Plan de Desarrollo para un Sistema de Gestión Hospitalaria Distribuido con NestJS, Prisma, FDWs y QueueT

I. Resumen Ejecutivo

Este documento presenta un plan de desarrollo detallado para un Sistema de Gestión Hospitalaria (SGH) distribuido, diseñado para operar con bases de datos PostgreSQL en entornos híbridos: AWS, Azure y un servidor local. La arquitectura se fundamenta en microservicios desarrollados con Node.js y TypeScript, utilizando NestJS como *framework* principal y Prisma como ORM para la interacción con las bases de datos.

La clave para la interoperabilidad de datos reside en una estrategia dual:

1. **PostgreSQL Foreign Data Wrappers (FDWs)**: Para habilitar consultas de lectura eficientes y en tiempo real entre las bases de datos distribuidas, permitiendo el acceso transversal a información compartida como historias clínicas y el inventario de farmacia sin duplicación de datos.[1]
2. **QueueT (Message Broker basado en Redis)**: Para gestionar la consistencia de datos en operaciones de escritura que abarcan múltiples microservicios, implementando patrones de comunicación asíncrona y el patrón Saga para asegurar la consistencia eventual en transacciones distribuidas .

El plan aborda la complejidad de la gestión de roles y permisos (RBAC), la seguridad de contraseñas (hashing con bcryptjs y flujos de restablecimiento), y la conectividad de red segura entre los entornos de nube y local. **Para el entorno de desarrollo, se priorizará la reducción de costos mediante el uso extensivo de instancias de bases de datos locales y la configuración de niveles de servicio económicos en la nube, con la opción de detener los recursos cuando no estén en uso.** Se proporciona una estructura de proyecto monorepo, una guía paso a paso para la implementación y una sección dedicada a la configuración óptima de Azure Database for PostgreSQL para desarrollo. Este enfoque holístico busca construir un SGH robusto, escalable, seguro y mantenible, capaz de satisfacer los complejos requisitos

de un entorno hospitalario moderno.

II. Visión del Proyecto y Requisitos Fundamentales (Reiteración y Clarificación)

El Sistema de Gestión Hospitalaria (SGH) se concibe como una solución distribuida que equilibra la autonomía departamental con la necesidad crítica de interoperabilidad y acceso centralizado a datos compartidos.

A. Contexto del Sistema de Gestión Hospitalaria

El SGH permitirá a departamentos como Cardiología, Neurología y Urgencias gestionar de forma independiente su información específica (empleados, pacientes, citas, tratamientos) [User Query]. Paralelamente, se implementará una gestión centralizada para recursos compartidos como la Farmacia (inventario de medicamentos), Historias Clínicas y Agendas de Citas Médicas [User Query].

La coexistencia de la gestión independiente y el acceso compartido a datos es el desafío central. La solución propuesta permitirá que los servicios o departamentos posean sus datos de forma independiente, pero facilitará el acceso controlado y bajo demanda a través de FDWs para lecturas [1] y un Message Broker para la coordinación de escrituras distribuidas .

B. Desglose de Requisitos Funcionales

- **Autonomía Departamental:** Cada departamento gestiona sus Empleados, Pacientes, Citas y Prescripciones específicos [User Query]. Esto se mapea al patrón de "base de datos por servicio".[5]
- **Control de Acceso Basado en Roles (RBAC):** Diferentes roles (Médicos, Administrativos) tendrán distintos niveles de acceso y funciones, incluyendo la modificación de historias clínicas y la prescripción de medicamentos [User

Query]. Esto implica operaciones de escritura distribuidas.[8]
- **Inventario Centralizado de Farmacia:** La farmacia gestionará un inventario centralizado de Medicamentos, accesible por todos los departamentos para consulta y prescripción [User Query].
- **Acceso Transversal a Historias Clínicas:** El personal médico autorizado de cualquier departamento podrá consultar Historias_Clinicas [User Query].
- **Gestión de Contraseñas:** Encriptación (hashing) y flujos de cambio/olvido de contraseña [User Query, 45, 49, 56, 60].
- **Tablas Sugeridas:** Departamentos, Empleados, Pacientes, Citas, Historias_Clinicas, Medicamentos, Prescripciones, Equipamiento [User Query].

C. Requisitos Técnicos

- **PostgreSQL Distribuido:** Instancias de PostgreSQL en AWS, Azure y local [User Query].
- **Node.js y TypeScript:** Lenguajes de desarrollo para la capa de aplicación [User Query, 38].
- **Consultas Cruzadas entre Bases de Datos:** Capacidad de las bases de datos para consultarse entre sí (con FDWs) [User Query, 1, 2, 7, 8, 10, 19, 24, 25, 34].
- **Gestión de Contraseñas:** Hashing con bcryptjs y flujo de restablecimiento [User Query, 45, 49, 56, 60].
- **Message Broker:** Para la comunicación asíncrona y la consistencia eventual en operaciones de escritura distribuidas .

III. Plan Arquitectónico Revisado: Microservicios, Datos Distribuidos y Event-Driven

La arquitectura se basará en microservicios, cada uno con su base de datos dedicada cuando sea posible, y una comunicación orquestada por eventos para las operaciones que cruzan los límites de los servicios.

A. Paradigma de Microservicios para el SGH

La descomposición en microservicios se alinea con las capacidades de negocio y la propiedad de los datos.[5]

Servicios Propuestos:

- **Servicio de Autenticación y Autorización:** Gestiona usuarios, roles, permisos y credenciales.[8]
- **Servicios Departamentales (ej., CardiologyService, NeurologyService, UrgenciasService):** Cada uno gestiona Pacientes, Empleados (departamentales), Citas y Prescripciones para su respectivo departamento. Implementan el patrón "base de datos por servicio".[5]
- **Servicio de Farmacia:** Gestiona el inventario centralizado de Medicamentos.
- **Servicio de Historias Clínicas:** Gestiona las Historias_Clinicas como fuente única de verdad.
- **Servicio de Programación de Citas:** Coordina y gestiona las Agendas de Citas Médicas.
- **API Gateway:** Punto de entrada único para clientes, enrutamiento, autenticación y seguridad.[12]

B. Estrategia de Base de Datos Distribuida con PostgreSQL y FDWs

- **Bases de Datos por Servicio:** Para los servicios departamentales, cada uno tendrá su propia instancia PostgreSQL.[5]
- **Bases de Datos Centralizadas:** Para Medicamentos e Historias_Clinicas, habrá bases de datos PostgreSQL centralizadas, gestionadas por sus respectivos microservicios.[5]
- **PostgreSQL Foreign Data Wrappers (FDWs) para Consultas de Lectura:**
    - Permiten que una instancia de PostgreSQL acceda a datos de servidores externos como si fueran tablas locales.[1]
    - **Uso:** Principalmente para que los servicios *consumidores* lean datos de las bases de datos de los servicios *productores*. Por ejemplo, un Servicio Departamental puede usar FDW para consultar Medicamentos del Servicio de Farmacia o Historias_Clinicas del Servicio de Historias Clínicas.[4]
    - **Configuración:** Se detalla en la sección de implementación.

C. Estrategia de Comunicación Event-Driven con QueueT

Para las operaciones de escritura que afectan a múltiples servicios y para el desacoplamiento, se implementará una arquitectura basada en eventos utilizando **QueueT**.

- **QueueT como Message Broker:**
  - **Elección:** QueueT es un Message Broker "dead simple" basado en Redis para Node.js . Es ligero, fácil de configurar y rápido de implementar, ideal para este proyecto.
  - **Funcionamiento:** Permite a los servicios publicar eventos (mensajes) en colas y a otros servicios suscribirse a esos eventos para reaccionar de forma asíncrona .
  - **Redis:** QueueT utiliza Redis como su backend de almacenamiento de mensajes, lo que significa que necesitarás una instancia de Redis (local para desarrollo, o un servicio gestionado en la nube para producción).
- **Patrón Saga para Consistencia Eventual:**
  - Cuando una operación de negocio requiere cambios en múltiples bases de datos (ej., un médico prescribe un medicamento, lo que afecta la base de datos departamental y el inventario de farmacia), se utiliza el patrón Saga.[5]
  - **Flujo Saga (Ejemplo: Prescripción de Medicamento):**
    1. El Servicio Departamental (ej., Cardiología) recibe la solicitud de prescripción.
    2. Crea el registro de Prescripciones en su base de datos local.
    3. **Publica un evento** (ej., PrescripcionCreadaEvent) en QueueT .
    4. El Servicio de Farmacia **consume** el PrescripcionCreadaEvent de QueueT.
    5. Actualiza el stock del Medicamento correspondiente en su base de datos local.[19]
    6. Si el Servicio de Farmacia falla, puede publicar un evento de compensación (ej., PrescripcionFallidaEvent) para que el Servicio Departamental revierta su operación o notifique el error.[5]
- **Desacoplamiento Asíncrono:**
  - Los servicios no necesitan conocer directamente a sus consumidores. Publican eventos, y cualquier servicio interesado puede suscribirse . Esto mejora la resiliencia y la escalabilidad .

D. Implementación de Seguridad

- **Autenticación (JWT):** Tras la autenticación (usuario y contraseña), el Servicio de Autenticación emitirá un JSON Web Token (JWT) . Este token se incluirá en las solicitudes subsiguientes a otros microservicios para verificar la identidad del usuario.[13]
- **Autorización (RBAC):**
  - Los permisos se asignarán a los usuarios en función de sus roles (Médico, Administrativo).[12]
  - Cada microservicio implementará su propia lógica de autorización granular, verificando los roles y permisos del usuario (extraídos del JWT o consultando el Servicio de Autenticación) antes de permitir una acción.[20]
- **Hashing de Contraseñas (bcryptjs):**
  - Las contraseñas se almacenarán hasheadas utilizando bcryptjs, que es computacionalmente intensivo y utiliza "salting" para proteger contra ataques de fuerza bruta y tablas arcoíris .
  - El factor de trabajo (saltRounds) se configurará para equilibrar seguridad y rendimiento (ej., 10 rondas) .
- **Flujo de Restablecimiento de Contraseña:**
  - Generación de un token criptográficamente seguro y de un solo uso (con caducidad) almacenado en la base de datos .
  - Envío de un enlace por correo electrónico con el token .
  - Verificación del token y actualización segura de la contraseña hasheada .
- **API Gateway:** Actuará como un punto de control de seguridad inicial, manejando la autenticación y, potencialmente, la autorización básica antes de enrutar las solicitudes.[12]

IV. Plan de Desarrollo e Implementación Detallado

Este plan se divide en fases, permitiendo un enfoque incremental y la validación de componentes clave.

Fase 1: Configuración del Entorno y Microservicios Base

1. **Configuración del Monorepo:**
   - Crea la carpeta raíz del proyecto: hospital-sgh.
   - Inicializa npm workspaces en el package.json raíz para gestionar apps/* (microservicios) y packages/* (librerías compartidas) .
   - Crea las carpetas apps/ y packages/.
   - hospital-sgh/package.json:

```JSON
{
  "name": "hospital-sgh-monorepo",
  "version": "1.0.0",
  "private": true,
  "workspaces": [
    "apps/*",
    "packages/*"
  ],
  "scripts": {
    "start:dev": "npm run start:dev -ws",
    "build": "npm run build -ws",
    "lint": "npm run lint -ws",
    "test": "npm run test -ws"
  }
}
```

2. **Inicialización de Microservicios NestJS:**
   - Instala la CLI de NestJS globalmente: npm install -g @nestjs/cli .
   - Genera los proyectos base para cada microservicio dentro de apps/. Empieza con auth-service y cardiology-service como ejemplos.

```Bash
cd apps
nest new auth-service --package-manager npm
nest new cardiology-service --package-manager npm
# Repite para pharmacy-service, medical-records-service, appointment-service, api-gateway
```

   - Vuelve a la raíz del monorepo e instala todas las dependencias: npm install.[21]
3. **Integración de Prisma en cada Microservicio:**

- Para cada microservicio:
  - Navega a la carpeta del microservicio (ej., apps/auth-service).
  - Instala Prisma CLI y el cliente: npm install prisma @prisma/client.[22]
  - Inicializa Prisma: npx prisma init (crea prisma/schema.prisma y .env).[22]
  - Define los modelos de datos en schema.prisma según las tablas sugeridas y la responsabilidad del servicio (ej., User en auth-service, Paciente, Empleado, Cita, Prescripcion en cardiology-service, Medicamento en pharmacy-service, Historia_Clinica en medical-records-service).[22]
  - **Para desarrollo, configura DATABASE_URL en el .env del microservicio apuntando a su base de datos PostgreSQL local.**
  - Ejecuta la migración inicial: npx prisma migrate dev --name init.[22]
  - Crea un PrismaService en NestJS para encapsular la lógica de DB.[13]
  - Importa y provee PrismaService en el AppModule del microservicio.[13]

4. **Configuración de Paquetes Compartidos (packages/):**
   - Crea un paquete common-types para interfaces, DTOs y definiciones de eventos compartidas entre microservicios.[21]

     ```
     Bash
     mkdir packages/common-types
     cd packages/common-types
     npm init -y
     # Añade un tsconfig.json para compilar a JS
     ```

   - Crea un paquete shared-utils para utilidades comunes (ej., funciones de hashing de contraseñas con bcryptjs).

Fase 2: Despliegue de Bases de Datos y Configuración de Red

Esta fase es crítica para la interconexión de tus bases de datos, con consideraciones específicas para el entorno de desarrollo.

1. **Despliegue de Instancias PostgreSQL:**
   - **Local (Entorno de Desarrollo Principal):**
     - Asegúrate de tener una instancia de PostgreSQL local funcionando para **todos** los microservicios en desarrollo. Esta será la opción principal para minimizar costos y simplificar la configuración de red durante el desarrollo.
     - Cada microservicio se conectará a su propia base de datos (o esquema)

en esta instancia local.

- **AWS RDS for PostgreSQL (Entorno de Desarrollo - Opcional para pruebas de integración en la nube):**
  - Si necesitas probar la integración en la nube durante el desarrollo, crea instancias de PostgreSQL en AWS RDS.
  - **Para reducir costos, elige la clase de instancia "Burstable" (ej., db.t3.micro o db.t4g.micro)**.[39] Estos son los tipos de instancia más pequeños y económicos.
  - **Utiliza la función de "Start/Stop"** para detener las instancias cuando no estén en uso y evitar cargos.[28]
  - Configura **Security Groups** y **Network ACLs (NACLs)** para permitir el tráfico entrante y saliente en el puerto 5432 (PostgreSQL) **solo desde las IPs de tu entorno de desarrollo local o las IPs de tus servicios de Azure de desarrollo**.[7]
  - Habilita SSL/TLS forzado (rds.force_ssl = 1).[7]
  - Configura copias de seguridad automáticas y monitoreo básico.[39]
- **Azure Database for PostgreSQL - Flexible Server (Entorno de Desarrollo - Opcional para pruebas de integración en la nube):**
  - Si necesitas probar la integración en la nube durante el desarrollo, crea instancias de PostgreSQL en Azure Database for PostgreSQL - Flexible Server.
  - **Guía Detallada para Crear Azure Database for PostgreSQL - Flexible Server (Enfoque de Desarrollo):**
    1. **Inicia sesión en Azure Portal:** Ve a portal.azure.com.[24]
    2. **Busca el servicio:** En la barra de búsqueda, escribe "Azure Database for PostgreSQL" y selecciona "Azure Database for PostgreSQL - Flexible Servers".[24]
    3. **Haz clic en "Crear":** Se abrirá la página de configuración del servidor.[24]
    4. **Pestaña "Básico":**
       - **Suscripción:** Selecciona tu suscripción de Azure.[24]
       - **Grupo de recursos:** Selecciona uno existente o crea uno nuevo.[24]
       - **Nombre del servidor:** Introduce un nombre único (ej., dev-pharmacy-db-server).[24]
       - **Región:** Elige la región geográfica más cercana o adecuada.[24]
       - **Versión de PostgreSQL:** Selecciona la versión deseada (mínimo 12 recomendado).[24]
       - **Tipo de carga de trabajo: Elige "Desarrollo"**.[24] Esto seleccionará automáticamente un nivel de proceso y almacenamiento

optimizado para costos bajos.

5. **Sección "Proceso + almacenamiento":** Haz clic en "Configurar servidor".[24]
    - **Nivel de proceso: Selecciona el nivel "Burstable" (ej., B1MS o B2S)**.[28] Estos son los niveles más económicos y adecuados para cargas de trabajo de desarrollo intermitentes.
    - **Tamaño de almacenamiento:** Elige el tamaño mínimo requerido (ej., 20 GB).[24]
    - **Crecimiento automático del almacenamiento:** Habilítalo.[27]
    - Haz clic en "Guardar".[24]
6. **Sección "Autenticación":**
    - **Método de autenticación:** Elige "Solo autenticación de PostgreSQL" para simplificar en desarrollo.[24]
    - **Nombre de usuario de administrador:** Introduce un nombre de usuario (ej., devadmin).[24]
    - **Contraseña:** Configura una contraseña segura y confírmala.[24]
7. **Pestaña "Redes":**
    - **Método de conectividad:** Selecciona **"Acceso público (direcciones IP permitidas)"** .
    - **Reglas de firewall:**
        - Para permitir la conexión desde tu IP actual (para administración), haz clic en "Agregar dirección IP de cliente actual" .
        - **Para la comunicación con AWS y Local en desarrollo:** Deberás agregar las IPs públicas específicas de tus servicios de desarrollo en AWS y de tu red local. **Evita "Permitir acceso público desde cualquier servicio de Azure dentro de Azure a este servidor" (0.0.0.0/0) en desarrollo** .
        - Asegúrate de que el puerto TCP 5432 esté abierto.[28]
8. **Haz clic en "Revisar + crear" y luego en "Crear"**.[24]
9. **Después de la creación:** Copia el "Punto de conexión" (host) y el "Inicio de sesión de administrador".[25]

- **Utiliza la función de "Start/Stop"** para detener el servidor cuando no esté en uso y minimizar los costos.[28]

2. **Configuración de Conectividad de Red Segura (Enfoque de Desarrollo):**
    - **Para desarrollo, la configuración de VPN Site-to-Site (IPsec) entre AWS, Azure y Local puede ser excesiva y costosa.**
    - **Alternativa de Desarrollo:** Para pruebas de integración entre nubes y local, puedes optar por **acceso público con reglas de firewall muy restrictivas**

(solo IPs específicas de tus entornos de desarrollo) [7],.
- **Importante:** Esta configuración es **solo para desarrollo**. Para producción, las VPNs Site-to-Site o conexiones dedicadas (AWS Direct Connect, Azure ExpressRoute) son indispensables para la seguridad y el rendimiento.
- **SSL/TLS:** Asegúrate de que todas las conexiones entre bases de datos utilicen SSL/TLS para cifrar los datos en tránsito, incluso en desarrollo.[3]

Fase 3: Configuración de Foreign Data Wrappers (FDW)

La configuración de FDWs se realiza directamente en tus instancias de PostgreSQL. Para desarrollo, los host en las opciones de CREATE SERVER apuntarán a tus instancias de desarrollo (locales o en la nube con sus IPs públicas/privadas de desarrollo).

1. **Habilitar la Extensión postgres_fw:**
   SQL
   ```
   CREATE EXTENSION postgres_fdw;
   ```
   Verifica: SELECT * FROM pg_extension WHERE extname = 'postgres_fdw';.[1]
2. Crear un Servidor Foráneo (FOREIGN SERVER):
   Define un servidor foráneo para cada base de datos remota (AWS, Azure, Local).
   SQL
   ```
   -- Ejemplo en la DB local (Medical Records) para consultar Cardiología (AWS Dev)
   CREATE SERVER foreign_server_aws_cardiology_dev
   FOREIGN DATA WRAPPER postgres_fdw
   OPTIONS (host 'tu_host_aws_rds_cardiology_dev.rds.amazonaws.com', port '5432', dbname 'cardiology_db_dev');

   -- Ejemplo en la DB de Cardiología (AWS Dev) para consultar Farmacia (Azure Dev)
   CREATE SERVER foreign_server_azure_pharmacy_dev
   FOREIGN DATA WRAPPER postgres_fdw
   OPTIONS (host 'tu_host_azure_db_pharmacy_dev.postgres.database.azure.com', port '5432', dbname 'pharmacy_db_dev');

   -- Ejemplo en la DB de Farmacia (Azure Dev) para consultar Historias Clínicas (Local Dev)
   CREATE SERVER foreign_server_local_medical_records_dev
   FOREIGN DATA WRAPPER postgres_fdw
   OPTIONS (host 'tu_ip_local_medical_records_dev', port '5432', dbname
   ```

```
'medical_records_db_dev');
```

Ajusta host, port y dbname a tus entornos de desarrollo.[1]

3. Crear un Mapeo de Usuario (USER MAPPING):
   Para cada servidor foráneo, especifica las credenciales del usuario remoto.
   ```
   SQL
   -- Para el usuario local que consulta la DB de Cardiología en AWS Dev
   CREATE USER MAPPING FOR local_user_querying_db_dev
   SERVER foreign_server_aws_cardiology_dev
   OPTIONS (user 'usuario_remoto_aws_cardiology_dev', password
   'contraseña_remota_aws_cardiology_dev');
   ```

   Asegúrate de que local_user_querying_db_dev sea el usuario que ejecuta las consultas FDW en la base de datos local. Las credenciales remotas deben tener el mínimo privilegio.[1]
4. Crear Tablas Foráneas (FOREIGN TABLE):
   Define las tablas foráneas en la base de datos local que apuntan a las tablas remotas. La estructura debe coincidir.1
   ```
   SQL
   -- Ejemplo en la DB local (Medical Records) para consultar Pacientes de Cardiología (AWS Dev)
   CREATE FOREIGN TABLE pacientes_cardiologia_dev (
       cod_pac INT NOT NULL,
       nom_pac TEXT,
       dir_pac TEXT,
       tel_pac TEXT,
       fecha_nac DATE
   )
   SERVER foreign_server_aws_cardiology_dev
   OPTIONS (schema_name 'public', table_name 'Pacientes');

   -- Ejemplo en la DB de Cardiología (AWS Dev) para consultar Medicamentos de Farmacia (Azure
   Dev)
   CREATE FOREIGN TABLE medicamentos_farmacia_dev (
       cod_med INT NOT NULL,
       nom_med TEXT,
       descrip TEXT,
       stock INT
   )
   SERVER foreign_server_azure_pharmacy_dev
   OPTIONS (schema_name 'public', table_name 'Medicamentos');
   ```

Ahora puedes consultar pacientes_cardiologia_dev o medicamentos_farmacia_dev como si fueran tablas locales en la base de datos donde las definiste.[4]

5. **Optimización y Seguridad de FDWs:**
    - **Rendimiento:** Utiliza LIMIT y WHERE para reducir datos transferidos, ajusta FETCH_SIZE, y considera vistas materializadas o cachés locales para datos frecuentemente accedidos.[3]
    - **Seguridad:** Mínimo privilegio para usuarios remotos, gestión segura de credenciales (ej., AWS Secrets Manager, Azure Key Vault), y monitoreo de logs.[3]

Fase 4: Integración del Message Broker (QueueT)

1. **Despliegue de Redis:**
    - **Desarrollo Local:** Instala y ejecuta una instancia de Redis localmente. Esta es la opción más sencilla y económica para el desarrollo.
    - **Producción:** Utiliza un servicio gestionado de Redis en la nube (ej., Amazon ElastiCache for Redis, Azure Cache for Redis).
2. **Instalación de QueueT:**
    - En cada microservicio que necesite publicar o suscribirse a eventos:
      Bash
      ```
      npm install @tradologics/queuet
      ```
      Asegúrate de que el microservicio pueda conectarse a tu instancia de Redis.
3. **Configuración de QueueT en NestJS:**
    - Crea un módulo o servicio NestJS para encapsular la lógica de QueueT (conexión, publicación, suscripción).
    - apps/your-service/src/events/event.publisher.ts (ejemplo):
      TypeScript
      ```
      import { Injectable, OnModuleInit, OnModuleDestroy } from '@nestjs/common';
      import * as QueueT from '@tradologics/queuet';

      @Injectable()
      export class EventPublisherService implements OnModuleInit, OnModuleDestroy {
        private readonly queueName = 'hospital_events'; // Nombre de la cola/canal
      ```

```
  async onModuleInit() {
    // Inicializa la conexión a Redis para QueueT
    // Asegúrate de que Redis esté corriendo y accesible (ej. localhost:6379 para desarrollo)
    QueueT.initialize(this.queueName, { /* opciones de Redis si es necesario */ });
    console.log(`QueueT Publisher initialized for queue: ${this.queueName}`);
  }

  async publishEvent(eventType: string, payload: any): Promise<void> {
    const message = { eventType, payload, timestamp: new Date().toISOString() };
    await QueueT.publish(JSON.stringify(message));
    console.log(`Published event: ${eventType}`);
  }

  async onModuleDestroy() {
    QueueT.disconnect();
    console.log('QueueT Publisher disconnected.');
  }
}
```

- apps/your-service/src/events/event.subscriber.ts (ejemplo):

```
TypeScript
import { Injectable, OnModuleInit, OnModuleDestroy } from '@nestjs/common';
import * as QueueT from '@tradologics/queuet';

@Injectable()
export class EventSubscriberService implements OnModuleInit, OnModuleDestroy {
  private readonly queueName = 'hospital_events'; // Debe coincidir con el publisher

  async onModuleInit() {
    QueueT.initialize(this.queueName, { /* opciones de Redis */ });
    console.log(`QueueT Subscriber initialized for queue: ${this.queueName}`);

    // Suscribirse a nuevos mensajes
    QueueT.subscribe(this.handleMessage.bind(this));
    // Opcional: Leer el backlog al iniciar
    // QueueT.read_backlog(this.handleMessage.bind(this));
  }

  private async handleMessage(id: string, data: string) {
```

```
    try {
      const message = JSON.parse(data);
      console.log(`Received event: ${message.eventType} with payload:`,
message.payload);

      // Aquí se implementa la lógica para manejar el evento específico
      switch (message.eventType) {
        case 'PrescripcionCreada':
          // Llama a un servicio para procesar la prescripción y actualizar stock
          // await this.pharmacyService.updateStock(message.payload.medicamentoId,
message.payload.cantidad);
          console.log(`Processing PrescripcionCreada for medicamento
${message.payload.cod_med}`);
          break;
        // Otros tipos de eventos
        case 'PacienteRegistrado':
          console.log(`Processing PacienteRegistrado for paciente
${message.payload.cod_pac}`);
          break;
        default:
          console.warn(`Unhandled event type: ${message.eventType}`);
      }

      // Eliminar el mensaje de la cola una vez procesado
      await QueueT.delete(id);
    } catch (error) {
      console.error(`Error processing message ${id}:`, error);
      // Manejo de errores: reintentar, mover a cola de mensajes fallidos (DLQ)
    }
  }

  async onModuleDestroy() {
    QueueT.disconnect();
    console.log('QueueT Subscriber disconnected.');
  }
}
```

- Importa y provee estos servicios en los módulos de NestJS que los necesiten.

Fase 5: Implementación de Lógica de Negocio con Eventos y FDWs

1. **Ejemplo: Prescripción de Medicamentos (Operación de Escritura Distribuida - Saga):**
   - **CardiologyService (o servicio departamental):**
     - Recibe una solicitud de prescripción.
     - Usa Prisma para crear el registro de Prescripciones en su base de datos local.
     - Inyecta EventPublisherService y publica un evento PrescripcionCreada con los detalles relevantes (ej., cod_hist, cod_med, dosis, instrucciones).

```TypeScript
// En CardiologyService
async createPrescription(data: any) {
  const prescription = await this.prisma.prescripciones.create({ data });
  await this.eventPublisherService.publishEvent('PrescripcionCreada', {
    cod_hist: prescription.cod_hist,
    cod_med: data.cod_med,
    dosis: data.dosis,
    instrucciones: data.instrucciones,
    //... otros datos necesarios para el Servicio de Farmacia
  });
  return prescription;
}
```

   - **PharmacyService:**
     - Inyecta EventSubscriberService y tiene un método para manejar el evento PrescripcionCreada.
     - Cuando recibe el evento, usa Prisma para actualizar el stock del Medicamento en su base de datos central.

```TypeScript
// En PharmacyService (o un handler de eventos dentro de él)
async handlePrescripcionCreada(payload: any) {
  // Lógica para decrementar el stock
  await this.prisma.medicamentos.update({
    where: { cod_med: payload.cod_med },
    data: { stock: { decrement: 1 } }, // O la cantidad prescrita
  });
```

```
  // Si hay errores, publicar un evento de compensación si es necesario
}
```

2. **Ejemplo: Acceso Transversal a Historias Clínicas (Operación de Lectura - FDW):**
   - **MedicalRecordsService:** Es el propietario de la tabla Historias_Clinicas en su base de datos local.
   - **CardiologyService (o cualquier otro servicio departamental):**
     - No necesita duplicar los datos de Historias_Clinicas.
     - En su base de datos PostgreSQL, se habrá configurado una FOREIGN TABLE que apunta a la tabla Historias_Clinicas del MedicalRecordsService.[4]
     - El CardiologyService puede realizar consultas directamente a esta tabla foránea a través de Prisma (si se mapea el modelo) o SQL crudo.[13]

```
TypeScript
// En CardiologyService
async getPatientMedicalHistory(patientCod: number) {
  // Asumiendo que 'historias_clinicas_central' es la FOREIGN TABLE
  return this.prisma.$queryRaw`SELECT * FROM historias_clinicas_central WHERE cod_pac = ${patientCod}`;
}
```

     - Las modificaciones a Historias_Clinicas siempre deben pasar por la API del MedicalRecordsService, que a su vez podría publicar un evento HistoriaClinicaActualizada para notificar a otros servicios.

Fase 6: Implementación de Seguridad y Funcionalidades Adicionales

1. **Manejo de Contraseñas:**
   - En el AuthService, al registrar un usuario, hashea la contraseña con bcryptjs.hash() .
   - Al iniciar sesión, usa bcryptjs.compare() para verificar la contraseña .
   - Implementa el flujo de restablecimiento de contraseña (generación de token, envío de email, verificación, actualización) .
2. **Autenticación y Autorización (JWT y RBAC):**
   - El AuthService emitirá JWTs al iniciar sesión .
   - Crea *guards* (NestJS) para proteger rutas y verificar JWTs .

    - Implementa lógica de RBAC en los *guards* o *interceptors* para verificar los roles del usuario antes de permitir el acceso a ciertas funcionalidades o datos.[12]
3. **API Gateway:**
    - Configura el api-gateway (también un microservicio NestJS) para enrutar solicitudes a los servicios internos.[12]
    - Puede manejar la autenticación inicial y la validación de JWTs, reduciendo la carga en los microservicios individuales.[12]

V. Composición del Espacio de Trabajo y Estructura de Archivos/Carpetas

La estructura de monorepo con npm workspaces es clave para la organización y la reutilización de código .

```
hospital-sgh/
├── apps/
│   ├── auth-service/           # Microservicio de Autenticación y Autorización
│   │   ├── src/
│   │   │   ├── auth/           # Módulo de autenticación (controladores, servicios, estrategias JWT)
│   │   │   ├── users/          # Módulo de usuarios (modelos, servicios, controladores)
│   │   │   ├── prisma/         # PrismaService y configuración
│   │   │   ├── config/         # Configuraciones específicas (ej. JWT, bcrypt)
│   │   │   ├── main.ts         # Punto de entrada de la aplicación
│   │   │   └── app.module.ts   # Módulo raíz
│   │   ├── prisma/             # Esquema Prisma para User, PasswordResetToken
│   │   ├──.env                 # Variables de entorno del servicio (apunta a DB local en dev)
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
```

```
│   ├── cardiology-service/      # Microservicio de Cardiología
│   │   ├── src/
│   │   │   ├── patients/        # Módulo de pacientes (modelos, servicios, controladores)
│   │   │   ├── employees/       # Módulo de empleados (departamentales)
│   │   │   ├── appointments/    # Módulo de citas (departamentales)
│   │   │   ├── prescriptions/   # Módulo de prescripciones
│   │   │   ├── prisma/
│   │   │   ├── events/          # Publicadores/Suscriptores de eventos (ej. PrescripcionCreada)
│   │   │   ├── config/
│   │   │   ├── main.ts
│   │   │   └── app.module.ts
│   │   ├── prisma/              # Esquema Prisma para Pacientes, Empleados, Citas, Prescripciones
│   │   ├──.env
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   ├── pharmacy-service/        # Microservicio de Farmacia
│   │   ├── src/
│   │   │   ├── medicines/       # Módulo de medicamentos (inventario centralizado)
│   │   │   ├── events/          # Suscriptor de eventos (ej. PrescripcionCreada)
│   │   │   ├── prisma/
│   │   │   ├── config/
│   │   │   ├── main.ts
│   │   │   └── app.module.ts
│   │   ├── prisma/              # Esquema Prisma para Medicamentos
│   │   ├──.env
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   ├── medical-records-service/  # Microservicio de Historias Clínicas
│   │   ├── src/
│   │   │   ├── medical-records/  # Módulo de historias clínicas
│   │   │   ├── events/          # Publicador de eventos (ej. HistoriaClinicaActualizada)
│   │   │   ├── prisma/
│   │   │   ├── config/
│   │   │   ├── main.ts
```

```
│   │   │   └── app.module.ts
│   │   ├── prisma/             # Esquema Prisma para Historias_Clinicas
│   │   ├──.env
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   ├── appointment-service/      # Microservicio de Agendas de Citas
│   │   #... estructura similar
│   │
│   └── api-gateway/            # API Gateway
│       ├── src/
│       │   ├── auth/           # Lógica de autenticación/validación JWT
│       │   ├── proxy/          # Enrutamiento a microservicios
│       │   ├── main.ts
│       │   └── app.module.ts
│       ├──.env
│       ├── package.json
│       └── tsconfig.json
│
├── packages/
│   ├── common-types/           # Tipos e interfaces compartidas (DTOs, Eventos)
│   │   ├── src/
│   │   │   ├── dtos/
│   │   │   ├── interfaces/
│   │   │   └── events/         # Definiciones de eventos (ej.
PrescripcionCreadaEventPayload)
│   │   ├── package.json
│   │   └── tsconfig.json
│   │
│   └── shared-utils/           # Utilidades compartidas (ej. password-hasher.ts)
│       ├── src/
│       │   └── security/
│       │       └── password.util.ts
│       ├── package.json
│       └── tsconfig.json
│
├── scripts/                  # Scripts de CI/CD, despliegue, etc.
├──.env.example               # Variables de entorno de ejemplo para el monorepo
├── package.json               # package.json raíz del monorepo
```

```
├── tsconfig.json              # Configuración TypeScript global
└── README.md
```

Explicación de la Estructura:

- **apps/**: Contiene cada microservicio como un proyecto NestJS independiente. Cada uno tiene su propio package.json, tsconfig.json y .env para encapsular sus dependencias y configuraciones.[38]
- **packages/**: Contiene librerías o módulos de código que se comparten entre los microservicios. Esto evita la duplicación de código y asegura la consistencia de tipos .
    - common-types: Esencial para definir los contratos de datos (DTOs) y los esquemas de los eventos que se publicarán y consumirán a través del Message Broker.
    - shared-utils: Para funciones transversales como el hashing de contraseñas, que se usarán en el auth-service y potencialmente en otros servicios para validaciones.
- **src/events/ dentro de cada microservicio**: Aquí se ubicarán los servicios o clases responsables de interactuar con QueueT, ya sea publicando eventos (EventPublisherService) o suscribiéndose y manejándolos (EventSubscriberService).
- **prisma/ dentro de cada microservicio**: Contiene el schema.prisma específico para la base de datos de ese microservicio y las migraciones.

VII. Conclusiones y Próximos Pasos

La implementación de este SGH distribuido es un proyecto ambicioso que aprovecha las fortalezas de NestJS, Prisma, FDWs y QueueT para construir un sistema escalable, resiliente y seguro. La clave del éxito residirá en una planificación meticulosa, una implementación por fases y una atención constante a la seguridad y el rendimiento.

Recomendaciones Clave:

1. **Optimización de Costos en Desarrollo:**
    - **Prioriza el uso de bases de datos PostgreSQL locales** para la mayoría de las tareas de desarrollo.

    - Si necesitas instancias en la nube para pruebas de integración, utiliza los **niveles de servicio más económicos (Burstable/Development)** y **aprovecha las funciones de "Start/Stop"** para detener los recursos cuando no estén en uso.[28]
    - Para la conectividad en desarrollo, considera el acceso público con **reglas de firewall muy restrictivas** (solo IPs específicas) en lugar de VPNs complejas, pero **nunca uses esta configuración en producción** [7],.
2. **Validación Temprana:** Comienza con un MVP (Producto Mínimo Viable) que incluya un par de microservicios, sus bases de datos distribuidas (una en la nube, otra local), la conectividad de red, FDWs para lecturas y QueueT para una operación de escritura distribuida sencilla. Valida cada componente antes de escalar.
3. **Seguridad Primero:** La seguridad debe ser una prioridad desde el diseño hasta el despliegue. Asegura las conexiones de red, gestiona las credenciales de forma segura (utilizando servicios de gestión de secretos de AWS y Azure), y aplica el principio de mínimo privilegio en todas las capas [35], [39], [40], [6], [27], [7], [3], [27], [12], [20],,,, [7], [18],, [41], [23], [42], [34], [38], [33], [24].
4. **Monitoreo Continuo:** Implementa herramientas de monitoreo y *logging* centralizado para observar el rendimiento de los microservicios, la latencia de las consultas FDW y el flujo de eventos en QueueT. Esto te permitirá identificar y resolver problemas de forma proactiva.[39]
5. **Pruebas Exhaustivas:** Dada la complejidad de un sistema distribuido, las pruebas unitarias, de integración y de extremo a extremo son cruciales. Presta especial atención a las pruebas de rendimiento y resiliencia bajo carga.[38]
6. **Documentación:** Mantén una documentación clara y actualizada de la arquitectura, los esquemas de las bases de datos, las configuraciones de red, los contratos de eventos y los procedimientos de despliegue.[35]

Este plan te proporciona una hoja de ruta sólida para abordar tu proyecto, equilibrando la funcionalidad, la seguridad y la optimización de costos en el entorno de desarrollo. ¡Mucho éxito en su implementación!

Obras citadas

1. A Guide to Cross-Database Queries: PostgreSQL FDW & DbLink ..., fecha de acceso: junio 17, 2025, https://www.coding-dude.com/wp/databases/postgres-fdw-dblink/
2. Stop Building Data Pipelines: Cross-Database Queries With ..., fecha de acceso: junio 17, 2025, https://www.timescale.com/blog/cross-database-queries-with-postgresql-foreign-data-wrappers

3. A Guide to SetUp a Foreign Server in Amazon RDS for PostgreSQL - CloudThat, fecha de acceso: junio 17, 2025, https://www.cloudthat.com/resources/blog/a-guide-to-setup-a-foreign-server-in-amazon-rds-for-postgresql
4. F.36. postgres_fdw — access data stored in external PostgreSQL servers, fecha de acceso: junio 17, 2025, https://www.postgresql.org/docs/current/postgres-fdw.html
5. Microservices Database Design Patterns - GeeksforGeeks, fecha de acceso: junio 17, 2025, https://www.geeksforgeeks.org/sql/microservices-database-design-patterns/
6. Microservices Database Design Patterns - GeeksforGeeks, fecha de acceso: junio 17, 2025, https://www.geeksforgeeks.org/microservices-database-design-patterns/
7. Overview of security best practices for Amazon RDS for PostgreSQL ..., fecha de acceso: junio 17, 2025, https://aws.amazon.com/blogs/database/overview-of-security-best-practices-for-amazon-rds-for-postgresql-and-amazon-aurora-postgresql-compatible-edition/
8. Managing Multiple Packages with npm Workspaces: A Complete Guide - GeekyAnts, fecha de acceso: junio 17, 2025, https://geekyants.com/blog/managing-multiple-packages-with-npm-workspaces-a-complete-guide
9. Tutorial: How to Create Your First NestJS Project - CBT Nuggets, fecha de acceso: junio 17, 2025, https://www.cbtnuggets.com/tutorial-create-first-nestjs-project
10. Fantastic 13 Node Microservice Framework In 2024 - ThemeSelection, fecha de acceso: junio 17, 2025, https://themeselection.com/node-microservice-framework/
11. Performance Tips for Postgres FDW | Crunchy Data Blog, fecha de acceso: junio 17, 2025, https://www.crunchydata.com/blog/performance-tips-for-postgres-fdw
12. Authentication and authorization in a microservice architecture: Part 1 - Introduction, fecha de acceso: junio 17, 2025, https://microservices.io/post/architecture/2025/04/25/microservices-authn-authz-part-1-introduction.html
13. 10 Node.js Microservices Best Practices 2024 - Daily.dev, fecha de acceso: junio 17, 2025, https://daily.dev/blog/10-nodejs-microservices-best-practices-2024
14. Distributed Data in PostgreSQL with postgres_fdw: A Guide to Enhanced Performance and Flexibility - Stormatics, fecha de acceso: junio 17, 2025, https://stormatics.tech/blogs/distributed-data-in-postgresql-with-postgres_fdw
15. Node.js project architecture best practices - LogRocket Blog, fecha de acceso: junio 17, 2025, https://blog.logrocket.com/node-js-project-architecture-best-practices/
16. What are TypeScript Microservices? - Capicua, fecha de acceso: junio 17, 2025, https://www.capicua.com/blog/typescript-microservices
17. How To Install And Setup First NestJS Application? - GeeksforGeeks, fecha de acceso: junio 17, 2025,

https://www.geeksforgeeks.org/javascript/how-to-install-and-setup-first-nestjs-application/
18. Password hashing in Node.js with bcrypt - Honeybadger Developer Blog, fecha de acceso: junio 17, 2025, https://www.honeybadger.io/blog/node-password-hashing/
19. Managing distributed data in a microservice architecture - Eventuate, fecha de acceso: junio 17, 2025, https://eventuate.io/docs/manual/eventuate-tram/latest/distributed-data-management.html
20. Mastering Microservices Authorization: Strategies for Secure Access Control - KrakenD, fecha de acceso: junio 17, 2025, https://www.krakend.io/blog/microservices-authorization-secure-access/
21. Prisma | NestJS - A progressive Node.js framework, fecha de acceso: junio 17, 2025, https://docs.nestjs.com/recipes/prisma
22. An Overview of Distributed PostgreSQL Architectures | Crunchy Data Blog, fecha de acceso: junio 17, 2025, https://www.crunchydata.com/blog/an-overview-of-distributed-postgresql-architectures
23. Cloud Networking Across AWS, Azure and GCP - Alkira, fecha de acceso: junio 17, 2025, https://www.alkira.com/cloud-networking-across-aws-azure-and-gcp/
24. Password Reset Flow in Nodejs - YouTube, fecha de acceso: junio 17, 2025, https://www.youtube.com/watch?v=lLVmH6SB2Z4
25. Querying Across Databases in a PostgreSQL Microservices Architecture: Looking for Advice - Reddit, fecha de acceso: junio 17, 2025, https://www.reddit.com/r/PostgreSQL/comments/1esb24r/querying_across_databases_in_a_postgresql/
26. Before you can create a PostgreSQL database, you need to set up a flexible server to run it. You can use the Microsoft Azure portal to create and configure an Azure PostgreSQL flexible server to use with your postgreSQL database. - Genetec TechDoc Hub, fecha de acceso: junio 17, 2025, https://techdocs.genetec.com/r/en-US/GenetecTM-Data-Exporter-Plugin-Guide-1.0.0/Creating-an-Azure-PostgreSQL-flexible-server?contentId=~J1A7EDVX0InsqC00sBj9A
27. PostgreSQL best practices | Trend Micro, fecha de acceso: junio 17, 2025, https://www.trendmicro.com/cloudoneconformity/knowledge-base/azure/PostgreSQL/
28. Architecture Best Practices for Azure Database for PostgreSQL ..., fecha de acceso: junio 17, 2025, https://learn.microsoft.com/en-us/azure/well-architected/service-guides/postgresql
29. Quickstart: Create an Azure Database for PostgreSQL flexible server - Learn Microsoft, fecha de acceso: junio 17, 2025, https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/quickstart-create-server
30. Next-generation ORM for Node.js & TypeScript - Prisma, fecha de acceso: junio

17, 2025, https://www.prisma.io/orm
31. Building Scalable Microservices with Node.js and Event-Driven Architecture, fecha de acceso: junio 17, 2025, https://dev.to/dhrumitdk/building-scalable-microservices-with-nodejs-and-event-driven-architecture-4ckc
32. Networking - Azure Database for PostgreSQL flexible server - Learn Microsoft, fecha de acceso: junio 17, 2025, https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/how-to-networking
33. How to easily set up a VPN between Azure and AWS using managed services (updated 2024) | Microsoft Community Hub, fecha de acceso: junio 17, 2025, https://techcommunity.microsoft.com/blog/startupsatmicrosoftblog/how-to-easily-set-up-a-vpn-between-azure-and-aws-using-managed-services-updated-/4278966
34. Password Encryption in Node.js using bcryptjs Module - GeeksforGeeks, fecha de acceso: junio 17, 2025, https://www.geeksforgeeks.org/node-js/password-encryption-in-node-js-using-bcryptjs-module/
35. Best Practices for Postgres Data Management - Timescale, fecha de acceso: junio 17, 2025, https://www.timescale.com/learn/postgres-data-management-best-practices
36. Sequelize | Feature-rich ORM for modern TypeScript & JavaScript, fecha de acceso: junio 17, 2025, https://sequelize.org/
37. Add firewall rules - Azure Database for PostgreSQL flexible server | Microsoft Learn, fecha de acceso: junio 17, 2025, https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/how-to-networking-servers-deployed-public-access-add-firewall-rules
38. Networking services compared: AWS vs Azure vs Google Cloud - Pluralsight, fecha de acceso: junio 17, 2025, https://www.pluralsight.com/resources/blog/cloud/networking-services-compared-aws-vs-azure-vs-google-cloud
39. AWS RDS PostgreSQL Hardening - eraki Blog, fecha de acceso: junio 17, 2025, https://eraki.hashnode.dev/aws-rds-hardening-and-best-practices
40. Choosing Distribution Column — Citus 13.0.1 documentation, fecha de acceso: junio 17, 2025, https://docs.citusdata.com/en/stable/sharding/data_modeling.html
41. Implementing a secure password reset in Node.js - LogRocket Blog, fecha de acceso: junio 17, 2025, https://blog.logrocket.com/implementing-secure-password-reset-node-js/
42. Understanding Azure to AWS Site-to-Site VPN Connections | Datasturdy Consulting, fecha de acceso: junio 17, 2025, https://datasturdy.com/understanding-azure-to-aws-site-to-site-vpn-connections/