ADMINISTRACIÓN
Y NEGOCIOS

Inacap

Manual Técnico Profesional Nuam

Nombres: Valeska Aguirre y Nicolás Espejo
Carrera: Ingeniería en informática
Asignatura: Programación Back End TI3041
Profesor: Javier Arturo García Barrientos
Fecha: 12/12/25

Introducción y Alcance Estratégico

Visión general del proyecto

El presente Manual Técnico Integral (Versión 3.0) constituye la referencia autoritativa para la arquitectura, despliegue, operación y mantenimiento de la plataforma tecnológica del Proyecto NUAM. Este documento ha sido diseñado para servir a ingenieros de software, arquitectos de sistemas y especialistas en operaciones (DevOps) que requieren una comprensión profunda de las decisiones de diseño que sustentan el sistema. La plataforma NUAM representa una evolución significativa en el procesamiento de transacciones financieras, migrando de paradigmas monolíticos heredados hacia una arquitectura de microservicios distribuida, resiliente y altamente escalable.[1]

La arquitectura se fundamenta en la orquestación de contenedores Docker, utilizando Django como el núcleo de la lógica de negocio bajo el patrón Modelo-Vista-Template (MVT), y Apache Kafka como la columna vertebral para la transmisión asíncrona de eventos de alta velocidad. Esta combinación tecnológica no es accidental; responde a la necesidad crítica de procesar volúmenes masivos de datos financieros con garantías de integridad transaccional (ACID) y seguridad de transporte (SSL/TLS) desde el entorno de desarrollo hasta la producción.[3]

Objetivos

Esta actualización del manual tiene como objetivo cerrar la brecha entre la implementación técnica y la excelencia operativa. Se incorporan directrices exhaustivas sobre:

1. **Seguridad Avanzada:** Implementación de SSL en desarrollo mediante `django-sslserver` y políticas HSTS.
2. **Integridad de Datos:** Uso de transacciones atómicas y optimización de índices en bases de datos relacionales.
3. **Orquestación Robusta:** Protocolos de instalación automatizada y recuperación ante fallos en entornos Windows (WSL2) y Linux.
4. **Justificación Arquitectónica:** Análisis comparativo detallado de tecnologías clave (Kafka vs. Pulsar).

Arquitectura del Sistema y Tecnologías Core

La arquitectura de NUAM se adhiere a los principios de los "Twelve-Factor App", priorizando la paridad entre desarrollo y producción, la gestión de configuraciones mediante variables de entorno y el tratamiento de servicios de respaldo (backing services) como recursos adjuntos.

Paradigma Django MVT (Modelo-Vista-Template)

Arquitectura de Interfaz Híbrida: Aunque el sistema opera con una lógica de microservicios en el backend, la interfaz de usuario se entrega mediante una arquitectura de **SPA (Single Page Application) embebida**. Django sirve la estructura base HTML y los scripts JS, los cuales consumen posteriormente la API REST (/api/) mediante fetch. Esto elimina la necesidad de un servidor Node.js adicional (puerto 3000), simplificando el despliegue en Docker a un solo contenedor de aplicación que maneja tanto la lógica de negocio como la entrega de la interfaz.

El Modelo (La Verdad de los Datos)

La capa "Modelo" actúa como la fuente única de verdad. En NUAM, los modelos no son simples mapeos a tablas de base de datos; son clases Python que encapsulan la lógica de validación, las relaciones y los comportamientos de los datos financieros. El ORM (Object-Relational Mapping) de Django abstrae la complejidad del SQL, permitiendo operaciones seguras y portables.

- **Seguridad Intrínseca:** El uso del ORM previene inyecciones SQL al escapar automáticamente los parámetros de consulta, una característica vital para proteger activos financieros.[5]
- **Abstracción de Base de Datos:** Permite cambiar el motor de base de datos subyacente (de SQLite en pruebas a MySQL/PostgreSQL en producción) con cambios mínimos en el código.[6]

La Vista (Orquestación Lógica)

En la arquitectura de NUAM, las "Vistas" gestionan el flujo de información. Reciben solicitudes HTTP, aplican lógica de negocio (como validación de transacciones o autenticación) y devuelven respuestas.

- **Implementación RESTful Estricta:** Se adoptó una arquitectura RESTful Nivel 2 (Richardson Maturity Model) mediante el uso de `DefaultRouter` y `ModelViewSets` en `api/urls.py` y `api/views.py`. Esta decisión arquitectónica elimina la definición manual de rutas (URL hardcoding), delegando en el framework la generación automática de los endpoints estándar:
- `GET /api/recurso/`: Listado y paginación.
- `POST /api/recurso/`: Creación de entidades.
- `PATCH /api/recurso/{id}/`: Modificación parcial.
- `DELETE /api/recurso/{id}/`: Eliminación lógica o física.

- **Decoradores de Seguridad:** Las vistas son el punto de entrada para aplicar decoradores de seguridad como `@permission_classes` y `@authentication_classes`, asegurando que solo los actores autorizados interactúen con el sistema.[8]

El Template (Interfaz Administrativa y Documentación)

Aunque el frontend de usuario final es una SPA (Single Page Application), el componente "Template" de Django es crucial para:

- **Panel de Administración:** Proporciona una interfaz gráfica lista para usar (`/admin`) que permite a los administradores gestionar usuarios y revisar registros de auditoría sin interactuar directamente con la base de datos.[1]
- **Documentación Viva:** Generación de interfaces de documentación de API como Swagger y Redoc (`/docs`), que permiten a los desarrolladores probar endpoints en tiempo real.

Columna vertebral de mensajería: Kafka vs. Pulsar

Una de las decisiones arquitectónicas más trascendentales fue la elección del broker de mensajería. Se evaluaron Apache Kafka y Apache Pulsar bajo criterios estrictos de rendimiento financiero y mantenibilidad operativa.

Análisis comparativo técnico:

Criterio	Apache Kafka	Apache Pulsar	Veredicto para NUAM
Arquitectura de logs	Monolítica (Log distribuido simple). Escritura secuencial en disco, optimizada para throughput bruto.	Tiered Storage (Separación de Cómputo y Almacenamiento). Brokers sin estado y bookies para persistencia.	**Kafka.** Para el perfil de tráfico de NUAM, la simplicidad del log secuencial de Kafka ofrece un rendimiento predecible y superior en escrituras masivas.[10]
Madurez y ecosistema	**Muy alta.** Estándar de la industria (Fortune 500). Ecosistema masivo de conectores (Kafka Connect) y herramientas.	**Alta.** Crecimiento rápido, pero comunidad más pequeña y menos recursos de terceros disponibles.	**Kafka.** La disponibilidad de documentación, bibliotecas cliente maduras (confluent-kafka) y soporte comunitario reduce el riesgo del proyecto.[10]
Latencia vs. Throughput	Latencia ultrabaja (< 5 ms p99) bajo carga alta. Throughput masivo (millones de eventos/seg).	Mejor latencia de cola en escenarios de altísima concurrencia, pero throughput pico inferior a Kafka.	**Kafka.** Priorizamos el throughput para manejar picos de transacciones de mercado sobre las características de multitenancy de Pulsar.[10]
Complejidad operativa	Centralizada (Zookeeper/KRaft). Gestión de particiones conocida y manejable.	Alta complejidad. Requiere gestionar Zookeeper + BookKeeper + Brokers + Proxies.	**Kafka.** Pulsar introduce una carga operativa excesiva (más componentes móviles) que no justifica sus beneficios para nuestro caso de uso actual.[12]

Conclusión estratégica: Kafka fue seleccionado por su **simplicidad operativa relativa**, su **madurez inigualable** y su capacidad probada para manejar el alto throughput requerido por los sistemas financieros modernos. Su integración con herramientas de big data (Spark, Flink) asegura la viabilidad futura de la plataforma para análisis de datos.[1]

Implementación Técnica del Productor (Producer)

Para cumplir con el rol de productor de eventos asignado a la arquitectura, se desarrolló un módulo dedicado de desacoplamiento:

- **Módulo:** api/producers.py
- **Librería Core:** confluent-kafka
- **Patrón de Diseño:** Singleton para la conexión persistente.

La integración se realiza a nivel de Vistas (views.py). Específicamente, en el CalificacionViewSet, se sobrescribe el método perform_create. Esto garantiza que el evento se dispare hacia Kafka **solo** cuando la transacción en la base de datos (MySQL) ha sido validada, asegurando la consistencia entre el almacenamiento relacional y el log de eventos distribuidos.

Seguridad e integridad de datos

La seguridad en NUAM se aborda mediante un modelo de "defensa en profundidad", asegurando capas de protección desde el transporte hasta la persistencia.

Cifrado en tránsito y SSL/TLS

La protección de los datos en tránsito es obligatoria. NUAM implementa una estrategia dual para manejar SSL/TLS, diferenciando entre entornos de desarrollo y producción para maximizar la productividad sin sacrificar la seguridad.

Desarrollo: `django-sslserver`

En el entorno local Dockerizado, utilizamos `django-sslserver`. Esto permite a los desarrolladores trabajar con protocolos HTTPS (`https://localhost:8000`) simulando un entorno de producción real.

- **Comando de ejecución:**

```
python manage.py runsslserver 0.0.0.0:8000 --certificate /app/certs/cert.pem --key /app/certs/key.pem
```

- **Propósito:** Habilita la prueba de características que requieren contextos seguros, como cookies con la bandera `Secure`, geolocalización o acceso al portapapeles, y previene errores mixtos de contenido antes del despliegue.[1]
- **Limitación:** Este servidor es de un solo hilo y no está auditado para seguridad en internet pública. Nunca debe usarse en producción.[14]

HTTP Strict Transport Security (HSTS)

Para mitigar ataques de "SSL Stripping" (donde un atacante fuerza una degradación de HTTPS a HTTP), NUAM implementa cabeceras HSTS.

- **Mecanismo:** El servidor envía la cabecera `Strict-Transport-Security` que instruye al navegador a **nunca** cargar el sitio mediante HTTP durante un periodo definido (`max-age`).
- **Configuración crítica:**
    - `SECURE_HSTS_SECONDS`: Define la duración de la política (ej. 31536000 segundos = 1 año).
    - `SECURE_HSTS_INCLUDE_SUBDOMAINS`: Extiende la protección a todos los subdominios, cerrando brechas de seguridad en servicios satélite.
    - `SECURE_HSTS_PRELOAD`: Permite incluir el dominio en las listas de precarga de los navegadores principales (Chrome, Firefox), protegiendo incluso la primera conexión del usuario.[16]

Endurecimiento de Cookies y Sesiones

Además del cifrado en tránsito, se configuró el backend para rechazar cookies no seguras, incluso en entornos de desarrollo local Dockerizados. En `settings.py` se forzaron las siguientes directivas para mitigar el secuestro de sesiones (Session Hijacking):

- `SESSION_COOKIE_SECURE = True`: Obliga al navegador a enviar la cookie de sesión solo si el canal es HTTPS.
- `CSRF_COOKIE_SECURE = True`: Protege el token contra falsificación de peticiones en sitios cruzados bajo canales encriptados.
- **Certificados:** Se implementaron certificados X.509 autofirmados (`cert.pem` y `key.pem`) montados como volúmenes en el contenedor del backend.

Optimización y consistencia de base de datos

La base de datos MySQL 8.0 no es solo un almacén pasivo; se configura activamente para garantizar rendimiento y consistencia.

Indexación estratégica (`db_index`)

Para evitar la degradación del rendimiento O(N) en tablas grandes, se utilizan índices B-Tree.

- **Uso:** Se aplica `db_index=True` en campos utilizados frecuentemente en filtros (`filter()`), ordenamientos (`order_by()`) y claves foráneas.
- **Impacto:** Transforma búsquedas lineales en logarítmicas, reduciendo drásticamente el tiempo de respuesta en consultas de lectura. Sin embargo, se debe equilibrar cuidadosamente, ya que cada índice añade sobrecarga a las operaciones de escritura (`INSERT`/`UPDATE`).[18]

Transacciones atómicas (transaction.atomic)

Para garantizar la integridad financiera, se utiliza el principio de atomicidad (la "A" en ACID).

- **Implementación:** Mediante el decorador `@transaction.atomic` o el gestor de contexto `with transaction.atomic():`.
- **Funcionamiento:** Agrupa una serie de operaciones de base de datos en un bloque indivisible. Si ocurre una excepción en cualquier punto del bloque (ej., falla la red, error de validación), Django

ejecuta automáticamente un `ROLLBACK`, revirtiendo todos los cambios pendientes. Si el bloque termina exitosamente, se hace `COMMIT`. Esto evita estados de datos inconsistentes o "registros huérfanos".[20]

Guía de instalación y despliegue

Los procedimientos de instalación están automatizados para reducir el error humano y garantizar la reproducibilidad del entorno.

Requisitos previos del sistema

Antes de ejecutar los scripts de instalación, el entorno anfitrión debe cumplir con lo siguiente[1]:

Clonar el repositorio de GitHub:

Con el comando: git clone https://github.com/Saebloom/Code_NUAM_Backend.git

Windows:

- Windows 10/11 (build reciente).
- **WSL2:** Activado y actualizado (`wsl --update`). Es crítico para el rendimiento de Docker.
- **Docker Desktop:** Instalado y en ejecución (icono de la ballena verde).
- **Git:** Para clonar el repositorio.
- **Python 3.10+:** Opcional, solo si se ejecutan scripts de utilidad locales fuera de los contenedores.

Linux (Ubuntu 20.04+):

- **Docker Engine** y **Docker Compose V2**.
- **Git**.
- Permisos de superusuario (`sudo`) para gestionar el demonio de Docker.

Instalación automatizada en Windows

El script oficial `install_windows.bat` orquesta todo el proceso. A continuación se detalla su lógica paso a paso[1]:

1. **Validación de motor (motor encendido):**
   a. Comando: `docker info nul 2>&1`
   b. *Lógica:* Verifica si el demonio de Docker responde. Si el código de salida (`%ERRORLEVEL%`) no es 0, el script se detiene y alerta al usuario. Esto previene errores en cascada.
2. **Limpieza y Construcción (Clean Start):**
   a. Comandos: `docker compose down` seguido de `docker compose up -d --build`.
   b. *Lógica:* Elimina contenedores previos para evitar conflictos de estado y reconstruye las imágenes para asegurar que se use el código más reciente.
3. **Espera activa ("wait trick"):**
   a. Comando: `ping -n 16 127.0.0.1 nul`
   b. *Lógica:* Introduce una pausa determinista de ~15 segundos. Esto es crucial porque Docker inicia los contenedores en paralelo; sin esta espera, el backend intentaría conectarse a la base de datos antes de que el socket de MySQL esté listo, causando fallos de conexión.
4. **Migraciones de base de datos:**
   a. Comando: `docker compose exec backend python manage.py migrate`
   b. *Lógica:* Aplica el esquema de datos relacional dentro del contenedor backend ya activo.

5. **Aprovisionamiento de superusuario:**
    a. Comando: `docker compose exec... createsuperuser --noinput`
    b. *Lógica:* Crea el usuario `admin` con credenciales predefinidas. El operador `|| echo...` maneja el caso donde el usuario ya existe, asegurando que el script sea idempotente (puede ejecutarse múltiples veces sin fallar).

Instalación automatizada en Linux

El script `install_linux.sh` sigue una lógica paralela adaptada a entornos Unix[1]:

1. **Permisos:** Requiere `chmod +x install_linux`.
2. **Validación:** Comprueba la existencia del binario `docker`.
3. **Espera:** Utiliza `sleep 15` explícito para la sincronización de la base de datos.
4. **Ejecución:** Levanta el stack y ejecuta comandos de gestión de Django dentro de los contenedores.

Verificación postinstalación

Una vez finalizada la ejecución de los scripts, es imperativo validar la salud del sistema.

Estado de los contenedores

Ejecute el siguiente comando para verificar el estado:

```
docker-compose                                                    ps
```

Criterio de éxito: Todos los servicios (`backend`, `db`, `kafka`, `zookeeper`) deben mostrar un estado de `running` o `healthy`. Si algún servicio está en `restarting` o `exited`, consulte la sección de Errores comunes.

Validación de puertos

Asegúrese de que los puertos críticos estén escuchando y no tengan conflictos:

- `lsof -i :8000` (Backend API y Frontend SPA)
- `lsof -i :3307` (MySQL)
- `lsof -i :9092` (Kafka Broker – Acceso Externo)

Credenciales de acceso

El sistema se despliega con credenciales por defecto para facilitar el desarrollo inicial. **Nota de seguridad:** Estas deben cambiarse inmediatamente en un entorno de producción real modificando el archivo `.env`.

- **Administrador (Django Admin):**
    - URL: https://localhost:8000/admin
    - Usuario: `admin`

- Clave: admin123
- **Swagger (Documentación API):**
  - URL: https://localhost:8000/swagger
  - Permite inspeccionar y probar interactivamente todos los endpoints REST del sistema.
- **Frontend (portal de usuario):**
  - URL: https://localhost:8000

Operación diaria y comandos para desarrolladores

Para simplificar la interacción con la orquestación de Docker, se proporcionan los scripts `compose-helper` (Linux/Mac) y `compose-helper-w.bat` (Windows). Estos scripts actúan como una capa de abstracción sobre comandos complejos de Docker Compose.[1]

Tabla de comandos oficiales

Comando (compose-helper <cmd>)	**Acción subyacente**	**Descripción y uso**
`up`	`docker compose up -d`	Levanta todo el entorno en segundo plano (detached mode). Usar al iniciar la jornada.
`down`	`docker-compose down`	Detiene y elimina contenedores y redes. Usar para liberar recursos o reiniciar desde cero.
`Logs`	`docker compose logs -f backend`	Muestra y sigue (`-f`) los logs del servicio backend. Esencial para *debugging* en tiempo real.
`bash`	`docker compose exec backend bash`	Abre una terminal interactiva dentro del contenedor backend. Útil para ejecutar scripts manuales o inspeccionar archivos.
`MySQL`	`docker compose exec db mysql -u`	Acceso directo a la consola SQL de la base de datos. Permite ejecutar consultas crudas.
`migrate`	`python manage.py migrate`	Ejecuta migraciones pendientes. Usar después de actualizar el código si hay cambios en modelos.
`makemigrations`	`python manage.py makemigrations`	Genera nuevos archivos de migración basados en cambios en `models.py`.
`Rebuild`	`docker compose up -d --build`	Fuerza la reconstrucción de las imágenes. Obligatorio si se modifica `requirements.txt` o el `Dockerfile`.
`test`	`python manage.py test`	Ejecuta la suite de pruebas unitarias y de integración de Django.

Comandos manuales adicionales

- Enviar mensaje de prueba a Kafka:

Se incluye un script Python para verificar el flujo de mensajes.

```
python                                                    enviar_prueba.py
```

Configuración: `{'bootstrap.servers': 'localhost:9092'}`. Confirma que el productor puede conectar con el broker y que el tópico `topic-calificaciones` está operativo.

- **Reiniciar un servicio específico:**

```
docker                compose                restart                backend
```

Útil para aplicar cambios de código si el "hot reloading" falla o para reiniciar conexiones perdidas.

Solución de problemas (troubleshooting) y FAQ

Esta sección aborda los incidentes más frecuentes reportados durante el despliegue y operación.[1]

Errores comunes

Caso 1: WSL debe actualizarse (Windows)

- **Síntoma:** Error explícito indicando "WSL2 debe actualizarse" al intentar iniciar Docker.
- **Causa:** El kernel de Linux en Windows está desactualizado.
- **Solución:** Ejecutar en PowerShell como administrador:
  - `wsl --update`
  - `shutdown /r /t 0` (Reinicio inmediato del sistema).

Caso 2: Docker no levanta contenedores (Exit Code 137 o similar)

- **Síntoma:** Los contenedores se inician y mueren inmediatamente.
- **Causa:** Falta de memoria RAM asignada a Docker (Kafka y Java son intensivos en memoria) o imágenes corruptas.
- **Solución:**
  - Aumentar la RAM en Docker Desktop (mínimo 4 GB recomendado).
  - Limpieza profunda: `docker system prune -af` (Borra todo: imágenes, redes, caché).
  - Reconstruir: `compose-helper-w rebuild`.

Caso 3: Backend "Unhealthy" o fallo de inicio

- **Síntoma:** El servicio backend no llega a estado `running`.
- **Causa:** Puerto 8000 ocupado por otro proceso en el host, o variables de entorno (`.env`) mal configuradas.
- **Solución:**
    - Identificar el proceso: `lsof -i :8000` (Mac/Linux) o `netstat -ano | findstr :8000` (Windows).
    - Matar el proceso conflictivo: `kill -9 <PID>` o `taskkill /PID <PID> /F`.
    - Verificar `.env` y reiniciar.

Caso 4: Kafka no recibe mensajes

- **Síntoma:** El script `enviar_prueba.py` da timeout o el consumidor no procesa datos.
- **Solución:**
    - Verificar logs: docker-compose `logs kafka`.
    - Hay que confirmar que `ADVERTISED_LISTENERS` en el `docker-compose.yml` apunte correctamente a `localhost:9092` para el tráfico externo.

Caso 5: Kafka falla con "Connection Refused" o conflicto de puertos

- **Síntoma: El contenedor de Kafka se detiene inmediatamente** (Exited 1) y los logs muestran `IllegalArgumentException: Each listener must have a different port`.
- **Causa:** Conflicto de direccionamiento en Docker. Kafka intentaba usar el puerto 9092 tanto para tráfico interno (entre contenedores) como externo (hacia el host Windows/Linux).
- **Solución Implementada:** Se reconfiguró el archivo `docker-compose.yml` utilizando la directiva KAFKA_ADVERTISED_LISTENERS con puertos diferenciados (Split Horizon DNS):
    - **Puerto 29092:** Exclusivo para tráfico interno (Backend Django -> Kafka).
    - **Puerto 9092:** Exclusivo para tráfico externo (Host/Developers -> Kafka).
    - **Acción:** Actualizar la variable de entorno en el backend a KAFKA_SERVER=`kafka:29092`.

Preguntas frecuentes (FAQ)

- ¿Qué hacer si Docker no inicia?

Cierre sesión en su sistema operativo, reinicie Docker Desktop manualmente y verifique que la virtualización esté habilitada en la BIOS.

- ¿Cómo reinstalar todo desde cero?

La forma más segura de obtener un estado limpio es ejecutar docker system prune -af seguido del script de instalación (install_windows.bat o install_linux).

- ¿Por qué usamos microservicios en lugar de un monolito?

Para garantizar escalabilidad (podemos escalar el backend sin tocar la base de datos), modularidad (equipos pueden trabajar en servicios separados) y separación de responsabilidades.

- ¿NUAM funciona sin Docker?

No. La arquitectura depende de versiones específicas de servicios (Zookeeper, Kafka, MySQL) y configuraciones de red que están codificadas en los contenedores. Intentar ejecutarlo manualmente en el host ("bare metal") conducirá a conflictos de dependencias ("Dependency Hell").

Estructura del proyecto

La organización del código fuente sigue una estructura estándar para facilitar la navegación y el mantenimiento.

Anexos

Variables de entorno (.env)

(Este archivo no se incluye en el repositorio por seguridad, pero se genera una plantilla `env.example`).

- `DEBUG`: `True` (Desarrollo), `False` (Producción).
- `SECRET_KEY`: Llave criptográfica de Django.
- `DB_NAME`, `DB_USER`, `DB_PASSWORD`, `DB_HOST`: Credenciales de base de datos.
- `KAFKA_BOOTSTRAP_SERVERS`: Dirección del broker de Kafka.

Calidad de código y estándares

El proyecto utiliza herramientas automatizadas para mantener la calidad del código, configuradas en `.pre-commit-config.yaml`[1]:

- **Black:** Formateador de código intransigente.
- **Flake8:** Linter para estilo y errores lógicos (límite 120 caracteres).
- **Isort:** Ordenamiento de importaciones.
- **Pre-commit hooks:** Verifican espacios en blanco y sintaxis YAML antes de cada commit.