TRABAJO PRÁCTICO INTEGRADOR

MATERIA: Programación Avanzada

PROFESOR: Gianluca Piriz

COMISIÓN: 4

TEMA: Aplicaciones Web con Frameworks Python

ALUMNOS: Celina Pereyra, Junior Flores, Viviana Enriquez, Nancy Salvatierra.

Índice

Aplicaciones Web con Frameworks Python

1.La elección de "Aplicaciones Web con Frameworks Python" para nuestro TPI se basó en el interés por el **desarrollo web** y la **versatilidad de Python**, relevante en la industria por tener cualidades que facilitan el desarrollo y la creación de sistemas mantenibles.

1.1Proyecto elegido: *Clear Pass*

El proyecto ***Clear Pass*** se originó como un trabajo previo, lo que nos permitió profundizar en principios avanzados de programación. Seleccionamos ***Flask*** que nos permitió construir una arquitectura flexible que permite expandir funcionalidades sin afectar la base del proyecto, permitiendo agregar módulos de autenticación, integración con bases de datos, efectos visuales sin comprometer el rendimiento, estructura clara para organizar el código, reutilizar componentes, facilidad en el mantenimiento para futuras mejoras y optimización para un diseño responsivo en cualquier dispositivo.

El objetivo de *Clear Pass* es crear una plataforma segura y transparente de importadores. Abordando la falta de veracidad en la información, proporcionando un ranking de verificados para la compra de productos.

Este trabajo nos brindó la oportunidad de poder aplicar y demostrar los conocimientos de *Programación Orientada a Objetos (POO)*, Diseño Orientado a Objetos (DOO) y *Patrones de Diseño*. La complejidad del proyecto nos desafió a implementar una arquitectura limpia, modular y basada en principios fundamentales de la ingeniería de software.

1.2 Metodología de desarrollo

Para llevar a cabo este proyecto, comenzamos identificando y documentando los requerimientos funcionales y no funcionales, dejando en claro los objetivos y la funcionalidad esperada. Esta etapa fue clave para establecer una base sólida de guía para el desarrollo.

A continuación, elaboramos distintos diagramas —de casos de uso, de clases y de secuencia— que nos permitieron visualizar la arquitectura del sistema y planificar cómo se integrarían las distintas partes del código.

Adoptamos una dinámica de trabajo colaborativa, dividiéndonos tareas específicas y realizando reuniones semanales para revisar avances tanto en el frontend como en el backend. Para el diseño de la interfaz utilizamos ***Figma***, lo que nos permitió prototipar la maquetación inicial y definir visualmente la experiencia de usuario. El desarrollo del frontend se implementó en ***HTML.5*** *para la estructura de las paginas web*. ***CSS*** *para los estilos y el framework Tailwind CSS para un desarrollo de interfaz de usuario rápido, altamente personalizable y responsivo. Se optó por Tailwind sobre otros frameworks CSS por su control granular sobre el estilo y la minimización del CSS personalizado.* ***Jinja2:*** *Motor de plantillas integrado con Flask, utilizado para renderizar HTML dinámico y pasar datos desde el backend al frontend.* ***JavaScript (ES6 Modules):*** *Para la lógica interactiva del lado del cliente, con un enfoque modular (*`auth.js`*,* `main.js`*,* `products.js`*,* `ui.js`*).* ***Google Identity Services (GSI):*** *Para la integración del inicio de sesión con cuentas de Google.*

Para el desarrollo del backend se implementó en **Python** el lenguaje de programación principal. **Flask:** Micro-framework web para el desarrollo del servidor y las APIs. Elegido por su ligereza, flexibilidad y la capacidad de imponer una estructura modular orientada a objetos.

La base de datos fue muy debatible por el equipo, si optar por una base de datos SQL o una simple pero funcional como JSON, así que viendo las características del proyecto y lo flexible del mismo optamos por realizarlo con archivos **JSON** (data.json) que en un momento lo usamos como una simulación de base de datos local para los usuarios. Además, también implementamos una **Api Externa** para la simulación de productos y demostración de la integración entre servicios de terceros.

Organizamos nuestras tareas a través de ***Trello***, una herramienta que facilitó la asignación y el seguimiento de actividades semanales. Las reuniones de equipo se llevaron a cabo mediante ***Discord***, donde pudimos compartir pantalla y utilizar un servidor exclusivo para almacenar información de referencia. Además, mantuvimos un grupo de ***WhatsApp*** como canal de comunicación ágil para coordinar encuentros y resolver dudas rápidamente.

2.Diseño y estructura del código

2.1 Uso de la Programación Orientada a Objetos (POO):

La arquitectura de nuestro proyecto "Clear Pass" se fundamenta en principios sólidos de la Programación Orientada a Objetos (POO) y la aplicación estratégica de patrones de diseño. Esto nos ha permitido construir un sistema modular, escalable y mantenible.

2.2 Clases y Objetos: Hemos definido clases claras para representar entidades y responsabilidades dentro del sistema:

`-UserRepository`: Gestiona las operaciones de persistencia de datos para usuarios (obtener, añadir).

`-ProductRepository`: Gestiona las operaciones de persistencia/acceso a datos para productos (obtener todos, obtener por ID).

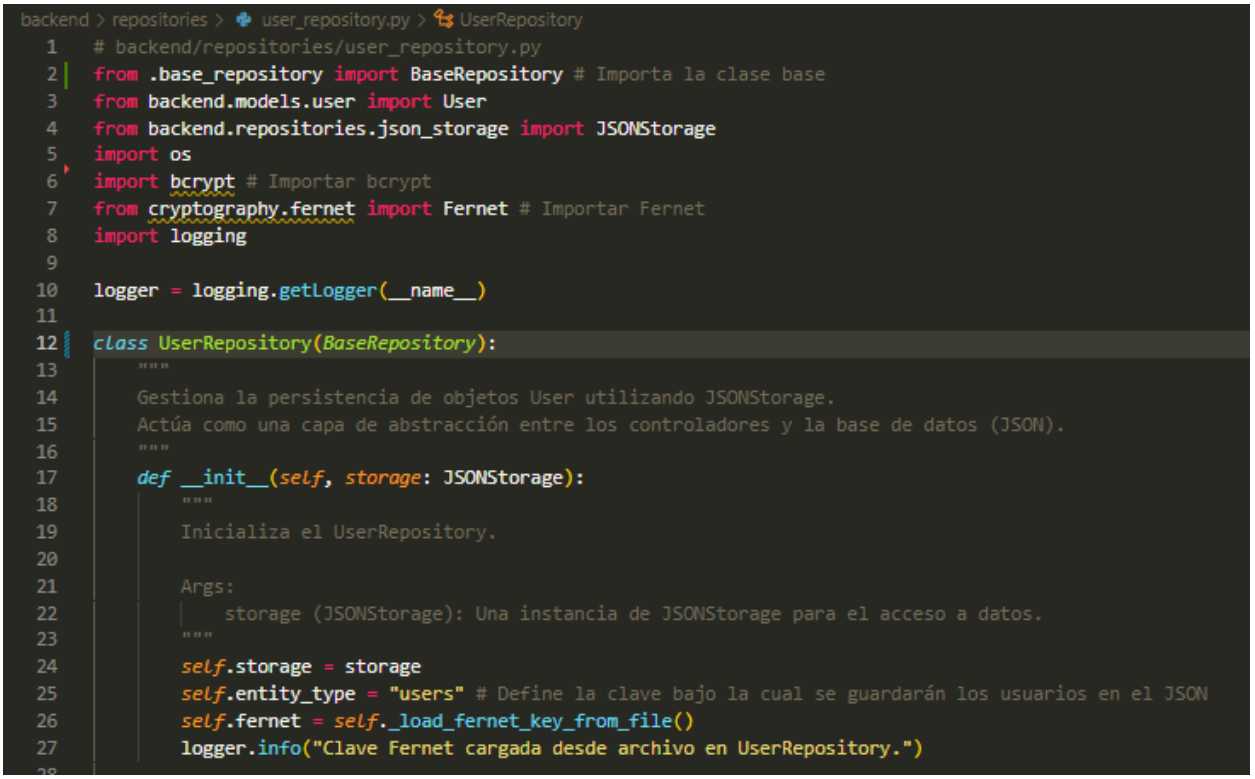
`-ExternalProductService`: Se encarga de la lógica para interactuar con APIs externas de productos.

`-JSONStorage`: Proporciona una interfaz para la lectura y escritura genérica de datos en un archivo JSON.

`-AuthController`: Contiene la lógica de negocio relacionada con la autenticación de usuarios (registro, login, integración con Google).

`-ProductController`: Contiene la lógica de negocio relacionada con la gestión de productos (búsqueda, detalles).

2.3 Encapsulamiento: Cada clase encapsula sus datos y la lógica interna para manipularlos, exponiendo solo una interfaz pública a través de sus métodos. Por ejemplo, `UserRepository` sabe cómo guardar un usuario, pero un controlador no necesita saber si eso implica escribir en un JSON o una base de datos.



```python
    def add_user(self, user: User) -> User | None:
        logger.info(f"Intentando añadir usuario: {user.email}")

        # Verificar si el email ya existe
        if self.find_user_by_email(user.email):
            logger.warning(f"Intento de registro con email existente: {user.email}")
            return None

        logger.info("Email no encontrado en el repositorio. Procediendo con el registro.") # Nuevo log

        # Hash de la contraseña si se proporciona
        if user.password:
            logger.info("Hasheando contraseña...") # Nuevo log
            try:
                hashed_password = bcrypt.hashpw(user.password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')
                user.password = hashed_password
                logger.info("Contraseña hasheada exitosamente.") # Nuevo log
            except Exception as e:
                logger.error(f"Error al hashear la contraseña: {e}")
                return None # Fallo en el hasheo
        else:
            logger.info("No se proporcionó contraseña (posiblemente registro por Google).") # Nuevo log

        # Encriptar el email
        if user.email:
            logger.info("Encriptando email...") # Nuevo log
            try:
                encrypted_email = self._encrypt_email(user.email)
                user.email = encrypted_email
                logger.info("Email encriptado exitosamente.") # Nuevo log
            except Exception as e:
                logger.error(f"Error al encriptar el email: {e}")
                return None # Fallo en la encriptación
        else:
            logger.info("No se proporcionó email para encriptar.") # Nuevo log

```

```python
        # Encriptar el email
        if user.email:
            logger.info("Encriptando email...") # Nuevo log
            try:
                encrypted_email = self._encrypt_email(user.email)
                user.email = encrypted_email
                logger.info("Email encriptado exitosamente.") # Nuevo log
            except Exception as e:
                logger.error(f"Error al encriptar el email: {e}")
                return None # Fallo en la encriptación
        else:
            logger.info("No se proporcionó email para encriptar.") # Nuevo log

        # Guardar en JSONStorage a través de save_entity
        logger.info("Convirtiendo objeto User a diccionario para guardar...") # Nuevo log
        user_data = user.to_dict()
        logger.info("Objeto User convertido a diccionario. Procediendo a guardar en JSONStorage.") # Nuevo log

        # Este es el punto crucial: la llamada a save_entity
        try:
            saved_data = self.storage.save_entity(self.entity_type, user_data)
            logger.info("save_entity en JSONStorage completado.") # Nuevo log de éxito
        except Exception as e:
            logger.error(f"ERROR CRÍTICO: Fallo en self.storage.save_entity: {e}") # Nuevo log de error
            return None
```

```python
        if saved_data:
            logger.info(f"Usuario {saved_data.get('email', '[email encriptado]')} añadido al repositorio. Desencriptando para retorno.")
            saved_user_obj = User.from_dict(saved_data)
            # Asegúrate de que el email en el objeto retornado esté desencriptado para el controlador
            # Nota: Si el email está encriptado en saved_data, deberías desencriptarlo aquí.
            # user_obj.email = self._decrypt_email(user_data['email']) # Si from_dict lo carga encriptado

            # Si User.from_dict ya maneja el email encriptado/desencriptado, entonces solo:
            saved_user_obj.email = self._decrypt_email(saved_user_obj.email) # Desencriptar para devolver al controlador
            logger.info(f"Usuario {saved_user_obj.email} guardado y desencriptado para retorno.")
            return saved_user_obj
        else:
            logger.error(f"Fallo al guardar el usuario {user.email}.")
            return None
```

```python
    def get_user_by_id(self, user_id: str) -> User | None:
        """
        Busca un usuario por su ID.

        Args:
            user_id (str): El ID del usuario a buscar.

        Returns:
            User | None: El objeto User si se encuentra, o None si no.
        """
        user_data = self.storage.get_by_id(self.entity_type, user_id)
        if user_data:
            return User.from_dict(user_data)
        return None
```

2.4 Abstracción: Las capas del repositorio (`UserRepository`, `ProductRepository`) actúan como abstracciones, permitiendo que los controladores interactúen con los datos sin conocer los detalles de su origen o almacenamiento. `ProductRepository` no sabe si los productos se leen de una base de datos o una API; solo sabe que puede `get_all_products()`.

backend > repositories > product_repository.py > ProductRepository

```python
# backend/repositories/product_repository.py
import logging
from typing import List, Dict, Any, Optional
from backend.services.external_product_service import ExternalProductService
from backend.models.product import Product # Importar la clase Product

logger = logging.getLogger(__name__)

class ProductRepository:
    """
    Repositorio para gestionar la obtención y almacenamiento (caché) de productos.
    Interactúa con ExternalProductService para obtener datos y los almacena en caché.
    """
    def __init__(self, external_product_service: ExternalProductService):
        self.external_product_service = external_product_service
        self.cache: Dict[int, Product] = {} # Caché en memoria para objetos Product
        logger.info("ProductRepository inicializado con caché en memoria (almacenará objetos Product filtrados).")

```

```python
    def get_all_products(self) -> List[Product]: # Tipo de retorno cambiado a List[Product]
        """
        Recupera todos los productos de la API externa o de la caché.
        Devuelve una lista de objetos Product.
        """
        if not self.cache: # Si la caché está vacía, cargar desde la API externa
            logger.info("Caché de productos vacía. Obteniendo productos de la API externa.")
            raw_products_data = self.external_product_service.get_all_products()
            # Convertir los diccionarios raw a objetos Product y poblar la caché
            for item in raw_products_data:
                product_obj = Product.from_dict(item) # Usa from_dict para crear objeto Product
                self.cache[product_obj.id] = product_obj
            logger.info(f"Caché de productos poblada con {len(self.cache)} productos.")
        else:
            logger.info(f"Productos obtenidos desde la caché. Total: {len(self.cache)}")

        return list(self.cache.values()) # Devolver una lista de objetos Product
```

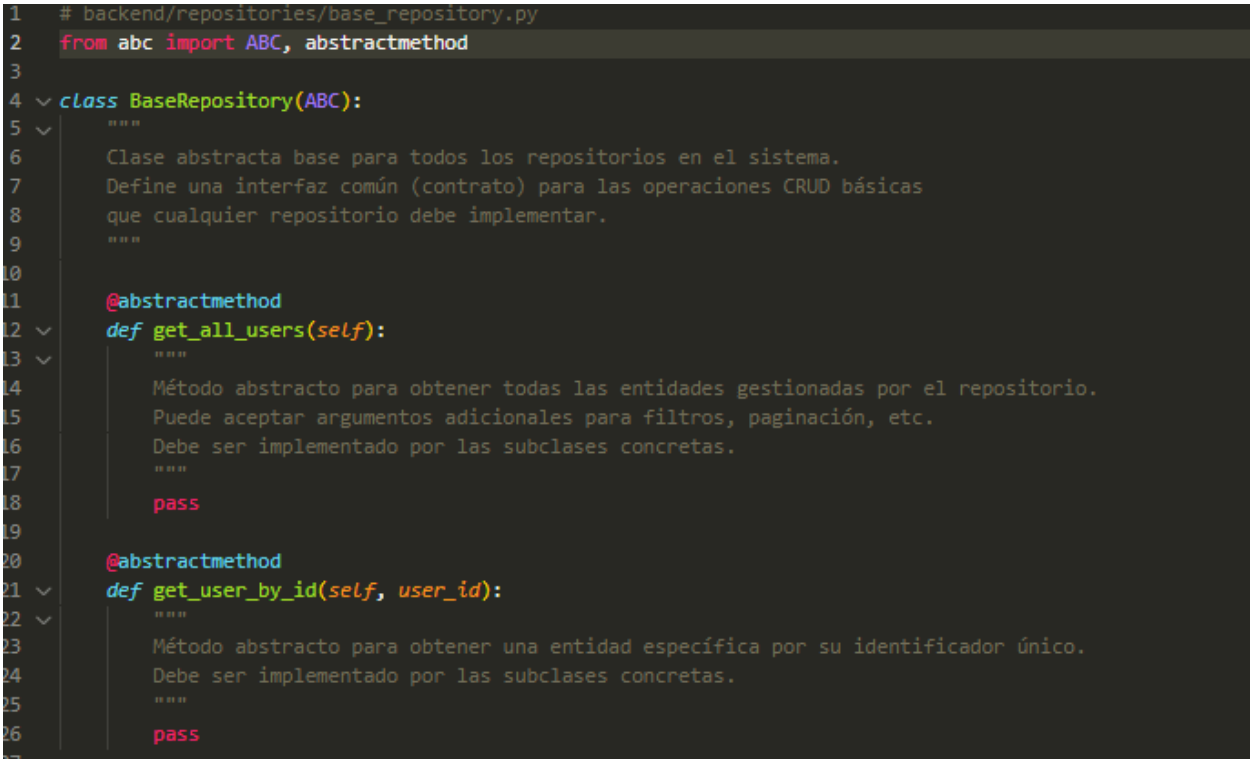
```python
    def get_product_by_id(self, product_id: int) -> Optional[Product]: # Tipo de retorno cambiado a Optional[Product]
        """
        Recupera un producto por su ID de la API externa o de la caché.
        Devuelve un objeto Product.
        """
        product_obj = self.cache.get(product_id)
        if product_obj:
            logger.info(f"Producto con ID {product_id} encontrado en caché.")
            return product_obj
        else:
            logger.info(f"Producto con ID {product_id} no encontrado en caché. Obteniendo de la API externa.")
            raw_product_data = self.external_product_service.get_product_by_id(product_id)
            if raw_product_data:
                product_obj = Product.from_dict(raw_product_data) # Usa from_dict para crear objeto Product
                self.cache[product_obj.id] = product_obj # Añadir a la caché
                logger.info(f"Producto con ID {product_id} obtenido de la API externa y añadido a caché.")
                return product_obj
            logger.warning(f"Producto con ID {product_id} no encontrado en la API externa.")
            return None
```

2.5 Herencia:

Para establecer una estructura común para nuestro componente de acceso a datos, hemos introducido una clase abstracta `BaseRepository`.

`BaseRepository` define métodos abstractos como `get_all()` y `get_by_id()` que la clase `UserRepository` debe implementar.

`UserRepository` hereda de `BaseRepository`, lo que asegura que cumple con una interfaz estándar para obtener datos.



```python
# backend/repositories/base_repository.py
from abc import ABC, abstractmethod

class BaseRepository(ABC):
    """
    Clase abstracta base para todos los repositorios en el sistema.
    Define una interfaz común (contrato) para las operaciones CRUD básicas
    que cualquier repositorio debe implementar.
    """

    @abstractmethod
    def get_all_users(self):
        """
        Método abstracto para obtener todas las entidades gestionadas por el repositorio.
        Puede aceptar argumentos adicionales para filtros, paginación, etc.
        Debe ser implementado por las subclases concretas.
        """
        pass

    @abstractmethod
    def get_user_by_id(self, user_id):
        """
        Método abstracto para obtener una entidad específica por su identificador único.
        Debe ser implementado por las subclases concretas.
        """
        pass
```

```python
# backend/repositories/user_repository.py
from .base_repository import BaseRepository # Importa la clase base
from backend.models.user import User
from backend.repositories.json_storage import JSONStorage
import os
import bcrypt # Importar bcrypt
from cryptography.fernet import Fernet # Importar Fernet
import logging

logger = logging.getLogger(__name__)

class UserRepository(BaseRepository):
```

2.6 Bajo Acoplamiento:

Los controladores (`AuthController`, `ProductController`) no dependen de la implementación específica del almacenamiento de datos. Si en el futuro decidimos cambiar `data.json` por una base de datos PostgreSQL o Firestore, solo necesitaríamos modificar la implementación del repositorio, sin afectar a los controladores. Esto reduce drásticamente el acoplamiento entre la lógica de negocio y la capa de persistencia.

2.7 Alta Cohesión:

Cada repositorio tiene una única responsabilidad: manejar las operaciones de acceso a datos para una entidad específica. Esto mantiene el código organizado y fácil de entender.

3. Patrones de Diseño

La arquitectura de nuestro proyecto sigue un patrón Model-View-Controller (MVC) modificado y aplica otros patrones de diseño para mejorar la organización y las propiedades de POO.

Patrón Model-View-Controller (MVC):

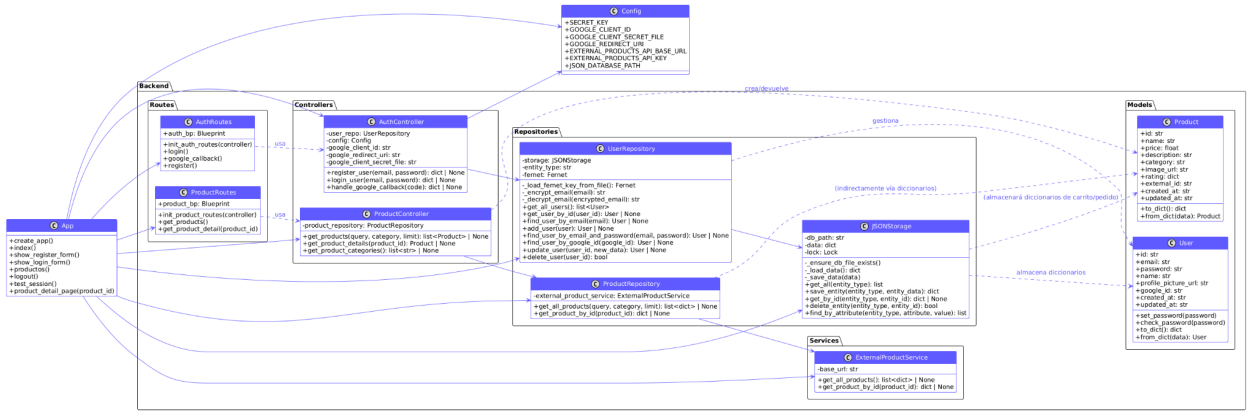
-Model: Representado por las capas de Repositorios (`UserRepository`, `ProductRepository`) y Servicios (`ExternalProductService`, `JSONStorage`). Estos componentes manejan los datos, la lógica de negocio y la persistencia.

-View: Constituyen las plantillas HTML (`.html`) en la carpeta `frontend/templates/`, junto con los archivos CSS y JavaScript que gestionan la presentación y la interactividad del lado del cliente.

-Controller: Los controladores (`AuthController`, `ProductController`) actúan como intermediarios. Reciben las solicitudes del usuario (desde las rutas de Flask), interactúan con los Modelos para obtener o manipular datos, y luego seleccionan la Vista adecuada para renderizar la respuesta.

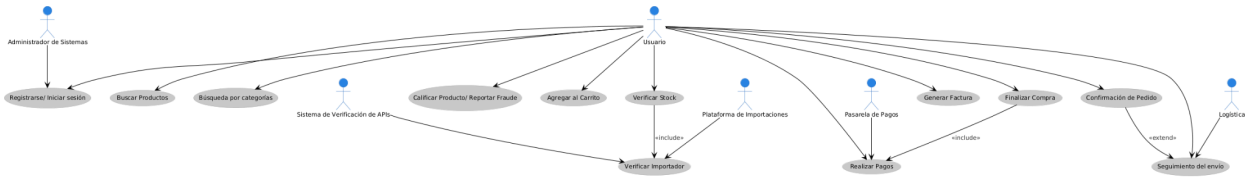
-Rutas (Blueprints): Los Blueprints de Flask (`auth_bp`, `product_bp`) definen los *endpoints* de la API y las rutas que dirigen las solicitudes a los métodos apropiados dentro de los controladores. Actúan como el "puente" entre las Vistas (llamadas JS a la API) y los Controladores.

3.1 Diagrama de Clases (general del proyecto):



Este diagrama representa la arquitectura de un backend, que visualiza la estructura de un sistema. Muestra las clases, atributos, operaciones, y las relaciones entre ellas. Se organiza en paquetes como `Models` (estructuras de datos), `Repositories` (acceso a datos), `Controllers` (lógica de negocio), `Routes` (endpoints API) y `Services` (lógica específica). Este diagrama facilita la comprensión de la organización de los componentes de un programa, ilustrando cómo las diferentes capas colaboran para gestionar usuarios, productos e imágenes.

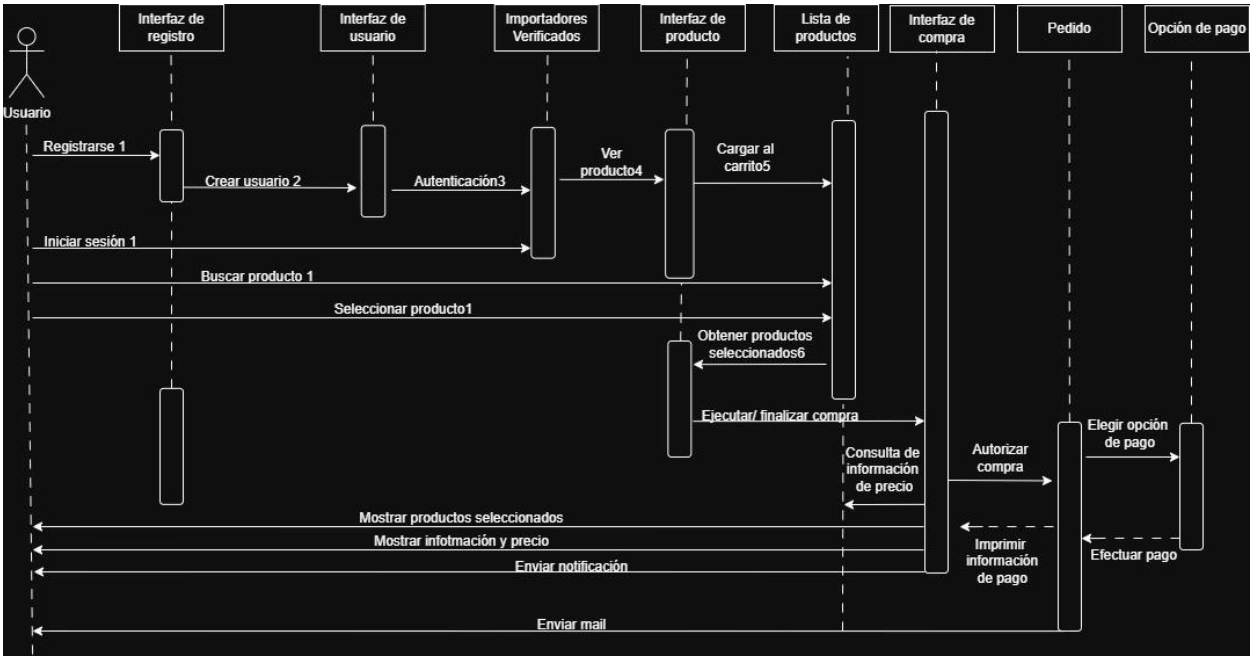
3.2 Diagrama de Uso:



Describe las funcionalidades de un sistema desde la perspectiva del usuario. Muestra los "actores" (roles como "Usuario", "Administrador de Sistemas", "Sistema de Verificación de APIs", "Plataforma de Importaciones", "Pasarela de Pagos", "Logística") y los "casos de uso" (acciones o tareas que los actores pueden realizar, representadas por óvalos como "Buscar Productos", "Agregar al Carrito", "Realizar Pagos").

Este diagrama expone las interacciones de un "Usuario" con el sistema para realizar diversas acciones de compra, desde la búsqueda y agregación de productos hasta la finalización de la compra y el seguimiento del envío. También, casos de uso de administración y la participación de sistemas externos para verificar stock e importadores, o para el procesamiento de pagos y logística. Las relaciones "include" (Verificar Stock incluye Verificar Importador) y "extend" (Seguimiento del envío extiende Confirmación de Pedido) indican dependencias entre los casos de uso.

3.3 Diagrama de Secuencias:



Ilustra la interacción entre objetos en un orden cronológico, en un escenario específico. Las líneas verticales representan los objetos o instancias ("Usuario", "Interfaz de registro", "Importadores Verificados", etc.), mientras que las flechas horizontales indican los mensajes intercambiados.

Este diagrama describe el registro y compra para un usuario. Comienza con el "Usuario" registrándose e iniciando sesión, que involucra las interfaces de registro y usuario, y autenticación. Luego, el usuario busca y selecciona productos, interactuando con la interfaz y lista de productos para obtener la información necesaria. Finalmente, se muestra el proceso de consulta de precios, autorización de la compra, impresión de información, elección y efecto del pago, culminando con notificaciones y envío de correos, detallando la secuencia de mensajes entre los diferentes componentes del sistema para completar estas tareas.

4.Patrones de Diseño Aplicados:

-Patrón Repositorio (Repository Pattern):

Clases como `UserRepository` y `ProductRepository` implementan este patrón. En lugar de que los controladores accedan directamente a `data.json` o realicen llamadas a la API, delegan estas responsabilidades a los repositorios.

```
user_repository = UserRepository(storage=json_storage)
auth_controller = AuthController(user_repository=user_repository, config=Config)
```

```
def login_user(self, email: str, password: str) -> Dict[str, str] | None:
    """
    Autentica a un usuario.
    """
    logger.info(f"Intentando iniciar sesión con email: {email}")
    user = self.user_repository.find_user_by_email_and_password(email, password)
    if user:
        session['user_id'] = user.id
        session['user_name'] = user.name
        session['user_email'] = user.email
        logger.info(f"Usuario {email} inició sesión exitosamente.")
        return {'message': 'Inicio de sesión exitoso.'}
    else:
        logger.warning(f"Intento de inicio de sesión fallido para el email: {email}")
        return {'message': 'Email o contraseña incorrectos.'}
```

-Patrón de Capa de Servicio (Service Layer Pattern):

`ExternalProductService` actúa como una capa de servicio. Su responsabilidad es abstraer los detalles de la comunicación con la API externa (URLs, cabeceras, manejo de respuestas HTTP, etc.). `ProductRepository` utiliza este servicio para obtener los datos crudos. Permite que el `ProductRepository` se enfoque puramente en la lógica de "obtener datos de producto", delegando la complejidad de las llamadas HTTP a un servicio especializado. Esto mejora la reusabilidad del servicio de API por si otras partes del sistema necesitaran interactuar con ella.

```python
def get_product_by_id(self, product_id: int) -> Optional[Product]: # Tipo de retorno cambiado a Optional[Product]
    """
    Recupera un producto por su ID de la API externa o de la caché.
    Devuelve un objeto Product.
    """
    product_obj = self.cache.get(product_id)
    if product_obj:
        logger.info(f"Producto con ID {product_id} encontrado en caché.")
        return product_obj
    else:
        logger.info(f"Producto con ID {product_id} no encontrado en caché. Obteniendo de la API externa.")
        raw_product_data = self.external_product_service.get_product_by_id(product_id)
        if raw_product_data:
            product_obj = Product.from_dict(raw_product_data) # Usa from_dict para crear objeto Product
            self.cache[product_obj.id] = product_obj # Añadir a la caché
            logger.info(f"Producto con ID {product_id} obtenido de la API externa y añadido a caché.")
            return product_obj
        logger.warning(f"Producto con ID {product_id} no encontrado en la API externa.")
        return None
```

4.1 Inyección de Dependencias:

En `app.py`, los objetos (ej., `AuthController`, `ProductController`) no crean sus propias dependencias (ej., `UserRepository`), sino que las reciben a través de sus constructores (`__init__`) al ser instanciados.

Este principio es fundamental para lograr un bajo acoplamiento. Al "inyectar" las dependencias, las clases son más independientes y más fáciles de probar en aislamiento (se pueden sustituir las dependencias reales por "mocks" en las pruebas unitarias).

```python
def create_app():
    app = Flask(
        __name__,
        static_folder=os.path.join('frontend', 'static'),
        template_folder=os.path.join('frontend', 'templates')
    )

    app.config.from_object(Config)

    app.config["SESSION_PERMANENT"] = True
    app.config["SESSION_TYPE"] = "filesystem"
    app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(hours=24)
    Session(app)

    # AÑADIR HTTPS A ORIGINS EN CORS
    CORS(app, supports_credentials=True, origins=["http://localhost:5173", "http://127.0.0.1:5173", "https://127.0.0.1:5000"])

    # --- Inicialización de Repositorios y Controladores (Inyección de Dependencias) ---
    json_storage = JSONStorage(data_file='data.json')
    logger.info("JSONStorage inicializado.")

    external_product_service = ExternalProductService(app.config['EXTERNAL_PRODUCTS_API_BASE_URL'])
    logger.info(f"ExternalProductService instanciado con base_url: {external_product_service.base_url}")
    user_repository = UserRepository(storage=json_storage)
    product_repository = ProductRepository(external_product_service=external_product_service)


    product_controller = ProductController(product_repository=product_repository)
    logger.info("AuthController y ProductController instanciados.")
```

```python
product_controller = ProductController(product_repository=product_repository)
logger.info("AuthController y ProductController instanciados.")

# --- Registro de Blueprints y Inyección de Controladores ---
init_product_routes(product_controller)
app.register_blueprint(product_bp)
logger.info("Blueprint 'product_bp' registrado con prefijo '/api'.")

init_auth_routes(auth_controller)
app.register_blueprint(auth_bp, url_prefix= '/api')
logger.info("Blueprint 'auth_bp' registrado con prefijo '/api'.")
```

Resultados / dificultades

El desarrollo de "Clear Pass" ha sido un proceso de aprendizaje continuo, donde la superación de desafíos técnicos nos ha permitido aplicar y consolidar los conocimientos adquiridos en la materia.

5. Desafíos Relacionados con POO y Arquitectura de Backend

5.1 Gestión del Ciclo de Vida y Coherencia de Objetos (Inyección de Dependencias):

Inicialmente, existía el riesgo de que las clases crearan sus propias dependencias, lo que llevaría a un alto acoplamiento y dificulte la gestión de instancias compartidas (ej., un solo `JSONStorage` para múltiples repositorios). Si no se manejaban correctamente las sesiones de usuario en Flask con un backend modular, podría haber inconsistencias.

Para solucionarlo se implementó una estrategia rigurosa de **Inyección de Dependencias**. Todas las instancias de repositorios, servicios y controladores se crean de forma centralizada en `app.py` y se pasan a sus dependientes a través de sus constructores.

Esto aseguró que el **bajo acoplamiento** se mantuviera en todo el sistema. Las clases son más simples y reusables, ya que no son responsables de crear sus dependencias, sólo de utilizarlas. Facilitó enormemente la **testeabilidad**.

5.2 Abstracción de Fuentes de Datos Múltiples (Patrón Repositorio y Capa de Servicio):

Necesidad de manejar datos de usuarios desde un archivo JSON local (`data.json`) y datos de productos desde una API externa, sin que los controladores tuvieran que lidiar con los detalles de cada fuente.

Se implementó el **Patrón Repositorio** para `UserRepository` y `ProductRepository`.

Para la API externa, se introdujo una **Capa de Servicio** (`ExternalProductService`) que `ProductRepository` consume.

Esta decisión de diseño encapsuló la lógica de persistencia y de comunicación externa, manteniendo los controladores limpios y desacoplados de los detalles de infraestructura. Es un

claro ejemplo de **Abstracción** y **Composición**, ya que los repositorios se componen de servicios de almacenamiento específicos, en lugar de heredar de ellos.

6. Modularización y Separación de Responsabilidades en el Backend (Blueprints y Controladores):

A medida que la aplicación crecía, existía el riesgo de que `app.py` se volviera un archivo monolítico con demasiadas rutas y lógica, dificultando el mantenimiento y la lectura.

Para solucionarlo se utilizó el concepto de **Blueprints de Flask** para agrupar las rutas relacionadas (ej., `auth_bp` para autenticación, `product_bp` para productos). La lógica de negocio fue delegada a clases **Controladoras** (`AuthController`, `ProductController`), que implementan la **Alta Cohesión** al tener responsabilidades únicas.

Esto mejoró significativamente la **organización del código**, la **legibilidad** y la **escalabilidad**. Cada módulo es independiente y fácil de entender, lo que facilita el desarrollo colaborativo y futuras expansiones.

7. Configuración Segura de la Aplicación (HTTPS y CORS):

Al desarrollar una aplicación web, la seguridad de la comunicación es primordial. Configurar HTTPS y las políticas de CORS correctamente es complejo y propenso a errores.

Por eso se configuró Flask para servir a través de HTTPS (`ssl_context`) y se implementó `flask-cors` con orígenes específicos para permitir la comunicación segura entre el frontend (ej., localhost) y el backend.

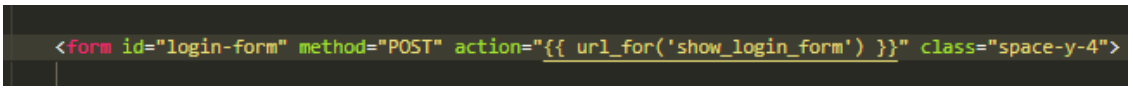
Esto aseguró que la aplicación cumple con estándares básicos de seguridad para la comunicación de datos, un aspecto fundamental en cualquier proyecto web.

8. Desafíos de Integración Frontend-Backend y UI/UX:

8.1 Sincronización de Rutas Flask y Atributos HTML `action` (Errores de `url_for`):

La integración del frontend (plantillas Jinja2) con el backend (rutas Flask) presentó desafíos, específicamente con el `BuildError` relacionado con `url_for()` en los atributos `action` de los formularios. Esto se debía a una sintaxis incorrecta de Jinja2 o un desajuste con los nombres de los *endpoints* de las funciones de vista de Flask.

La solución fue una depuración exhaustiva y ajuste preciso de la sintaxis `action="{{ url_for('show_login_form') }}"` en `login.html` y `register.html`. Validando que el nombre de la función Python (`show_login_form`, `show_register_form`) coincidiera con el *endpoint* referenciado.



8.2 Diseño Responsivo y Estilización Detallada con Tailwind CSS:

Lograr que la interfaz de usuario se viera impecable y con fidelidad al diseño gráfico en múltiples dispositivos (escritorio, tablet, móvil) utilizando un framework CSS *utility-first* como Tailwind, especialmente con el fondo continuo y la personalización de componentes.

Para solucionarlo hicimos una aplicación extensiva de las clases responsivas de Tailwind CSS (`md:`, `flex-col`, `grid-cols-1`, `hidden`/`inline`). Para el color exacto del botón "Registrarse", se definió un color hexadecimal personalizado (`#A299F7`) directamente en `tailwind.config` dentro de la plantilla.

9. Conclusión del tema

Este proyecto demostró cómo la estructura modular y la aplicación de principios como la Separación de Responsabilidades (SRP) y el Patrón Repositorio facilitan la creación de aplicaciones web robustas y escalables. La elección de Flask, un framework ligero y flexible, resultó fundamental para el desarrollo ágil y la implementación de una arquitectura limpia, permitiendo una clara organización del código y la reutilización de componentes. El proyecto valida la eficacia de POO/DOO en un entorno de desarrollo web real, destacando los beneficios de un framework como Flask para construir soluciones eficientes y mantenibles.

10. Conclusión de la cursada

La cursada de "Programación Avanzada" consolidó nuestra comprensión del desarrollo de software, sentando una base sólida en el Paradigma Orientado a Objetos (POO). Desde conceptos esenciales como Clases, Atributos, Operaciones e Interfaces.

El enfoque del curso en Diseño de Aplicaciones OO y las relaciones entre clases como Herencia y Polimorfismo fue crucial para el proyecto final. Esto nos permitió aplicar principios como abstracción y reuso, vitales para construir una arquitectura web robusta y modular. El proyecto brindó la oportunidad de llevar la teoría a la práctica, demostrando la eficacia del POO/DOO en la gestión de complejidad y escalabilidad.

Destacamos la habilidad de pensar en objetos, la relevancia del diseño previo a la codificación y la aplicación de patrones adecuados.

Integrar la teoría en la práctica y abordar requerimientos no funcionales como rendimiento y seguridad fue un desafío. La investigación complementaria y la experimentación fueron clave para superarlos.

En síntesis, el curso nos proveyó de herramientas conceptuales y prácticas para proyectos complejos, priorizando el diseño de calidad y la mantenibilidad del código.