TRABAJO DE GRADO
Opción Seminario-Diplomado.

PIGBU: Plataforma Institucional de Gestión Bibliotecaria Universitaria

Corporación Universitaria Remington.
Facultad de Ingeniería
Ingeniería en Sistemas

Jefferson Javier Salcedo Oliva
Hector Eduardo Betancourth Ibarra
Tutor: Diego Fernando Marín Lozano

Opción de Trabajo de grado Seminario-Diplomado
2025

Dedicatoria

A mis padres

Agradecimientos

Agradecemos a la Corporación Universitaria Remington por su buena labor y a nuestro tutor Diego Fernando Martin Lozano

Tabla de Contenido

Resumen

El trabajo se desarrolló con el fin de dar solución a una problemática que presentan algunas instituciones con sus bibliotecas, así generando una respuesta optima con el desarrollo de un sistema de gestión bibliotecaria, facilitando el acceso de préstamos reservas o alquiler de cualquier libro o material didáctico.

El proyecto fue desarrollado utilizando FastAPI en Python, aprovechando que es mas eficiente y así generando unos servicios de alto desempeño con unas estructuras más ordenadas basadas en microservicios.

Por último, con la parte de la documentación se incorporó documentación de manera automática utilizando la herramienta MkDocs que permite organizar de la forma más clara posible toda la información con relación a lo desarrollado

Palabras clave

Microservicios, gestión bibliotecaria, transformación digital, Python, metodología incremental

Pregunta orientadora de la búsqueda

¿Como puede un sistema de gestión bibliotecaria basada en microservicios mejorar y optimizar las bibliotecas en instituciones o universidades que requieran mejorar sus sistemas ya obsoletos siendo así la manera más optima a la hora de hacer reservas, alquileres o prestamos de algún libro o material didáctico?

Sustentación teórica de la pregunta

La arquitectura que se basa en microservicios mejora la escalabilidad la seguridad del sistema y la integración del mismo ya que son servicios más pequeños es moderna e independiente (Buñay Guisñan, 2024)

Estos microservicios son más pequeños y se basan en hacer una sola cosa, se puede organizar de mejor manera cuando se trata de la lógica de negocio cuando se buscar dividir la aplicación por segmentos (Contreras, 2018)

Considerando que algunas de las características de los microservicios, es posible afirmar que este enfoque abre un amplio campo de posibilidad3es a la hora de seleccionar tecnologías la forma de llevar a cabo esta información y algunas de las herramientas que pueden emplearse para dar una solución optima al problema (Contreras, 2018)

Combinando el uso de Python con FastAPI se logra una aplicación más ágil, eficiente y muy bien estructurada gracias a esto se puede decir que se puede escribir un código más limpio reducir al máximo los fallos y generar de forma más sencilla toda la información (Palacio, 2022)

El desarrollo basado en microservicios se acopla de manera más fácil y natural a los principios de las metodologías agiles, especialmente a esta metodología que manejamos en este proyecto que es la incremental. Cada avance sea mejora o avance es un avance que retroalimenta el código y favorece para que así disminuya la posibilidad de fallos en implementaciones futuras y mejoras del sistema (Battaglia, García, & Congiusti, 2024)

Así mismo se puede decir que ayuda a la colaboración entre equipos, ya que así cada uno puede enfocarse entre un microservicio y trabaja de manera propia su ciclo de desarrollo sí que los demás sufran o tengan daños

La arquitectura basada en microservicios combinado con Docker y PostgreSQL, MongoDB y Python mediante FastAPI se complementan para generar así una infraestructura más robusta para la creación de plataformas digitales más modernas y gracias a estas notamos la evolución de estas herramientas, (Moreno Bernal, 2022)para dar como desarrollo una plataforma adaptable capaz de llevar a cabo con despliegues más agiles un proceso de aprendizaje más eficiente en cuestión de usuario (Franco Noreña, 2022)

Problema

La problemática que hemos mirado a lo largo de este tiempo son la falta de tecnología en los sistemas los cuales quedan ya casi obsoletos sin embargo ese temor de probar las nuevas herramientas para este tipo de material (Romero, 2004)

Hoy en día es una necesidad crear espacios virtuales y modernos para las bibliotecas, sin embargo, son limitadas las que tienen estás actualizaciones. Un ejemplo es la casa de la cultura que ha dado a conocer un stock de biblioteca, que fue lanzada a internet y así se dio a conocer a los usuarios que pueden acceder al contenido de dicho lugar creando subgrupos de personas con características similares que previamente han seleccionado libros. (Coral Bastidas, 2003)

El enfoque que nosotros queremos presentar frente a esta situación es casi similar sin embargo nos enfocamos más en el préstamo y alquiler de libros sin necesidad de hacer grupos o subgrupos de personas.

Metodología

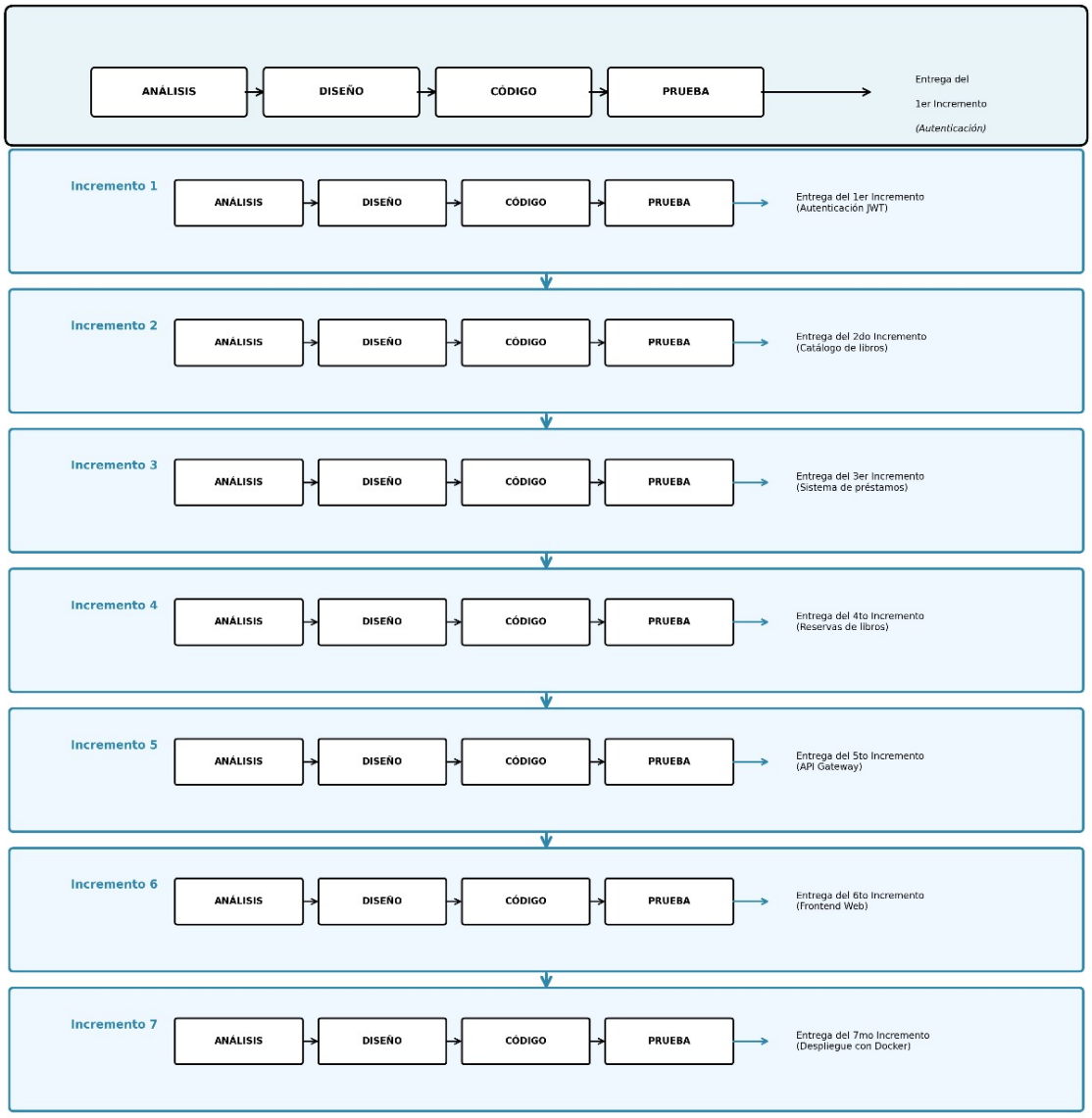
La metodología incremental es un enfoque de desarrollo de software presentado por Lehman en 1984 este modelo surge con la creación de prototipos de la combinación del enfoque cascada permitiendo del producto final de este se forme por partes o módulos y cada una parte de esta denominada **Incrementos** (Marcos E. Aguirre Zurita, 2002).

En este método cada vez que se utiliza un incremento es una nueva versión del anterior ya sea que se incorpore nuevas funciones o mejorando las que ya están así perfeccionando el código el cual equivale a una versión más amplia (Ericka Solano Fernández, 2020)

Este modelo se destaca por la capacidad que tiene de adaptarse a los proyectos en los que los requisitos pueden cambiar a un cierto tiempo, cada ciclo tiene un enfoque fundamental en el sistema y cada uno de estos puede recibir una retroalimentación por parte de los usuarios, lo que nos da las facilidades de ajustar algunos cambios sin necesidad de retrasar el proyecto, ya que debido a los microservicios estos se validan cada uno sin dañar los demás (Jefferson Alexis Paredes Plaza, 2024)

Finalmente la metodología incremental es una gran ayuda para el trabajo en equipo y así los distintos grupos pueden encargar de distintos incrementos de manera simultánea o escalonada, gracia a esto se aumenta la eficacia de los proyectos y mayor especialización en cada uno de los microservicios que se realiza (José Navarro, 2016)

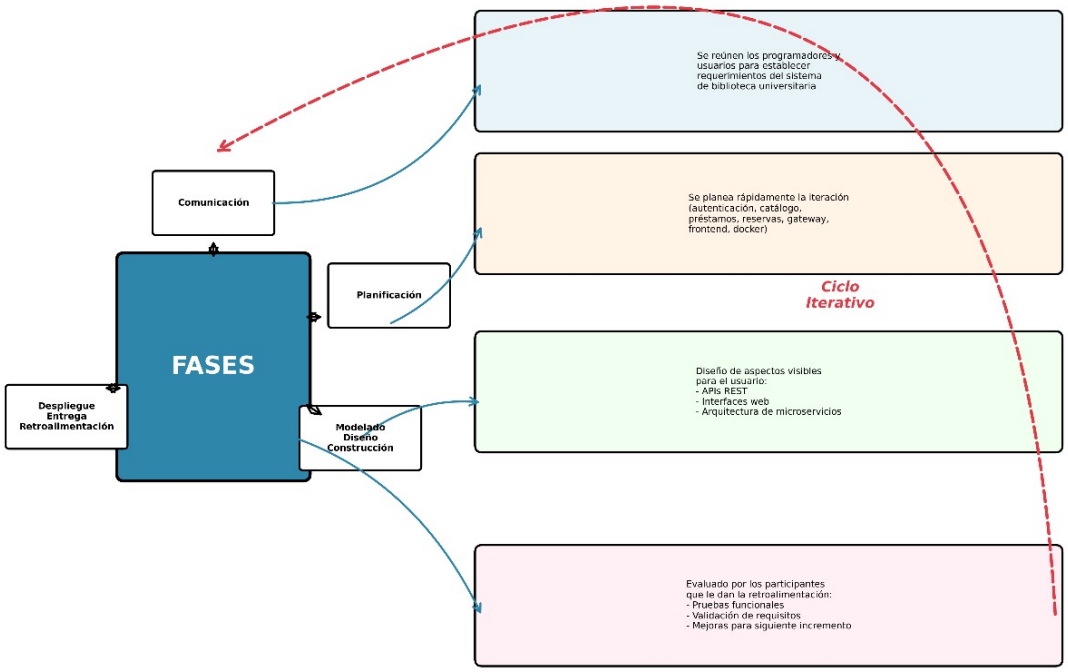
Figura 1 Modelo Incremental



fuente: Jefferson Salcedo, Hector Betancourth

A partir de la problemática que anterior mente se expone se propone el desarrollo de un sistema de gestión bibliotecaria que es construida bajo una arquitectura de microservicios distribuidos. Para esto se utilizó también una mitología ágil ya que se ajusta a la problemática y necesidad de la misma.

Con este enfoque principalmente se quiere facilitar la creación de esta aplicación para que sea escalable, flexible y sencilla de mantenerla además de que es más eficiente para generar lo que se requiere



Fuente: Jefferson Salcedo, Héctor Betancourth

Desarrollo

Para solventar problemática expuesta anteriormente, se propone un sistema basado en microservicios, tales como autenticación, gestión de perfiles, gestión de contenidos o información digital, evaluación y rendimiento estos se desarrollan a través de APIs e interfaz modular, cada uno actuando de manera autónoma sin afectarse entre sí.

Además, estarán aislados mediante contenedores Docker lo cual nos permite que la ampliación incluya todas las herramientas necesarias para ejecutarse de manera fiable y segura en cualquier equipo. Esta estructura facilita detectar fallas e intervenir en el microservicio afectado sin que se vean comprometidos los demás servicios, integrando FastAPI con Python para disminuir duplicidad de código garantizando tiempos de respuesta óptimos

Finalmente, los servicios se pueden incrementar según sea la necesidad sin afectar la estructura general, dando como resultado una aplicación web modular, ágil, escalable y sostenible de cursos en línea.

Comunicación y planificación.

Una vez analizada la problemática que presentan los sistemas de gestión bibliotecarias, se procede a crear diagramas y definir los requisitos para comenzar el desarrollo como se indica en la siguiente tabla.

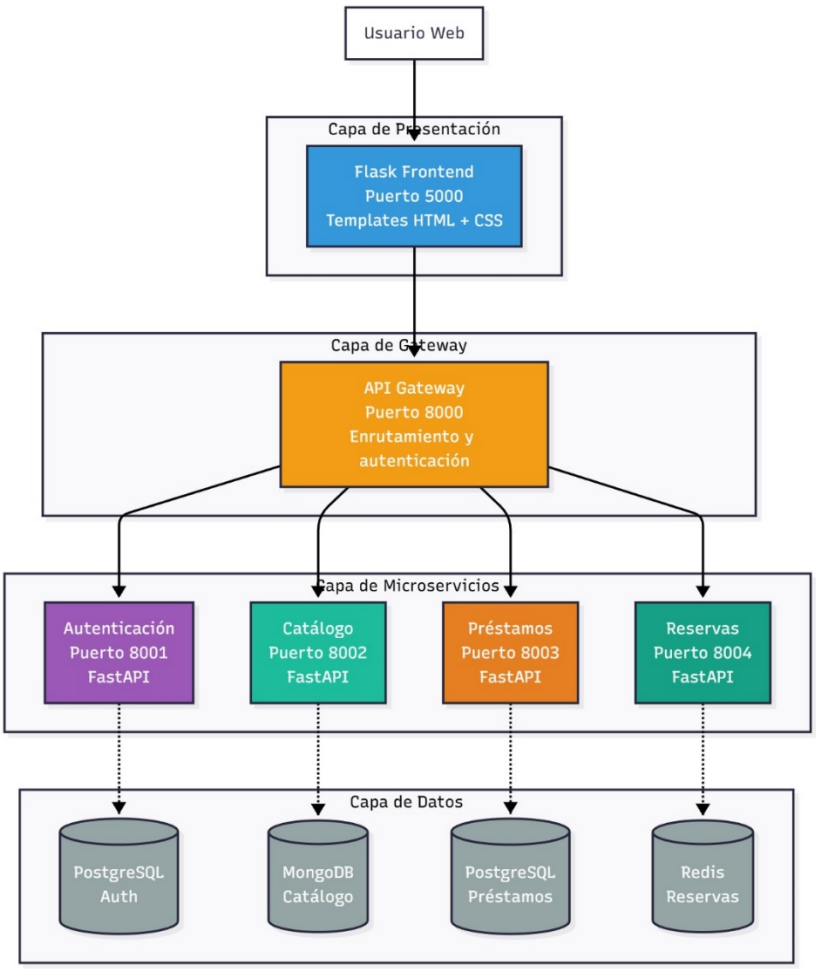
Tabla 1 Tabla de iteraciones o requisitos

Incremento	Desarrollo	Entregable principal	Responsable(s)
Incremento 1	Microservicio de autenticación con FastAPI	Código autenticación JWT, hash de contraseñas, registro/login pruebas	Jefferson salcedo, Hector Betancourth
Incremento 2	Gestión de catalogo de libros	CRUD MongoDB, búsqueda	Jefferson salcedo
Incremento 3	Sistema de prestamos	Prestamos, multas, historial	Jefferson salcedo, Hector Betancourth
Incremento 4	Sistema de reservas	Reservas, notificaciones	Hector Betancourth
Incremento 5	API Gateway	Proxy, health checks, enrutamiento	Hector Betancourth
Incremento 6	Frontend	Flask, HTML	Jefferson salcedo
Incremento 7	Despliegue docker	Docker compose, contenedores, MKDocs	Jefferson salcedo, Hector Betancourth

Diagrama de arquitectura de proyecto.

Se presenta el diagrama de arquitectura del sistema de biblioteca universitaria basado en microservicios como autenticación, catalogo, préstamos y reservas. Se muestra el funcionamiento con sus servicios implementados con FastAPI, el usuario a través del Frontend que se comunica con el API Gateway, cada microservicio usa su propia base de datos empleando PostgreSQL, MongoDB y Redis. (ver figura 3).

Figura 3 Diagrama de arquitectura de proyecto

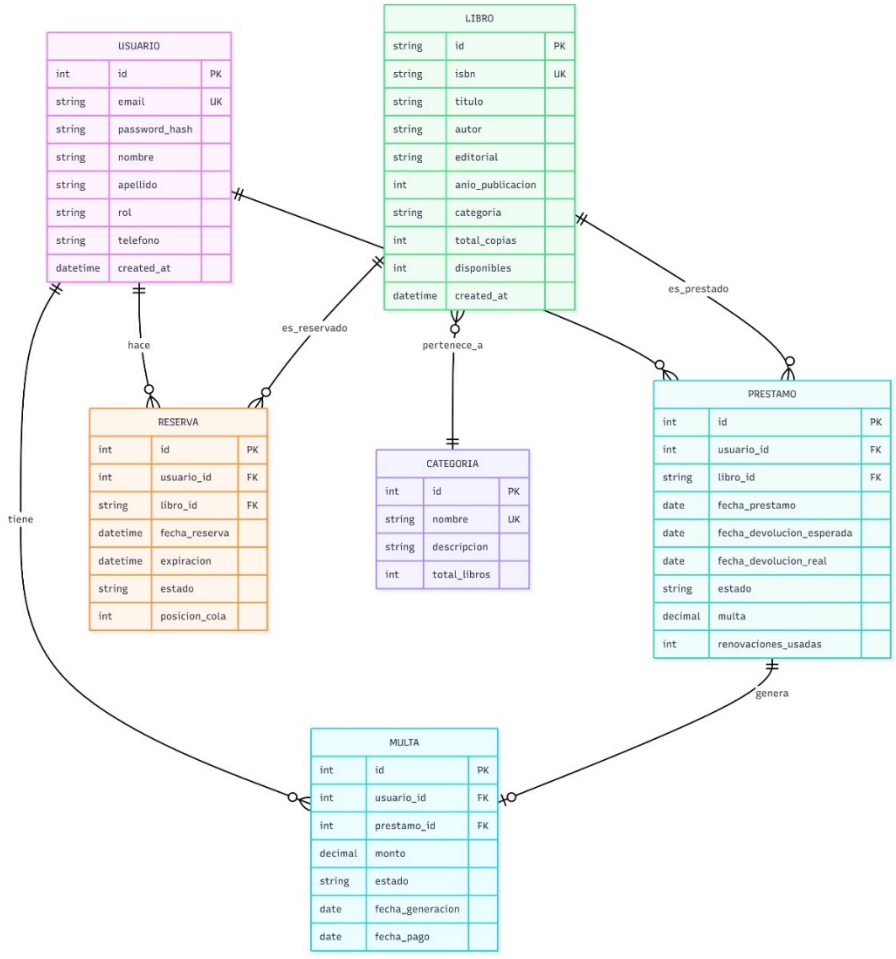


Fuente: Jefferson salcedo, Hector Betancourth

Diagrama de bases de datos.

Se presenta el diagrama entidad relación del sistema de biblioteca universitaria mostrando sus principales entidades y relaciones se utiliza PostgreSQL para datos relacionales, MongoDB para el catálogo de libros y Redis para gestión temporal de reservas (ver figura 4).

Figura 4 Diagrama de bases de datos



Fuente: Jefferson Salcedo, Hector Betancourth

Incremento 1: Autenticación

Para el desarrollo del sistema el primer incremento corresponde al microservicio de autentificación, se usan herramientas como FastAPI, PostgreSQL, Bcrypt y JWT lo que permite que solo los usuarios registrados logren ingresar de manera segura al sistema.

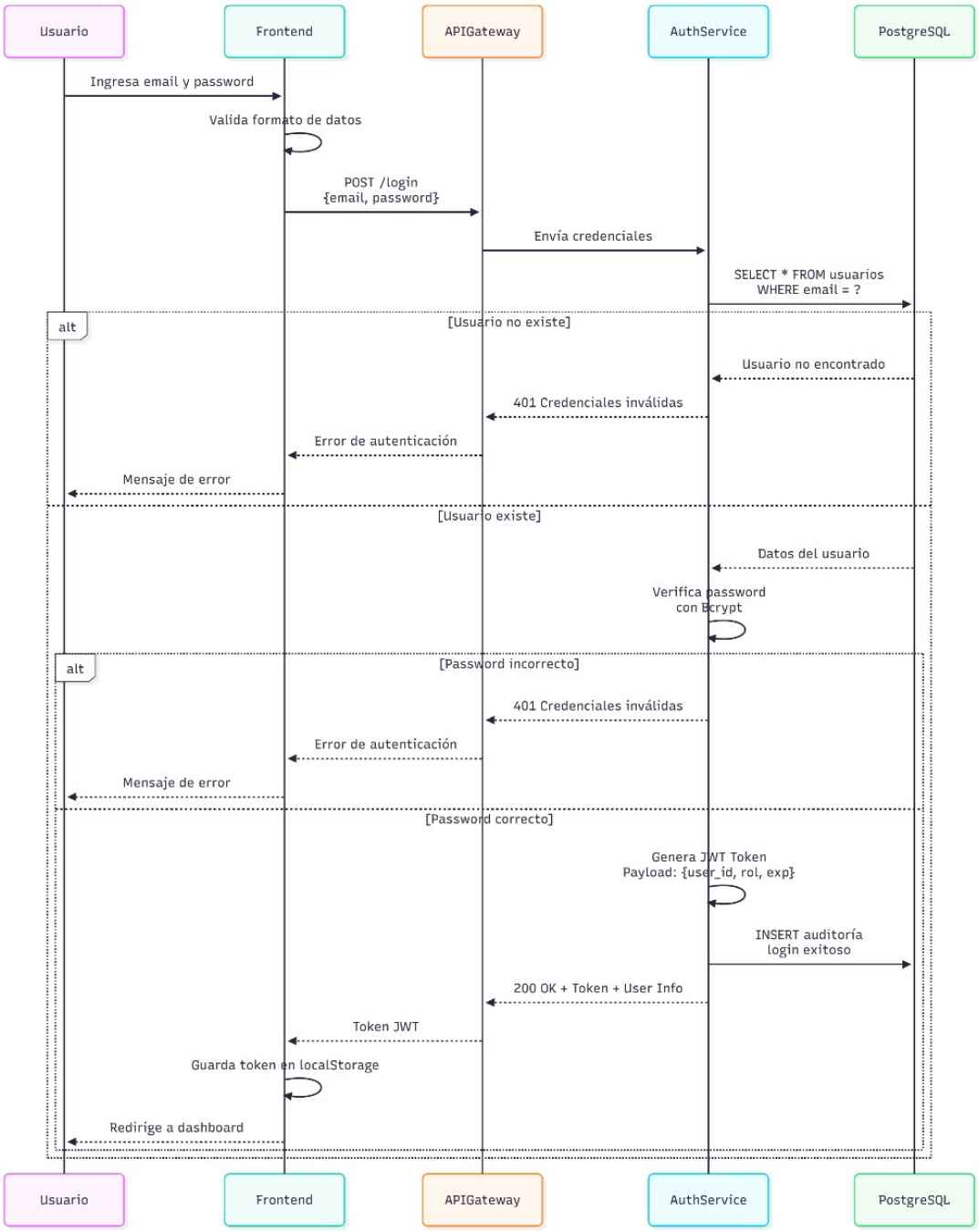
Desarrollos Implementados:

- Registro de usuarios
- Inicio de sesión
- Generador de JWT (JSON Web Token)
- Validación de credenciales
- Protección de rutas

Diagrama de autenticación:

Se presenta el programa de autenticación mediante JWT el usuario ingresa sus credenciales las cuales son verificadas con la base de datos PostgreSQL. Si las credenciales son válidas se genera un token JWT que contiene la información del usuario y sus permisos. (ver figura 5).

Figura 5 Diagrama de autenticación.



Fuente: Jefferson Salcedo, Hector Betancourth

Esta parte de código corresponde a un endpoint que responde a solicitudes POST asignadas a la URL /login utilizando el @app.post para definir el método y la ruta. Se realiza una validación de usuario y contraseña validando si existe el usuario en la base de datos.

```
@app.post("/login", response_model=Token)
async def login(login_data: LoginRequest, db: Session = Depends(get_db)):
    try:
        # Buscar usuario en la base de datos
        user_db = get_user_by_username(db, login_data.username)

        if not user_db:
            raise HTTPException(status_code=401, detail="Credenciales inválidas")

        # Verificar contraseña con bcrypt
        if not verify_password(login_data.password, user_db.password_hash):
            raise HTTPException(status_code=401, detail="Credenciales inválidas")

        # Crear token
        access_token = create_access_token(user_db.username, user_db.role)

        return {
            "access_token": access_token,
            "token_type": "bearer",
            "user": {
                "id": user_db.id,
                "username": user_db.username,
                "email": user_db.email,
                "role": user_db.role,
                "full_name": user_db.full_name
            }
        }

    except HTTPException:
        raise
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error interno: {str(e)}")
```

Incremento 2: Microservicio de catálogos

Se realiza el desarrollo mediante herramienta FastAPI con Python, del microservicio encargado de gestionar todo el contenido del catálogo. En este punto

interactúan el sistema con la base de datos MongoDB para almacenar y consultar información de libros autores y categorías. (ver figura 6).

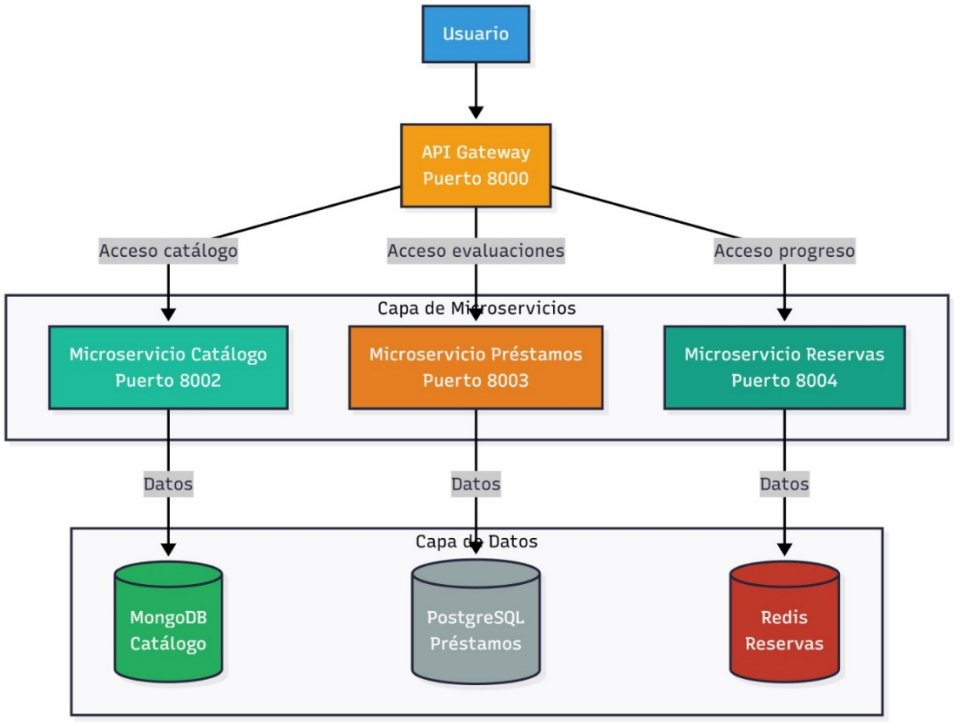
Desarrollos Implementados:

- Microservicio de libros.
- Microservicio de autores.
- Microservicio de categorías.

Diagrama de catálogos.

El diagrama representa la arquitectura de los microservicios, en donde el API Gateway es el punto central de las solicitudes dirigiéndolas a los servicios de catálogo, préstamos y reservas de manera independiente.

Figura 6 Diagrama de catálogos



Fuente: Jefferson Salcedo, Hector Betancourth.

El siguiente código de microservicio permite listar libros por medio del método GET, a través de la función **listar_libros** que acepta parámetros opcionales de búsqueda, validando el Id del libro y consultando al información en MongoDB

```python
@app.get("/libros", response_model=List[LibroResponse])
async def listar_libros(
    categoria: Optional[str] = Query(None, description="Filtrar por categoría"),
    autor: Optional[str] = Query(None, description="Filtrar por autor"),
    search: Optional[str] = Query(None, description="Buscar en título")
):
    # Obtiene la colección de libros de MongoDB
    collection = get_libros_collection()
    query = {}

    # Construye el filtro de búsqueda según los parámetros
    if categoria:
        query["categoria"] = categoria
    if autor:
        query["autor"] = autor
    if search:
```

```
        query["titulo"] = {"$regex": search, "$options": "i"}

    # Ejecuta la consulta en MongoDB
    libros = list(collection.find(query))

    # Convierte _id de MongoDB a id numérico para la respuesta
    for libro in libros:
        libro["id"] = libro["_id"]
        del libro["_id"]

    return libros
```

Incremento 3: Microservicio de prestamos

Se realiza el desarrollo mediante herramienta FastAPI con Python, del microservicio encargado de gestionar los préstamos de libros y reservas de usuarios. En esta parte de implementas las funcionalidades de creación de préstamos como devolución de libros y gestión de reservas.

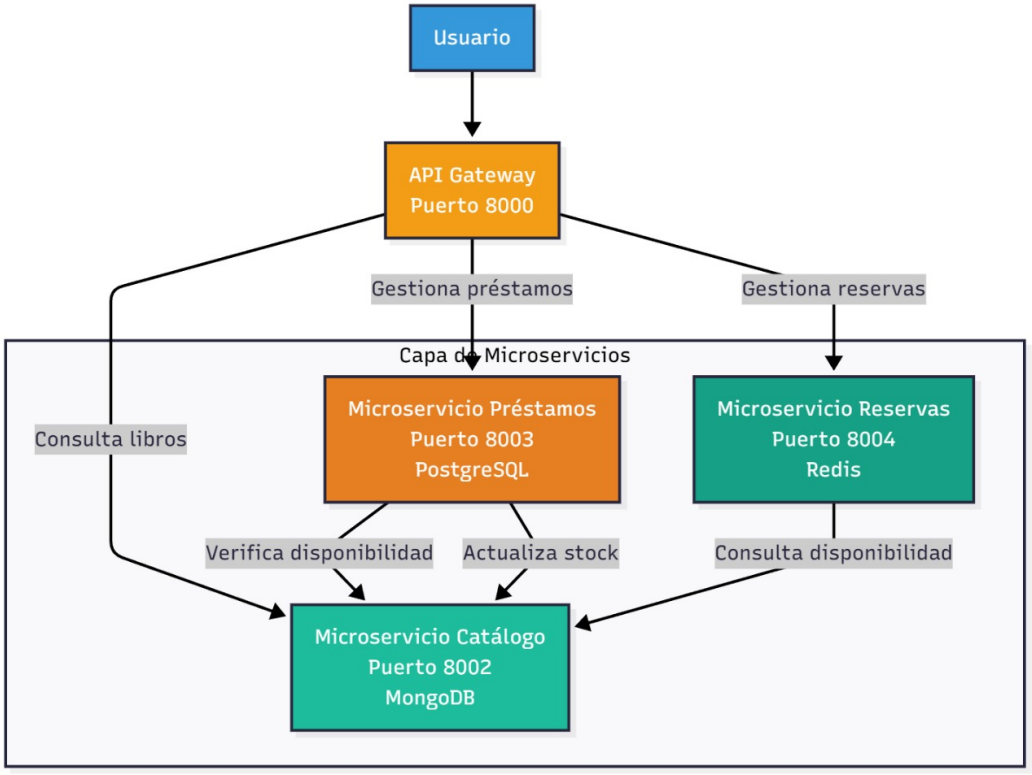
Desarrollos Implementados:

- ❖ Microservicio de préstamos.
- ❖ Microservicio de reservas.
- ❖ Cálculo de multas.

Diagrama de préstamos.

El microservicio de préstamos se comunica con el microservicio de catálogos para verificar disponibilidad y actualizar el stock de libros (ver figura 7).

Figura 7 Diagrama de prestamos



Fuente: Jefferson Salcedo, Hector Betancourth

El siguiente código de microservicios de préstamos permite crear un nuevo préstamo mediante el método POST, a través de la función **crear_prestamo** que recibe los datos del usuario y libro, calculando automáticamente la fecha de devolución (15 días por defecto) y registrando el préstamo en PostgreSQL con estado “activo”.

```
@app.post("/prestamos", response_model=PrestamoResponse)
async def crear_prestamo(prestamo_data: PrestamoCreate, db: Session =
Depends(get_db)):
    try:
        # Obtiene la fecha actual como fecha de préstamo
        fecha_prestamo = datetime.utcnow()

        # Calcula la fecha de devolución esperada
        fecha_devolucion = fecha_prestamo +
timedelta(days=prestamo_data.dias_prestamo)
```

```
        # Crea el objeto Prestamo con los datos
        prestamo = Prestamo(
            usuario_id=prestamo_data.usuario_id,
            libro_id=prestamo_data.libro_id,
            fecha_prestamo=fecha_prestamo,
            fecha_devolucion_esperada=fecha_devolucion,
            estado="activo"
        )

        # Guarda el préstamo en la base de datos
        db.add(prestamo)
        db.commit()
        db.refresh(prestamo)

        return prestamo

    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error creando préstamo:
{str(e)}")
```

Incremento 4: Microservicio de reservas

Se realiza el desarrollo mediante herramienta FastAPI con Python, del microservicio encargado de gestionar la reserva de libros y el sistema de noticiones para usuarios. En esta parte se implementa un sistema de reservas, verificación de reservas duplicadas y notificaciones que alertan a los usuarios sobre el estado de sus reservas, el microservicio utiliza MongoDB.

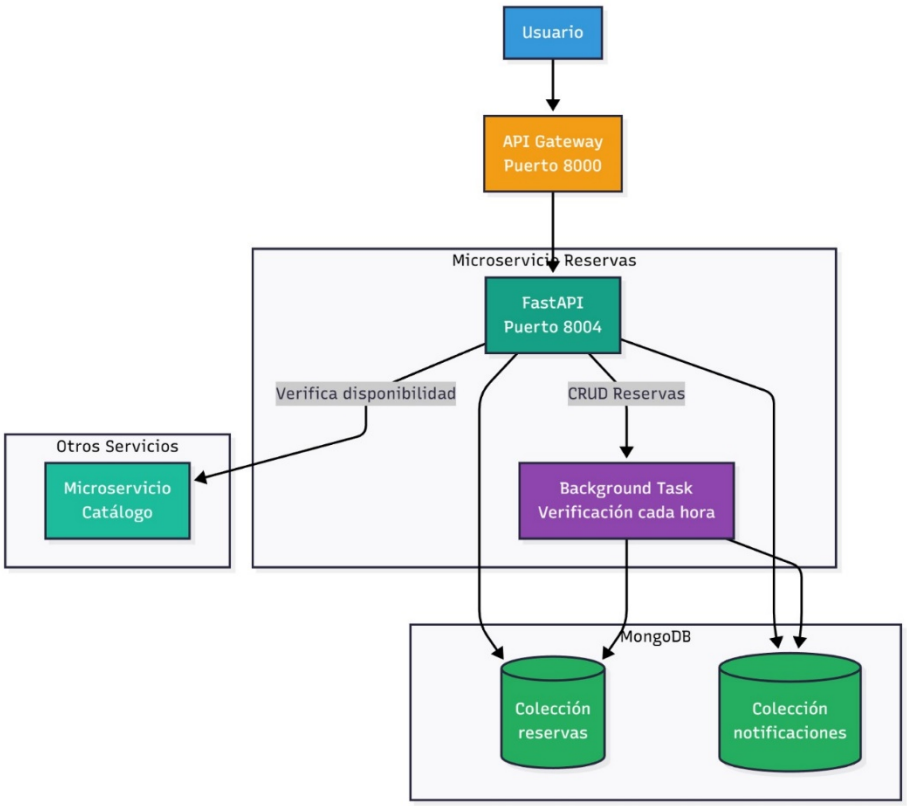
Desarrollos Implementados:

- ❖ Microservicio de reservas.
- ❖ Sistema de notificaciones.
- ❖ Verificación automática de vencimientos.

Diagrama de reservas.

El microservicio de reservas se comunica con el microservicio de catalogo para verificar disponibilidad de libros, y con MongoDB para almacenar reservas y notificaciones (ver figura 8).

Figura 8 Diagrama de reservas



Fuente: Jefferson Salcedo, Hector Betancourth

El siguiente código de microservicios de reservas permite crear una nueva reserva mediante el método POST, a través de la función **crear_reserva** que recibe los datos del usuario y libro, verifica que no exista una reserva activa duplicada, calcula

automáticamente la fecha de vencimiento y registra la reserva en MongoDB generando una notificación de confirmación para el usuario.

```
@app.post("/reservas", response_model=ReservaResponse)
async def crear_reserva(reserva_data: ReservaCreate, background_tasks:
BackgroundTasks):
    db = get_database()

    try:
        print(f"📝 Creando reserva para usuario {reserva_data.usuario_id},
libro {reserva_data.libro_id}")

        # Verificar si ya existe una reserva activa para el mismo libro y
usuario
        if verificar_reserva_activa(db, reserva_data.usuario_id,
reserva_data.libro_id):
            raise HTTPException(
                status_code=400,
                detail="Ya tienes una reserva activa para este libro"
            )

        # Calcular fecha de vencimiento
        fecha_vencimiento =
calcular_fecha_vencimiento(reserva_data.dias_reserva)

        # Crear reserva
        reserva_doc = ReservaDocument(
            usuario_id=reserva_data.usuario_id,
            libro_id=reserva_data.libro_id,
            fecha_vencimiento=fecha_vencimiento
        )

        # Convertir a dict y agregar campo notificado
        reserva_dict = reserva_doc.to_dict()
        reserva_dict["notificado"] = False

        result = db.reservas.insert_one(reserva_dict)

        # Obtener la reserva creada
        reserva_creada = db.reservas.find_one({"_id": result.inserted_id})

        # Crear notificación de reserva exitosa
```

```
        crear_notificacion(
            db,
            reserva_data.usuario_id,
            TipoNotificacion.RECORDATORIO,
            f"Reserva creada exitosamente para el libro ID
{reserva_data.libro_id}. Vence el
{fecha_vencimiento.strftime('%d/%m/%Y')}",
            str(result.inserted_id)
        )

        print(f"✅ Reserva creada exitosamente: {result.inserted_id}")

        return {
            "id": str(reserva_creada["_id"]),
            "usuario_id": reserva_creada["usuario_id"],
            "libro_id": reserva_creada["libro_id"],
            "fecha_reserva": reserva_creada["fecha_reserva"],
            "fecha_vencimiento": reserva_creada["fecha_vencimiento"],
            "estado": reserva_creada["estado"],
            "notificado": reserva_creada["notificado"]
        }

    except HTTPException:
        raise
    except Exception as e:
        print(f"❌ Error creando reserva: {str(e)}")
        raise HTTPException(status_code=500, detail=f"Error creando
reserva: {str(e)}")
```

Incremento 5: API Gateway

Este componente centraliza el acceso a los microservicios de autenticación, catalogo, préstamos y reservas. Gestiona errores de conexión y proporciona un Endpoint de health check que verifica todos los microservicios del sistema.

Desarrollos Implementados:

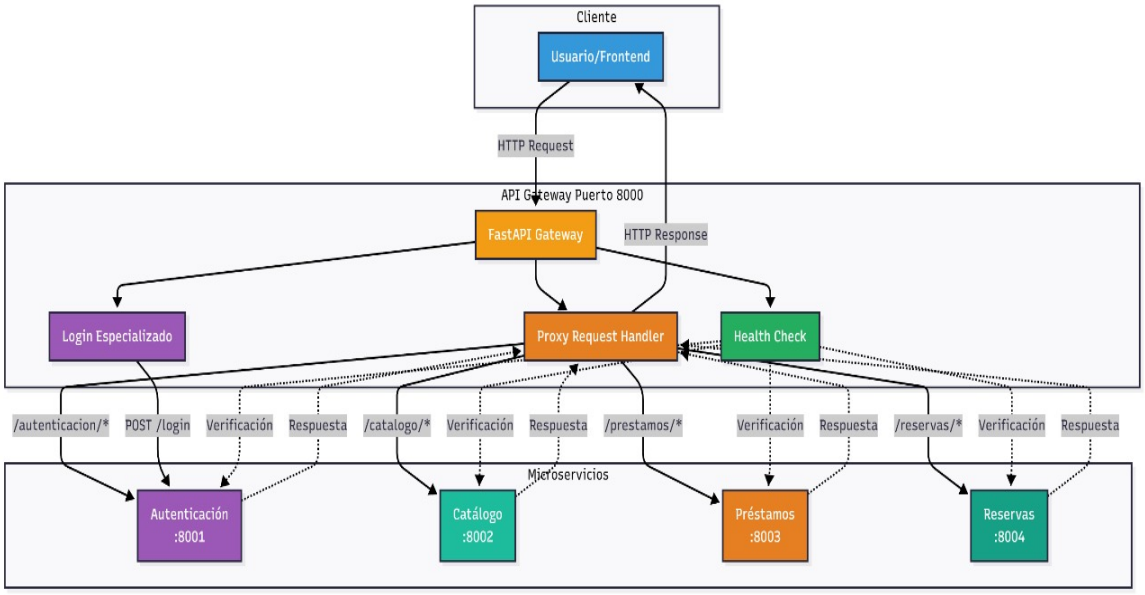
- ❖ API Gateway.

- ❖ Gestión de comunicación.
- ❖ Manejo robusto de errores.
- ❖ Health check
- ❖ Endpoint de login

Diagrama de API Gateway.

El Gateway recibe peticiones del usuario o Frontend, valida el servicio solicitado y enruta la petición al microservicio que le corresponde. Este patrón agrupa la comunicación y abrevia el manejo de errores (ver figura 9).

Figura 9 Diagrama de API Gateway



Fuente: Jefferson Salcedo, Hector Betancourth

El siguiente código implementa el núcleo del API Gateway mediante la función **proxy_request** que recibe todas las peticiones dirigidas a cualquier microservicio, esta función extraer el nombre del servicio y la ruta de la URL.

```
@app.api_route("/{service_name}/{path:path}", methods=["GET", "POST", "PUT",
"DELETE"])
async def proxy_request(service_name: str, path: str, request: Request):
    # Validar que el servicio esté registrado
    if service_name not in SERVICES:
        raise HTTPException(status_code=404, detail=f"Servicio {service_name} no
encontrado")

    # Construir URL destino
    base_url = SERVICES[service_name]
    target_url = f"{base_url}/{path}"

    # Preservar headers (excepto host y content-length)
    headers = {key: value for key, value in request.headers.items()
               if key.lower() not in ["host", "content-length"]}

    # Extraer body si es POST, PUT o PATCH
    body = None
    if request.method in ["POST", "PUT", "PATCH"]:
        body = await request.body()
```

Incremento 6: Frontend

Este componente de Frontend funciona como capa de presentación que usa los servicios de API Gateway, montando vistas para login, catálogo de libros, gestión de préstamos y reservas.

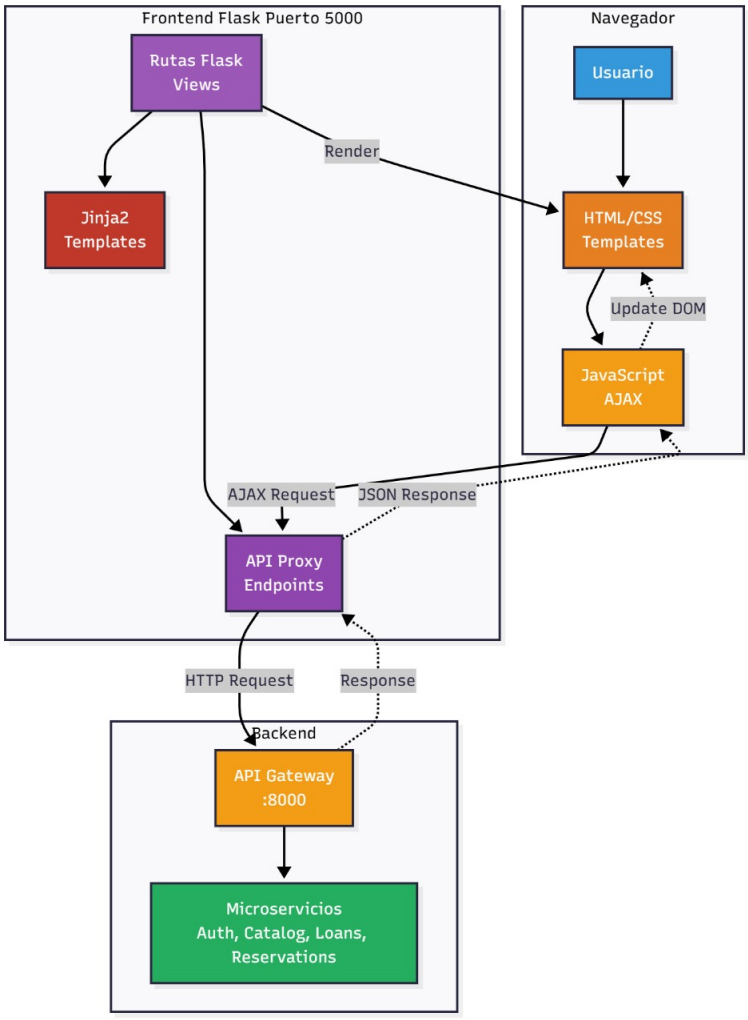
Desarrollos Implementados:

- ❖ Sistema de vistas Flask.
- ❖ API Proxy al Gateway.
- ❖ Manejo de comunicación.
- ❖ Gestión de sesiones.

Diagrama de Frontend.

El diagrama representa la arquitectura del Frontend, el usuario interactúa con la interface HTML, las peticiones se envían al Backend que las reenvía al API Gateway y las respuestas se encaminan de vuelta al navegador para actualizar la interface dinámicamente (ver figura 10), (ver figura 11).

Figura 10 Diagrama de Frontend.



Fuente: Jefferson Salcedo, Hector Betancourth

El siguiente código implementa el Endpoint del login del Frontend que actúa como Proxy entre navegador y el API Gateway, la función **login** recibe las credenciales del usuario desde una petición. Las reenvía al microservicio de autentificación a través del API Gateway y retorna la reserva.

```
@app.route('/api/login', methods=['POST'])
def login():
```

```
    try:
        # Obtener datos JSON del request del navegador
        data = request.get_json()

        # Realizar petición POST al API Gateway
        response = requests.post(f'{API_BASE}/autenticacion/login', json=data)

        # Retornar respuesta del backend al navegador
        return jsonify(response.json()), response.status_code

    except requests.RequestException as e:
        # Manejar errores de conexión
        return jsonify({'error': 'Error de conexión con el servidor'}), 503
```

La siguiente captura de pantalla muestra la vista de inicio de sesión en la dashboard del usuario estudiante, quien obtienen los permisos de ingreso mediante la autenticación.

Figura 11 Dashboard.

Fuente: Jefferson Salcedo, Hector Betancourth

Incremento 7: Despliegue de Docker

Se realiza el despliegue mediante Docker y Docker Compose, implementando una arquitectura de contenedores que permite iniciar los componentes de manera aislada y escalable

Desarrollos Implementados:

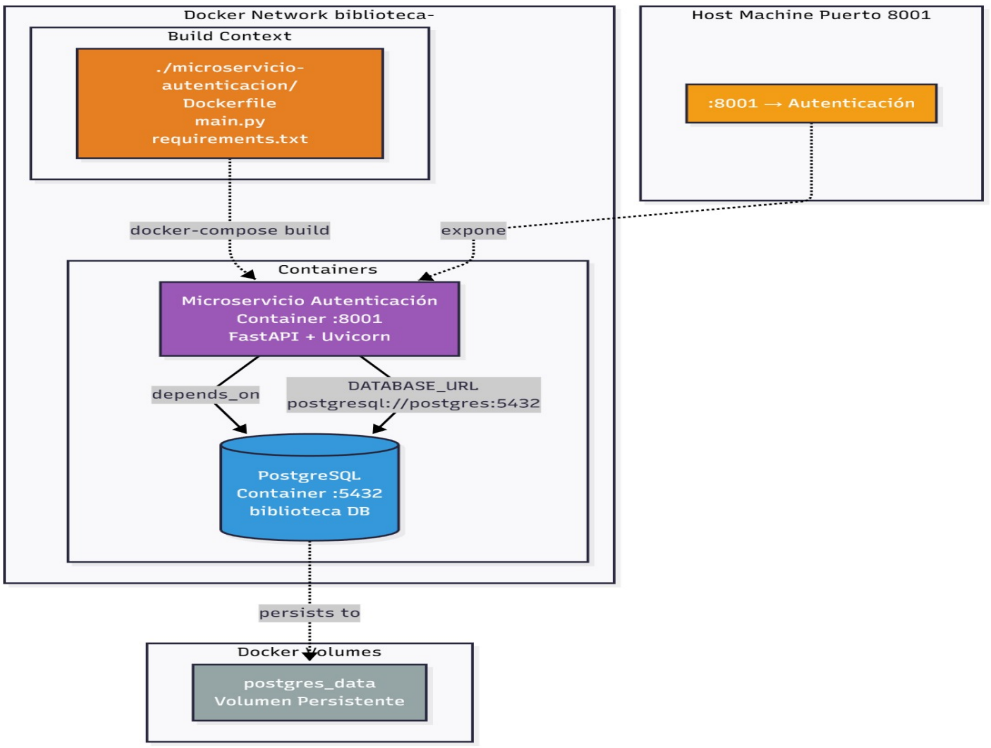
- Contenedores de microservicios.
- Gestión de variables de entorno.
- Orquestación con dependencias.

Diagrama de arquitectura de contenedores Docker

El siguiente diagrama muestra la arquitectura de contenedores Docker del sistema (ver figura 12, 13 y 14).

Se conecta a la red privada **biblioteca_network.** El microservicio de autentificación depende de PostgreSQL, recibe la URL de conexión como variable de entorno y expone el puerto 8001 para la comunicación con el API Gateway.

Figura 12 Diagrama de despliegue Docker desktop



Fuente: Jefferson Salcedo, Hector Betancourth

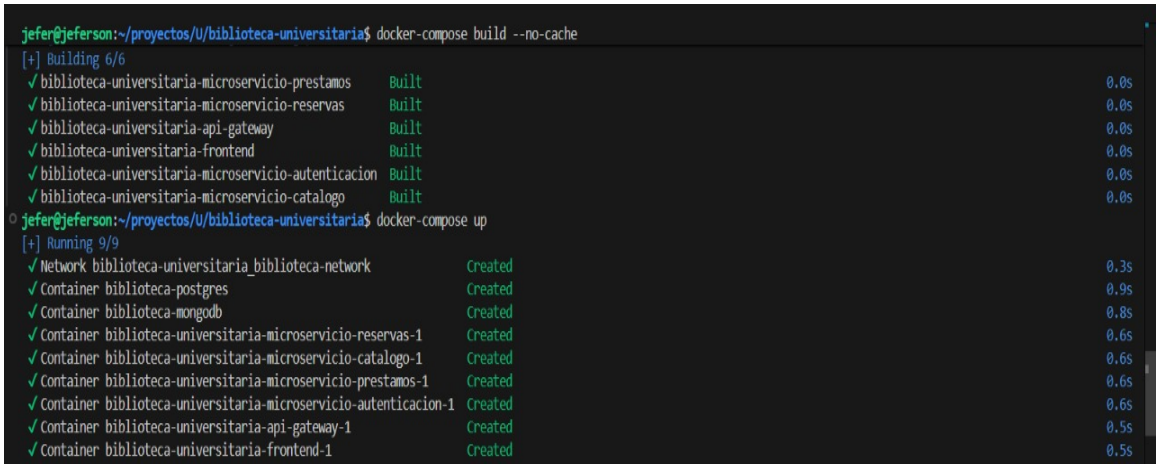
El siguiente código define el servicio **microservicio-autentificación** en Docker Compose ejecutando la directiva **build,** para garantizar que la base de datos inicie primero y conectando el contenedor a la red privada **biblioteca-network** para comunicación con otros servicios

```
# MICROSERVICIO AUTENTICACIÓN ACTUALIZADO
microservicio-autenticacion:
  # Construir imagen desde Dockerfile local
  build: ./microservicio-autenticacion

  # Mapeo de puertos: host:container
  ports:
```

```
    - "8001:8001"

  # Variables de entorno para configuración
  environment:
    - DATABASE_URL=postgresql://admin:admin123@postgres:5432/biblioteca

  # Dependencias de inicio
  depends_on:
    - postgres

  # Red privada compartida
  networks:
    - biblioteca-network
```

La siguiente captura de pantalla muestra la ejecución exitosa de Docker compose para construir y desplegar sistema completo, se observan 2 comandos principales **docker-compose build –no-cache** que hace la construcción de imágenes y **docker-compose up** que inicia los contenedores, por ultimo el sistema funciona con todos los servicios interconectados en la red **biblioteca network.**

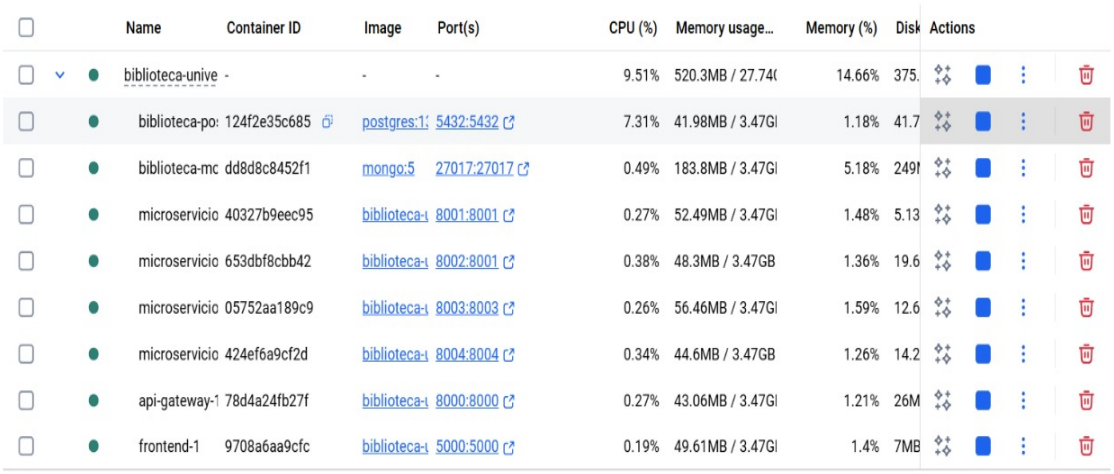
Figura 13 Creación de contenedores



Fuente: Jefferson Salcedo, Hector Betancourth

En la siguiente captura se muestra la interfaz grafica de **docker desktop** con todos los contenedores del sistema ejecutándose correctamente

Figura 13 Contenedores en App Docker



	Name	Container ID	Image	Port(s)	CPU (%)	Memory usage...	Memory (%)	Disk	Actions
	biblioteca-unive	-	-	-	9.51%	520.3MB / 27.74(	14.66%	375.	
	biblioteca-po:	124f2e35c685	postgres:1:	5432:5432	7.31%	41.98MB / 3.47GI	1.18%	41.7	
	biblioteca-mc	dd8d8c8452f1	mongo:5	27017:27017	0.49%	183.8MB / 3.47GI	5.18%	249I	
	microservicio	40327b9eec95	biblioteca-i	8001:8001	0.27%	52.49MB / 3.47GI	1.48%	5.13	
	microservicio	653dbf8cbb42	biblioteca-i	8002:8001	0.38%	48.3MB / 3.47GB	1.36%	19.6	
	microservicio	05752aa189c9	biblioteca-i	8003:8003	0.26%	56.46MB / 3.47GI	1.59%	12.6	
	microservicio	424ef6a9cf2d	biblioteca-i	8004:8004	0.34%	44.6MB / 3.47GB	1.26%	14.2	
	api-gateway-1	78d4a24fb27f	biblioteca-i	8000:8000	0.27%	43.06MB / 3.47GI	1.21%	26M	
	frontend-1	9708a6aa9cfc	biblioteca-i	5000:5000	0.19%	49.61MB / 3.47GI	1.4%	7MB	

Fuente: Jefferson Salcedo, Héctor Betancourth

Documentación.

A continuación, se muestra la documentación de código del proyecto elaborado con la herramienta MKDocs, llamando a los sitios donde están ubicados para que puedan ser visualizados a través del puerto local (ver figura 14).

```
nav:
  - Inicio: index.md
  - Arquitectura:
    - Visión General: architecture/overview.md
  - Microservicios:
    - Autenticación: microservices/authentication.md
    - Catalogo: microservices/catalog.md
    - Préstamos: microservices/loans.md
    - Reservas: microservices/reservations.md
  - Guías:
    - Instalación: guides/installation.md
```

La siguiente captura muestra la pantalla inicial, sus respectivas pestañas donde están organizadas por medio de pestañas con su documentación correspondiente

Figure 14 Documentación de código

Fuente: Jefferson Salcedo, Héctor Betancourth

Conclusiones

La integración de las herramientas Docker, FastAPI Y Python en el desarrollo de este sistema bibliotecario garantiza una plataforma segura moderna y sobre todo eficiente ya que Docker permite un despliegue estable de todos los servicios, mientras que FastAPI aporta mas rapidez y un diseño mas limpio en su código

El uso de una arquitectura basada en microservicios aporta mas flexibilidad y una mejor mantenibilidad a la hora de cuidar el sistema ya que cada módulo como autentificación, prestamos o reservas pueden ser modificados sin afectar todo el código

Esta propuesta contribuye mucho en el campo de la transformación digital de las bibliotecas en el sector educativo proporcionando así un enfoque tecnológica y metodológica que puede servir para futuros proyectos que busquen mejorar la gestión sobre esta información y optimizar procesos educativos e institucionales mediante soluciones escalables y modernas

Referencias

Battaglia, N., García, A. N., & Congiusti, A. (2024). XXX Congreso Argentino de Ciencias de la Computación (CACIC) . *XXX Congreso Argentino de Ciencias de la Computación (CACIC) (La Plata, 7 al 11 de octubre de 2024)*, (págs. 1506-1510).

Buñay Guisñan, P. A. (2024). *Microservicios para el módulo de proyectos del sistema de gestión de investigación de la Unach.* Riobamba, Universidad Nacional de Chimborazo.

Contreras, D. A. (2018). *Tecnología, investigación y academia TIA.*

Coral Bastidas, P. M. (2003). *Sistema de información para la Biblioteca y desarrollo de la página web de la Universidad de Nariño.* Ipiales-Nariño.

Ericka Solano Fernández, D. P. (2020). *El modelo iterativo e incremental para el desarrollo de la aplicación de realidad aumentada Amón_RA.*

Franco Noreña, Y. (2022). *Revisión sistemática de la literatura sobre Tecnologías Docker.*

Fuentes Romero, J. J. (2004). *La colección de materiales en las bibliotecas.*

Jefferson Alexis Paredes Plaza, D. P. (2024). *Análisis comparativo de metodologías de gestión de proyectos en construcción: Cascada, Ágil y Lean.*

José Navarro, R. M. (2016). *Desarrollo y desempeño en equipos de proyecto: validez incremental de la escala de desarrollo grupal.*

Marcos E. Aguirre Zurita, V. H. (2002). *· USO DE LA METODOLOGIA INCREMENTAL EN ELDESARROLLO DE UN SISTEMA INTEGRAL DE EVENTOS* . Lima-Peru.

Meneses-Tello, F., & Licea de Arenas, J. (2005). *El problema ideológico de la selección-eliminación- destrucción de libros y bibliotecas.* La Habana, Cuba.

Moreno Bernal, S. D. (2022). *Modelo de evolución arquitectónica de monolito a microservicios para el sistema de información ISys de la Dirección Nacional de Admisiones de la Universidad Nacional de Colombia.*

Palacio, V. F. (2022). *Desarrollo de Gateway de Seguridad en Python con FastAPI.*

Romero, J. J. (2004). *La colección de materiales en las bibliotecas problemas actuales.*