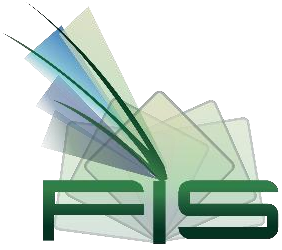
Escuela Politécnica Nacional

Facultad de Ingeniería de Sistemas



ISWD633
Construcción y Evolución de Software

PROYECTO 02

Análisis Integral del Stack Angular, NestJS y PostgreSQL

Integrantes: Jonathan Tipán, Javier Quilumba
Paralelo: GR2SW
Fecha de Entrega: 31 de enero de 2026
Docente: Vicente Eguez

Semestre 2025-B

Tabla de contenido

1. Marco Teórico de la Evolución de Software y el Paradigma de los Sistemas de Tipo E

La construcción de sistemas de software en el panorama tecnológico contemporáneo ha trascendido la mera escritura de código para convertirse en un ejercicio de gestión de la evolución continua. Este fenómeno se fundamenta en las investigaciones de Meir Lehman y Laszlo Belady, quienes desde 1974 establecieron que el software no es una entidad estática, sino un organismo que habita en un entorno operativo en constante cambio. En particular, los sistemas desarrollados con el stack NAP (NestJS, Angular y PostgreSQL) se clasifican como programas de tipo E (E-type), definidos como aquellos que mecanizan actividades humanas o sociales en el mundo real. La validez de un sistema de tipo E no reside únicamente en su cumplimiento de una especificación inicial, sino en su capacidad para satisfacer las necesidades cambiantes de sus usuarios y del entorno en el que opera.

La dinámica de estos sistemas está regida por ocho leyes fundamentales que describen las fuerzas de equilibrio entre la innovación y la estabilidad. La primera ley, la ley del cambio continuo, postula que un sistema de tipo E debe adaptarse constantemente o se volverá progresivamente menos satisfactorio. En el contexto de Angular y NestJS, esta ley se manifiesta a través de los ciclos de actualización semestrales de los frameworks, que introducen nuevas capacidades como los Signals en Angular 19 o mejoras en la inyección de dependencias en NestJS, obligando a los desarrolladores a evolucionar el código para evitar la obsolescencia técnica. La segunda ley, la complejidad creciente, advierte que a medida que un sistema evoluciona, su estructura tiende a degradarse a menos que se invierta un esfuerzo deliberado en el mantenimiento preventivo y la refactorización. Este principio justifica la adopción de arquitecturas modulares y patrones de capas, que actúan como contramedidas frente a la entropía del software.

Para medir la salud de esta evolución, se emplean métricas que cuantifican la calidad interna del sistema. El Índice de Mantenibilidad (MI) es una métrica compuesta que evalúa la facilidad con la que se pueden realizar cambios en el código fuente. La fórmula matemática para calcular este índice integra el Volumen de Halstead (HV), que mide la complejidad algorítmica; la Complejidad Ciclomática de McCabe (CC), que cuenta los caminos linealmente independientes a través del código; y las Líneas de Código (LOC). Un MI elevado indica un software con una arquitectura limpia y modular, mientras que un descenso en este valor sugiere que el sistema se está volviendo rígido y propenso a errores, lo que Lehman denomina como la ley de la calidad declinante.

Ley de Lehman	Definición Aplicada al Stack NAP	Implicación en la Construcción
Cambio Continuo	Necesidad de actualizar Angular y NestJS para mantener la relevancia funcional.	Planificación de migraciones y actualizaciones de dependencias periódicas.
Complejidad Creciente	La adición de nuevas historias de usuario en Jira tiende a desordenar	Inversión constante en refactorización y revisión de código

	la arquitectura modular.	(Peer Review).
Autorregulación	Los procesos de evolución muestran tendencias predecibles en el tamaño y la tasa de errores.	Uso de métricas de velocidad y calidad para planificar sprints en Jira.
Conservación de la Estabilidad Organizacional	La tasa de trabajo efectiva es invariante durante la vida del sistema, independientemente de los recursos.	Enfoque en la eficiencia de procesos de CI/CD sobre el simple aumento de personal (Ley de Brooks).
Conservación de la Familiaridad	El crecimiento incremental por versión debe ser limitado para que el equipo mantenga el dominio del sistema.	Fragmentación de grandes características en pequeñas historias de usuario manejables.
Crecimiento Continuo	El contenido funcional debe aumentar para mantener la satisfacción del usuario a largo plazo.	Desarrollo de nuevas capacidades y servicios en el backend de NestJS.
Calidad Declinante	La calidad percibida disminuye si no se adapta rigurosamente a los cambios del entorno.	Mantenimiento preventivo y auditorías de código constantes.
Sistema de Retroalimentación	La evolución constituye un sistema de bucles de retroalimentación de múltiples niveles.	Integración de GitHub, Jira y Slack para capturar feedback técnico y operativo.

La comprensión de estas leyes permite a los arquitectos de software diseñar sistemas que no solo funcionen en el presente, sino que posean una estructura interna que facilite su transformación futura. La elección de PostgreSQL como motor de base de datos relacional proporciona la consistencia necesaria para soportar estas transformaciones, permitiendo que el esquema de datos evolucione mediante migraciones controladas que aseguren la integridad de la información a través de las diferentes versiones del aplicativo.

2. Arquitecturas Modulares y Patrones de Capas: El Modelo MVC Evolucionado

La arquitectura modular es el pilar fundamental que permite a un sistema resistir la ley de la complejidad creciente. Al dividir el sistema en módulos independientes pero interrelacionados, se reduce el acoplamiento y se fomenta la cohesión, lo que simplifica tanto las pruebas como el mantenimiento. En el stack NAP, esta modularidad se implementa mediante la organización del backend en NestJS y el frontend en Angular utilizando patrones de capas y estructuras orientadas a servicios.

2.1. Estructura de Backend con NestJS y el Patrón de Capas

NestJS adopta una arquitectura fuertemente inspirada en Angular, promoviendo el uso de módulos como contenedores de lógica relacionada. La implementación de una arquitectura de capas (Layered Architecture) es la práctica recomendada para asegurar una clara separación de preocupaciones. En este modelo, el sistema se divide tradicionalmente en tres capas principales:

1. **Capa de Controladores**: Es el punto de entrada de las solicitudes externas. Su responsabilidad se limita a manejar el protocolo de transporte (HTTP/REST), validar los datos de entrada mediante Objetos de Transferencia de Datos (DTO) y delegar la ejecución de la lógica a la capa de servicios.
2. **Capa de Servicios (Lógica de Negocio)**: Es el núcleo del sistema. Aquí se implementan las reglas de negocio y se orquestan las operaciones necesarias para cumplir con los requerimientos. Los servicios son agnósticos a la base de datos y al protocolo de transporte, lo que facilita su reutilización y prueba en aislamiento.
3. **Capa de Persistencia (Repositorios)**: Gestiona la interacción con PostgreSQL. Utilizando ORMs como TypeORM, Prisma o Drizzle, esta capa abstrae las consultas SQL y proporciona una interfaz orientada a objetos para la manipulación de datos.

Esta separación permite que el sistema evolucione de un monolito modular hacia una arquitectura de microservicios sin necesidad de reescribir la lógica de negocio, simplemente extrayendo módulos específicos hacia servicios independientes cuando la escala lo demande.

2.2. Arquitectura de Frontend con Angular: Componentes y Señales

En el frontend, Angular propone una estructura basada en componentes que encapsulan la lógica de la interfaz de usuario, el marcado HTML y los estilos CSS. Con la llegada de Angular 19, la evolución hacia una arquitectura más reactiva se ha consolidado mediante el uso de Signals para la gestión del estado local. La modularidad en Angular se logra a través de NgModule o, más recientemente, mediante Standalone Components, que eliminan la necesidad de módulos tradicionales para simplificar la estructura del proyecto.

El patrón de diseño recomendado para el frontend es el de "Smart and Dumb Components". Los "Smart Components" (o componentes contenedores) gestionan el estado y se comunican con los servicios, mientras que los "Dumb Components" (o componentes de presentación) reciben datos mediante decoradores @Input y emiten eventos mediante @Output, garantizando una interfaz de usuario altamente reutilizable y fácil de probar.

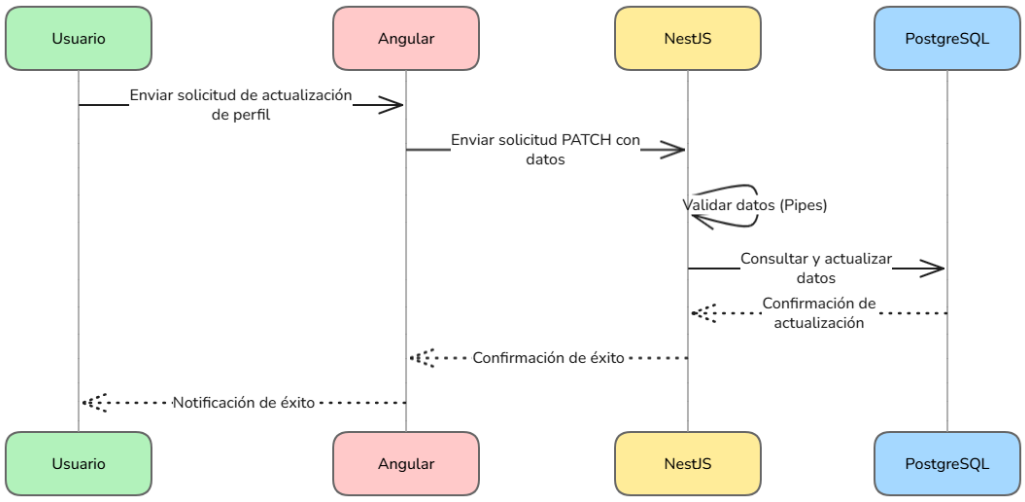
Capa / Patrón	**Responsabilidad en NestJS (Backend)**	**Responsabilidad en Angular (Frontend)**
Presentación / UI	Controladores (Exposición de endpoints REST/GraphQL).	Componentes (Visualización de datos y captura de eventos).
Lógica de Aplicación	Servicios (Reglas de negocio y flujos de trabajo).	Servicios de UI (Gestión de estado y lógica de presentación).

Acceso a Datos	Repositorios / ORM (Persistencia en PostgreSQL).	Servicios de API (Consumo de endpoints mediante HttpClient).
Entidades	Modelos de base de datos y esquemas.	Interfaces de TypeScript y DTOs de cliente.

2.3. Diagrama de Interacción y Flujo de Datos

El flujo de interacción entre el usuario y el sistema sigue una secuencia lógica que atraviesa todas las capas arquitectónicas. Para una operación común, como la actualización del perfil de un usuario, la secuencia de eventos se describe a continuación:

1. **Captura del Evento**: El usuario interactúa con un componente de Angular. Este componente captura los datos del formulario y llama a un método en el servicio de Angular correspondiente.
2. **Transmisión HTTP**: El servicio de Angular utiliza el HttpClient para enviar una solicitud PATCH a la API de NestJS, adjuntando un token JWT para autenticación.
3. **Recepción y Validación**: El controlador en NestJS recibe la solicitud. Los Pipes de validación verifican que el cuerpo de la solicitud cumpla con el esquema definido en el DTO.
4. **Procesamiento de Negocio**: El controlador delega la tarea al servicio. El servicio realiza comprobaciones adicionales (por ejemplo, verificar si el correo electrónico ya está en uso) y formatea los datos para la persistencia.
5. **Persistencia de Datos**: El servicio llama al repositorio, que traduce la operación en una consulta SQL ejecutada en PostgreSQL mediante el ORM.
6. **Respuesta Sincronizada**: Una vez confirmada la persistencia, NestJS devuelve un código de estado 200 OK. Angular recibe la respuesta, actualiza el estado local (Signal) y notifica al usuario del éxito de la operación.



Este flujo garantiza que cada componente del sistema tenga una responsabilidad única y clara, facilitando la identificación de fallos y la implementación de mejoras sin afectar colateralmente a otras partes del sistema.

3. Modelo de Datos y Persistencia Evolutiva con PostgreSQL

La base de datos PostgreSQL actúa como el ancla de estabilidad en un sistema que cambia rápidamente. A diferencia de las bases de datos NoSQL, PostgreSQL ofrece un esquema rígido que garantiza la integridad referencial, lo cual es vital para aplicaciones empresariales y de gestión donde la precisión de los datos es innegociable. Sin embargo, esta rigidez debe gestionarse cuidadosamente para permitir la evolución del software.

3.1. Estrategias de Mapeo Objeto-Relacional (ORM)

En el ecosistema NestJS de 2024-2025, la elección del ORM ha evolucionado. Mientras que TypeORM sigue siendo el estándar por su madurez e integración nativa con NestJS, Prisma ha ganado terreno por su seguridad de tipos y facilidad de uso. Emergiendo como una opción de alto rendimiento, Drizzle ORM se recomienda para proyectos donde la latencia es crítica y se prefiere un enfoque más cercano al SQL puro pero con seguridad de tipos en TypeScript.

La gestión de la evolución de la base de datos se realiza mediante el sistema de migraciones del ORM. Cada cambio en el modelo de datos (agregar una columna, crear una tabla, modificar una relación) genera un archivo de migración que se versiona en el repositorio de código. Esto permite que todos los desarrolladores y los entornos de CI/CD trabajen con la misma estructura de base de datos de forma predecible.

3.2. Optimización y Escalabilidad del Almacenamiento

Para soportar el crecimiento continuo exigido por las leyes de Lehman, PostgreSQL ofrece capacidades avanzadas como el particionamiento de tablas, la indexación eficiente mediante índices GIN y GiST, y el soporte nativo para JSONB, lo que permite integrar flexibilidad de esquema dentro de un entorno relacional. La evolución del sistema a menudo requiere pasar de una base de datos centralizada a una arquitectura con réplicas de lectura para manejar cargas elevadas, una transición que NestJS facilita mediante la configuración de múltiples conexiones en el módulo de base de datos.

4. Gestión de Proyectos con Jira: Historias de Usuario y Trazabilidad

La evolución del software no es solo técnica, sino también administrativa y estratégica. El uso de Jira como herramienta de gestión permite alinear el desarrollo con los objetivos del negocio mediante la implementación de metodologías ágiles como Scrum o Kanban. La unidad fundamental de trabajo es la historia de usuario, que describe una capacidad del sistema desde la perspectiva del usuario final.

4.1 Definición de Historias de Usuario con Criterios de Aceptación

Para asegurar que el desarrollo cumpla con las expectativas, cada historia de usuario debe seguir el modelo INVEST y contar con criterios de aceptación detallados que sirvan de base

para las pruebas de calidad.

Elemento de Jira	Descripción	Ejemplo para un Sistema de Gestión
Épica	Gran cuerpo de trabajo que abarca varios sprints.	Gestión de Inventario y Almacenes.
Historia de Usuario	Requerimiento específico con valor de negocio.	Como gestor de almacén, quiero recibir una alerta de bajo stock para evitar la falta de suministros.
Criterios de Aceptación	Condiciones de éxito para la historia.	1. La alerta se dispara cuando el stock < 10 unidades. 2. Notificación vía Slack y correo electrónico.
Tarea Técnica	Trabajo de ingeniería necesario para la historia.	Crear trigger en PostgreSQL para monitorear nivel de stock.
Bug	Defecto detectado en una funcionalidad existente.	El contador de stock no se actualiza correctamente tras devoluciones.

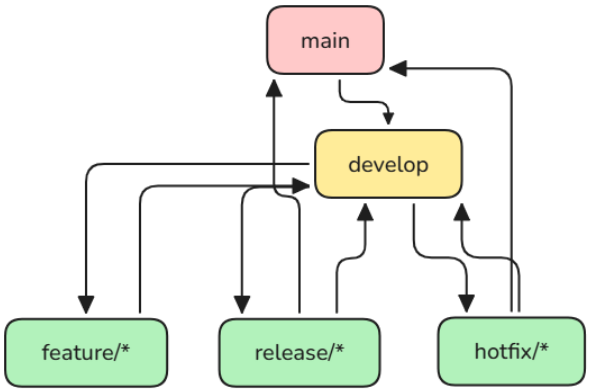
La vinculación de estas historias en Jira con el repositorio de GitHub es un requisito crítico para la trazabilidad. Al incluir el ID del ticket de Jira (ej. TJ-101) en el nombre de la rama y en los mensajes de commit, el sistema crea un vínculo automático que permite ver el progreso del código directamente desde el tablero de Jira.

4.2. Trazabilidad y Evolución del Sistema

La trazabilidad permite responder a preguntas críticas durante la evolución del software: ¿Por qué se cambió este bloque de código? ¿Qué requerimiento de negocio motivó esta actualización de la base de datos? ¿Quién revisó y aprobó este cambio?. Esta visibilidad es esencial para cumplir con la ley de conservación de la familiaridad, ya que permite a los nuevos miembros del equipo comprender el contexto histórico de las decisiones técnicas sin necesidad de una documentación manual exhaustiva que a menudo queda desactualizada.

5. Modelo de Ramificación GitFlow y Gestión de Versiones

Un sistema que evoluciona constantemente requiere un modelo de control de versiones que permita el desarrollo paralelo, las pruebas rigurosas y las correcciones de emergencia sin interferencias mutuas. GitFlow se ha consolidado como el estándar para proyectos que manejan ciclos de lanzamiento formales y requieren una gestión estricta de la rama de producción.



5.1. Dinámica de Ramas y Ciclo de Vida del Código

GitFlow define dos ramas principales de larga duración:

- **main**: Representa el estado actual del código en producción. Solo contiene código probado, estable y listo para el usuario final.
- **develop**: Es la rama de integración para el desarrollo. Aquí es donde se fusionan las nuevas funcionalidades que formarán parte de la próxima versión.

Complementando estas, existen ramas temporales que facilitan el flujo de trabajo:

- **feature/**: Ramas para el desarrollo de nuevas historias de usuario. Nacen de develop y vuelven a develop tras pasar por revisión por pares y pruebas automatizadas.
- **release/**: Ramas de preparación para un nuevo lanzamiento. Se crean desde develop para realizar tareas de pulido final, corrección de errores de integración y actualización de versiones. Finalmente, se fusionan en main y en develop.
- **hotfix/**: Ramas críticas que nacen de main para solucionar fallos urgentes en producción. Su cierre implica una fusión hacia main y hacia develop para asegurar que el parche persista en futuras versiones.

5.2. Flujo Detallado de Hotfixes y Versionado Semántico

La gestión de un hotfix es un proceso quirúrgico que ilustra la disciplina necesaria en la evolución del software. Cuando se detecta un fallo crítico en producción, el equipo detiene el flujo normal de trabajo para esa versión y sigue estos pasos:

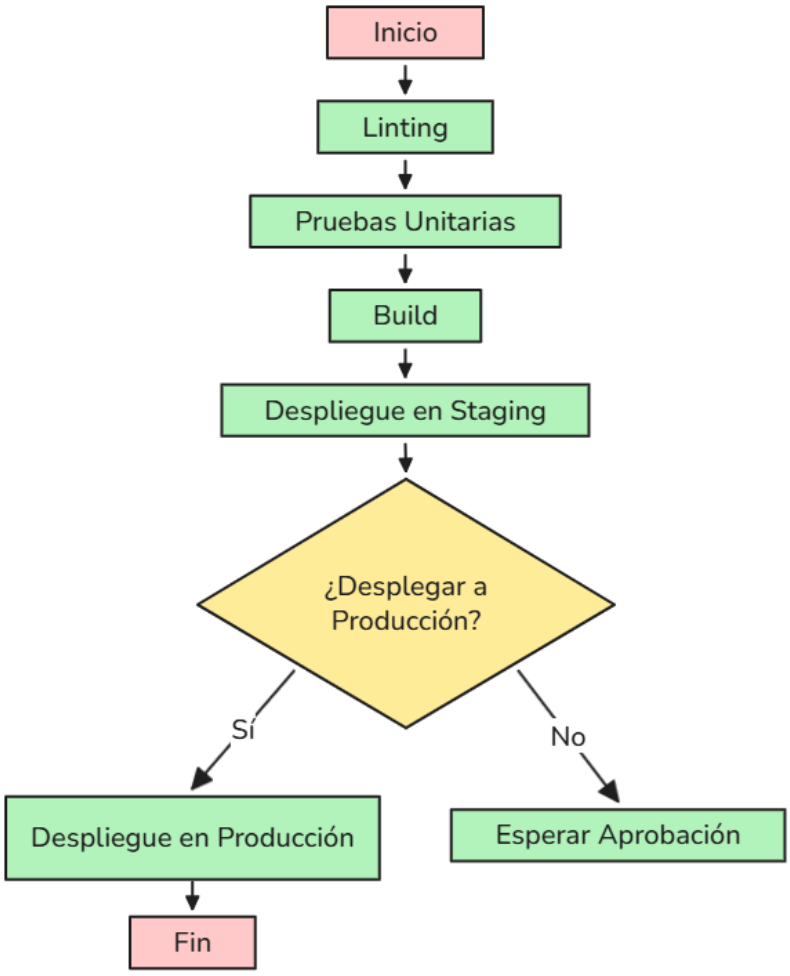
1. **Aislamiento**: Se crea la rama hotfix/v1.0.1 a partir del último tag en main.
2. **Corrección**: Se aplica la solución y se añaden pruebas unitarias específicas que verifiquen la corrección del fallo y prevengan regresiones.
3. **Integración**: La corrección se fusiona en main y se genera un nuevo tag. Inmediatamente después, se fusiona en develop (o en la rama de release actual si existe).
4. **Cierre**: Se elimina la rama de hotfix y se despliega la nueva versión parcheada mediante el pipeline de CI/CD.

El versionado sigue la convención SemVer (MAJOR.MINOR.PATCH). Un cambio en PATCH indica correcciones de errores; MINOR indica nuevas funcionalidades compatibles hacia atrás;

y MAJOR indica cambios que rompen la compatibilidad. Este sistema proporciona claridad a los usuarios y a los integradores sobre el impacto esperado de cada actualización, facilitando la adopción de nuevas versiones.

6. Automatización de la Evolución mediante Pipelines de CI/CD

La automatización es el catalizador que permite que las leyes de Lehman no se conviertan en un obstáculo infranqueable. Un pipeline de Integración Continua y Despliegue Continuo (CI/CD) bien configurado reduce el error humano y acelera el tiempo de llegada al mercado (Time-to-Market).



6.1. Configuración Práctica con GitHub Actions

GitHub Actions permite definir flujos de trabajo en archivos YAML que se disparan ante eventos específicos en el repositorio. Para un proyecto NAP, el pipeline se divide en etapas lógicas para asegurar que solo el código de alta calidad llegue a los entornos superiores.

Ejemplo:

```yaml
name: NAP Pipeline - CI/CD
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main, develop]

jobs:
  quality-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Use Node.js 22
        uses: actions/setup-node@v4
        with:
          node-version: 22
          cache: 'npm'
      - run: npm ci
      - run: npm run lint
      - run: npm run format:check
      - run: npm run test:ci -- --coverage

  build-and-deploy:
    needs: quality-check
    if: github.event_name == 'push'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Build NestJS & Angular
        run: |
         npm run build:backend
         npm run build:frontend
      - name: Deploy to Staging (Firebase/Heroku)
        if: github.ref == 'refs/heads/develop'
        run: npm run deploy:staging
        env:
          FIREBASE_TOKEN: ${{ secrets.FIREBASE_TOKEN }}
      - name: Deploy to Production
        if: github.ref == 'refs/heads/main'
        run: npm run deploy:production
        env:
          FIREBASE_TOKEN: ${{ secrets.FIREBASE_TOKEN }}
```

6.2. Gestión de Entornos y Despliegue Seguro

El despliegue se organiza en entornos progresivos:

- **Desarrollo (Local)**: Donde los desarrolladores prueban sus cambios contra una base de datos PostgreSQL local o en contenedores Docker.
- **Staging (Pre-producción)**: Un entorno idéntico a producción donde se realizan pruebas de integración, pruebas de carga y validación por parte del cliente. Se despliega automáticamente desde la rama develop.
- **Producción**: El entorno final de cara al usuario. El despliegue se activa únicamente tras la fusión exitosa en la rama main y la creación de un tag de versión.

Para el backend de NestJS, el proceso de CD debe incluir la ejecución de las migraciones de base de datos en PostgreSQL. Esto asegura que el código nuevo siempre encuentre el esquema de base de datos esperado, evitando errores de tiempo de ejecución tras el despliegue.

7. Protocolos de Revisión por Pares (Peer Review) y Estándares de Calidad

La revisión de código por pares es la herramienta más efectiva para combatir la ley de la complejidad creciente y asegurar que el conocimiento no quede centralizado en una sola persona. Un proceso de revisión robusto actúa como un filtro de calidad y un mecanismo de mentoría continua.

7.1. Checklist de Revisión Técnica para el Stack NAP

Los revisores deben verificar el cumplimiento de estándares específicos para cada tecnología, asegurando que el código sea mantenible, seguro y eficiente.

Categoría	Puntos de Verificación	Impacto en la Evolución
Arquitectura	¿La lógica de negocio reside en el Servicio y no en el Controlador de NestJS?	Facilita la reutilización de lógica en microservicios futuros.
Angular	¿Se están usando Pipes Asíncronos (async) para evitar fugas de memoria?	Mejora el rendimiento y la estabilidad del frontend a largo plazo.
TypeScript	¿Se ha evitado el uso de any y se han definido interfaces para todas las respuestas de API?	Reduce los errores silenciosos y mejora la legibilidad del código.
Seguridad	¿Los DTOs incluyen validaciones exhaustivas (class-validator)?	Protege el sistema contra inyecciones de datos maliciosos.
Base de Datos	¿La migración de PostgreSQL es reversible (método down)?	Permite realizar retrocesos (rollbacks) seguros en caso de fallo.

7.2. Casos de Prueba Específicos para Validación

Cada funcionalidad debe ir acompañada de pruebas que validen tanto el camino feliz como los casos de error. En NestJS, las pruebas unitarias con Jest permiten mockear las dependencias de base de datos para probar la lógica de negocio de forma instantánea.

Ejemplo de Caso de Prueba para Servicio de Pedidos:

- **Escenario**: Creación de un pedido con stock insuficiente.
- **Entrada**: ID de Producto, Cantidad solicitada > Stock disponible.
- **Resultado Esperado**: El servicio debe lanzar una BadRequestException con un mensaje descriptivo. No se debe crear ningún registro en la tabla de pedidos de PostgreSQL.

En Angular, las pruebas de componentes con Jasmine/Karma deben verificar la correcta representación de los datos y el manejo de eventos.

- **Escenario**: Visualización de error de conexión.
- **Entrada**: El servicio de API devuelve un error 500.
- **Resultado Esperado**: El componente debe mostrar un banner de error amigable al usuario y no bloquear la interfaz.

8. Documentación de APIs y Ecosistema de Integración

La comunicación entre el frontend de Angular y el backend de NestJS se rige por un contrato: la API. Documentar este contrato es vital para el desarrollo paralelo y la evolución del sistema.

8.1. Swagger y el Contrato de API en NestJS

NestJS utiliza el decorador @nestjs/swagger para generar automáticamente una especificación OpenAPI 3.0 basada en el código fuente. Esto garantiza que la documentación nunca esté desincronizada con la implementación real. La configuración en main.ts permite exponer esta documentación en una ruta interactiva (ej. /api/docs), facilitando las pruebas manuales por parte de desarrolladores y analistas.

8.2. Sincronización con Postman y Consumo en el Frontend

Para equipos que prefieren Postman para el desarrollo y las pruebas, la especificación generada por Swagger puede importarse directamente para crear colecciones de peticiones. Además, la calidad de la documentación de Swagger (usando decoradores como @ApiProperty y @ApiResponse) permite la generación automática de clientes de API en Angular, lo que reduce el código manual necesario para el consumo de servicios y asegura que los tipos de datos en el cliente coincidan exactamente con los del servidor.

9. Integración GitHub-Jira-Slack para la Trazabilidad

Para asegurar que la evolución del sistema sea transparente y colaborativa, se implementa un ecosistema de integración que conecta las herramientas de desarrollo, gestión y comunicación.

1. **GitHub e Jira**: La integración bidireccional permite que Jira se actualice automáticamente cuando se abre un PR o se fusiona una rama. Los desarrolladores pueden ver sus tareas asignadas desde el entorno de desarrollo y los gestores pueden ver el estado técnico desde el tablero de Jira.
2. **Jira y Slack**: Mediante webhooks, se envían notificaciones a canales específicos de Slack cuando una tarea cambia de estado (ej. de "Testing" a "Done") o cuando se crea un nuevo bug crítico.
3. **GitHub y Slack**: Los pipelines de CI/CD notifican el éxito o fallo de las construcciones y despliegues directamente en Slack, permitiendo una reacción inmediata del equipo DevOps ante fallos en producción o staging.

Este flujo de información continuo reduce el ruido comunicativo y asegura que todos los interesados tengan una visión unificada del estado del sistema, facilitando la toma de decisiones basada en datos reales de desarrollo.

10. Mantenimiento de Software: Tipos y Estrategias Evolutivas

El mantenimiento es la fase donde se aplican activamente las leyes de Lehman para preservar el valor del software a lo largo del tiempo. Se categoriza en cuatro tipos fundamentales que deben planificarse dentro del ciclo de vida del proyecto.

10.1. Categorías de Mantenimiento Aplicadas

- **Mantenimiento Correctivo**: Es la respuesta reactiva a fallos. En el stack NAP, implica corregir errores de lógica en NestJS o bugs visuales en Angular tras su detección en producción. La velocidad de respuesta es la métrica clave aquí.
- **Mantenimiento Adaptativo**: Modificaciones para mantener el sistema funcional en entornos cambiantes. Por ejemplo, actualizar el código para que sea compatible con una nueva versión de PostgreSQL o ajustar las cabeceras de seguridad debido a cambios en las políticas de CORS de los navegadores.
- **Mantenimiento Perfectivo**: Mejoras solicitadas por los usuarios para aumentar el valor del producto. Incluye la optimización de consultas SQL lentas en PostgreSQL o la mejora de la experiencia de usuario (UX) en componentes específicos de Angular.
- **Mantenimiento Preventivo**: Actividades proactivas para evitar fallos futuros. La refactorización de código "spaghetti" en módulos limpios, la actualización de librerías con vulnerabilidades de seguridad y la mejora de la cobertura de pruebas unitarias son ejemplos críticos de este tipo de mantenimiento.

Tipo de Mantenimiento	Ejemplo Práctico en el Proyecto	Objetivo Principal
Correctivo	Reparar un fallo en el proceso de login de NestJS.	Restaurar la funcionalidad básica.
Adaptativo	Migrar la base de datos PostgreSQL a una nueva versión de	Mantener la compatibilidad operativa.

	Cloud Hosting.	
Perfectivo	Añadir un sistema de "Infinite Scroll" en el catálogo de Angular.	Mejorar la experiencia del usuario y el rendimiento percibido.
Preventivo	Actualizar dependencias mediante npm audit fix y refactorizar servicios complejos.	Reducir la deuda técnica y riesgos de seguridad.

11. Conclusiones y Recomendaciones

La construcción de software bajo el stack Angular, NestJS y PostgreSQL representa un compromiso con la calidad y la escalabilidad. Sin embargo, el éxito a largo plazo no depende solo de la elección tecnológica inicial, sino de la disciplina en la gestión de su evolución. La arquitectura modular y el patrón de capas actúan como el armazón que permite al sistema crecer sin colapsar bajo su propia complejidad. La automatización mediante CI/CD y la rigurosidad en los flujos de trabajo de GitFlow son esenciales para garantizar que cada cambio en el código sea una mejora y no un riesgo para la estabilidad de la producción.

Se recomienda a las organizaciones adoptar una cultura de revisión por pares y mantenimiento preventivo como parte integral de su proceso de desarrollo, reconociendo que el esfuerzo invertido en la calidad interna hoy se traduce en una mayor agilidad funcional mañana. La integración total de las herramientas de gestión (Jira), desarrollo (GitHub) y comunicación (Slack) no es un lujo, sino una necesidad operativa para mantener la trazabilidad y la alineación entre los objetivos de negocio y la ejecución técnica. En última instancia, un sistema de software saludable es aquel que acepta el cambio como una constante y se estructura de manera que la evolución sea un proceso fluido, seguro y predecible.

12. Referencias

[1] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060-1076, Sept. 1980.
[2] V. Driessen, "A successful Git branching model," *nvie.com*, Jan. 5, 2010.
[3] *IEEE Standard for Software Verification and Validation*, IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998), June 2005.
[4] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution—the nineties view," in *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, 1997, pp. 20-32.
[5] G. Yusuf, "Synchronize Postman Collection Automatically in NestJS," *Medium*, Feb. 1, 2025.
[6] "Documentation: OpenAPI (Swagger)," *NestJS Official Documentation*, 2025.
[7] "Angular Architecture Guide to Building Maintainable Applications at Scale," *Nx Documentation*, Apr. 2025.