend):

* El ApiService recibe la respuesta JSON y la decodifica, creando una lista de objetos Dart.
* El estado de la aplicación se actualiza con los horarios recibidos.
* El widget ScheduleGridWidget se reconstruye y muestra la nueva cuadrícula con las previsualizaciones de cada horario generado. El contador de horarios también se actualiza.

**PRUEBAS (TESTING)**

En este proyecto, las pruebas son fundamentales para garantizar la fiabilidad y correctitud del backend, que es el cerebro de la aplicación. Dado que el frontend (Flutter) depende completamente de los datos que la API le proporciona, asegurar que la API se comporte de manera predecible y correcta es crucial para que toda la aplicación funcione bien.

Las pruebas automatizadas actúan como una red de seguridad. Cada vez que se realiza un cambio en el código del backend (por ejemplo, al optimizar el algoritmo o añadir un nuevo filtro), se pueden ejecutar todas las pruebas para verificar instantáneamente que las funcionalidades existentes no se han roto y que las nuevas funcionan como se espera.

El proyecto se centra en Pruebas de Integración a nivel de API. Analicemos qué significa esto:

* ¿Qué son? En lugar de probar una función pequeña y aislada (lo que sería una prueba unitaria), estas pruebas verifican la interacción de varios componentes del sistema a la vez. En este caso, una sola prueba simula una petición HTTP completa desde el frontend hacia el endpoint /api/schedules/generate.
* ¿Qué involucran? Cuando se ejecuta una prueba, ocurren los siguientes pasos:
  1. Se levanta la aplicación FastAPI.
  2. La prueba envía una petición HTTP (un POST) a la URL de la API.
  3. La API recibe la petición, la valida con los modelos de Pydantic.
  4. Llama al servicio schedule\_generator.
  5. El servicio consulta la base de datos a través del repository.
  6. El algoritmo procesa los datos, aplica los filtros y genera los horarios.
  7. La API devuelve una respuesta JSON.
  8. La prueba recibe esta respuesta y la analiza.

Al probar todo este flujo, se asegura que la "integración" entre el endpoint, los servicios, los modelos y la base de datos funciona correctamente.

¿Cómo aseguran que todo funciona correctamente? Las pruebas en el archivo test\_filters.py están diseñadas para validar reglas de negocio específicas y casos límite. Lo hacen mediante un patrón de "Preparar, Actuar, Afirmar".

1. Simulación de Escenarios Reales: Cada función de prueba define un escenario que un usuario podría encontrarse. Por ejemplo:

* test\_filter\_exclude\_professor: Simula a un usuario que quiere horarios de "Cálculo Diferencial" pero que explícitamente no quiere ver clases con el profesor "Juan Pérez".
* test\_filter\_unavailable\_slots: Simula a un usuario que dice "no quiero tener clases los viernes por la tarde".

1. Verificación de Resultados (Aserciones): Después de enviar la petición a la API, la prueba realiza aserciones (verificaciones) sobre la respuesta recibida. Esto es lo que realmente confirma si la lógica funcionó.

* Ejemplo con test\_filter\_exclude\_professor: Actuar: Enviar la petición con el filtro para excluir a "Juan Pérez". Afirmar (Assert): La prueba recorre todos los horarios generados en la respuesta y verifica que ninguna de las clases (ClassOption) tenga a "Juan Pérez" como profesor. Si encuentra, aunque sea una, la prueba falla, indicando un error en la lógica del filtro.
* Ejemplo con test\_filter\_unavailable\_slots: Actuar: Enviar la petición con el filtro que bloquea los viernes de 14:00 a 18:00. Afirmar (Assert): La prueba revisa cada horario devuelto y se asegura de que no contenga ninguna clase programada en ese día y rango de horas. Si encuentra una, la prueba falla.

1. Cobertura de Casos Límite y Regresiones: Las pruebas también cubren:

* Casos Positivos: ¿El filtro de incluir profesor funciona? (ejemplo: test\_filter\_include\_professor). La aserción sería que todas las clases de esa materia sean con el profesor seleccionado.
* Casos Vacíos: ¿Qué pasa si los filtros son tan restrictivos que no hay ningún horario posible? La prueba afirmaría que la respuesta es una lista vacía, lo cual es un resultado correcto.
* Regresiones: Si en el futuro alguien modifica el algoritmo y rompe accidentalmente la lógica de los filtros, estas pruebas fallarán inmediatamente, alertando al desarrollador del problema antes de que llegue a los usuarios.