Guía de Implementación: Microservicio de Asignación de Roles (role-assignment-service)

El role-assignment-service será un microservicio dedicado a orquestar la **especialización de usuarios** en roles específicos (MEDIC, PATIENT, ADMIN). Su base de datos **no almacenará los roles directamente**. En su lugar, este servicio actuará como un **orquestador de eventos**, verificando la existencia del usuario y publicando mensajes en QueueT para que los microservicios de dominio correspondientes (ej., CardiologyService para Empleado/Medico/P.Admin, Azure:Paciente para Paciente) creen las entradas especializadas.

I. Visión General del Flujo de Asignación de Roles (Actualizado)

Este flujo refleja la nueva lógica donde el rol se infiere por la existencia del idUser en las tablas de especialización de los microservicios de dominio.

1. **Creación de Usuario (Auth Service):**
    - El auth-service es el propietario de los datos básicos del usuario (cédula, nombre, email, etc.) y los persiste en su base de datos Usuario (local).
    - Luego, el auth-service (específicamente el endpoint POST /auth/root/create-user/:role) realiza una **llamada HTTP** al role-assignment-service con el idUser del usuario recién creado y el role deseado (ej., MEDIC, ADMIN, PATIENT) en la URL.
2. **Orquestación de Especialización (Role Assignment Service):**
    - El role-assignment-service recibe la solicitud HTTP del auth-service con el idUser y el role a especializar.
    - **Validación de Usuario (DBLINK):** Utiliza DBLINK para verificar que el idUser exista en la base de datos Usuario del auth-service. Esto es crucial para asegurar que solo se intente especializar a usuarios válidos.
    - **Publicación de Evento (QueueT):** Si el usuario existe, el role-assignment-service **publica un evento específico en QueueT** basado en el rol solicitado. Este evento contendrá el idUser y cualquier otro dato relevante para la especialización.

- **No hay persistencia de roles en la base de datos del role-assignment-service.** Su base de datos (si la tuviera) sería solo para fines internos del servicio (ej., logs, estado de procesamiento de sagas complejas, pero no para la información de roles en sí).

3. **Reacción de Microservicios de Dominio (QueueT):**
   - **CardiologyService (para MEDIC/ADMIN):** Se suscribe a eventos como DoctorSpecializationRequested o AdminSpecializationRequested. Al recibirlo, crea una entrada en la tabla Empleado de su base de datos Cardiologia. Si el rol es MEDIC, también crea una entrada en la tabla Medico. Si es ADMIN, crea una entrada en P.Admin.
   - **Azure:Paciente Service (para PATIENT):** Se suscribe a PatientSpecializationRequested. Al recibirlo, crea una entrada en la tabla Paciente de su base de datos en Azure.

II. Refactorización del Microservicio de Asignación de Roles (role-assignment-service)

A. Base de Datos y Prisma Setup (Actualizado y Simplificado)

1. **Elección de la Base de Datos:**
   - El role-assignment-service **ya no necesitará una base de datos propia para almacenar roles**. Su función es puramente de orquestación y publicación de eventos.
   - **Por lo tanto, este servicio NO tendrá un schema.prisma con modelos de negocio, ni una base de datos asociada a Prisma.** Su interacción con bases de datos se limitará a DBLINK para validación externa.
2. Configuración de .env:
   El archivo apps/role-assignment-service/.env contendrá la configuración de Redis y los detalles de DBLINK para la base de datos Usuario del auth-service.
   Fragmento de código
   ```
   # apps/role-assignment-service/.env (para desarrollo)
   PORT=3002
   REDIS_HOST=localhost
   ```

```
REDIS_PORT=6379
QUEUE_NAME=hospital_events

# Configuración para DBLINK al servicio de autenticación (Usuario DB)
AUTH_DB_DBLINK_HOST=localhost
AUTH_DB_DBLINK_PORT=5432
AUTH_DB_DBLINK_DBNAME=usuarios
AUTH_DB_DBLINK_USER=wally
AUTH_DB_DBLINK_PASSWORD=wally
```

Asegúrate de reemplazar wally y usuarios con los detalles de tu base de datos Usuario del auth-service.

3. **Eliminación del Schema Prisma y Migraciones:**
   - **Elimina el archivo apps/role-assignment-service/prisma/schema.prisma**.
   - **Elimina la carpeta apps/role-assignment-service/prisma/migrations**.
   - **Desinstala Prisma CLI y @prisma/client** de este servicio:
     ```
     Bash
     cd apps/role-assignment-service
     npm uninstall prisma @prisma/client
     ```
   - **Elimina apps/role-assignment-service/src/prisma.service.ts**.

B. Configuración de DBLINK (Actualizado)

Dado que el role-assignment-service ya no usará Prisma para su propia base de datos, la ejecución de consultas DBLINK se realizará a través de un cliente PostgreSQL (pg) que se conectará directamente a la base de datos Usuario del auth-service para ejecutar la función dblink.

1. **Instalar pg (cliente PostgreSQL) en role-assignment-service:**
   ```
   Bash
   cd apps/role-assignment-service
   npm install pg
   npm install --save-dev @types/pg
   ```

2. Crear un DblinkService:
   Este servicio encapsulará la lógica de conexión y ejecución de DBLINK.

apps/role-assignment-service/src/dblink/dblink.service.ts

TypeScript

```
import { Injectable, InternalServerErrorException, Logger, OnModuleDestroy } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { Pool } from 'pg'; // Importa el cliente pg

@Injectable()
export class DblinkService implements OnModuleDestroy {
  private readonly logger = new Logger(DblinkService.name);
  private readonly authDbPool: Pool; // Pool de conexiones a la DB del auth-service

  constructor(private configService: ConfigService) {
    // Configura el pool de conexiones a la base de datos del auth-service
    this.authDbPool = new Pool({
      host: this.configService.get<string>('AUTH_DB_DBLINK_HOST'),
      port: this.configService.get<number>('AUTH_DB_DBLINK_PORT'),
      database: this.configService.get<string>('AUTH_DB_DBLINK_DBNAME'),
      user: this.configService.get<string>('AUTH_DB_DBLINK_USER'),
      password: this.configService.get<string>('AUTH_DB_DBLINK_PASSWORD'),
    });

    // Asegura que la extensión dblink esté habilitada en la DB del auth-service
    // Esto es una operación de una sola vez, pero se verifica al inicio del servicio.
    this.authDbPool.query('CREATE EXTENSION IF NOT EXISTS dblink;')
    .then(() => this.logger.log('DBLINK extension ensured in Auth Service DB.'))
    .catch(err => this.logger.error(`Failed to ensure DBLINK extension: ${err.message}`));
  }

  /**
   * Verifica si un usuario existe en la base de datos del Auth Service (Usuario DB) usando DBLINK.
   * Se conecta a la DB del Auth Service y ejecuta la consulta DBLINK desde allí.
   * @param userId El ID del usuario (cédula).
   * @returns true si el usuario existe, false en caso contrario.
   */
  async userExistsInAuthDb(userId: number): Promise<boolean> {
    try {
      // La consulta dblink se ejecuta en la DB a la que este servicio está conectado (auth-service DB)
      // y consulta a sí misma para verificar la existencia del usuario.
```

```
      // Esto es necesario porque el role-assignment-service no tiene DB propia para ejecutar dblink.
      const query = `
       SELECT EXISTS (
         SELECT 1 FROM dblink('host=${this.configService.get<string>('AUTH_DB_DBLINK_HOST')} port=${this.configService.get<number>('AUTH_DB_DBLINK_PORT')} dbname=${this.configService.get<string>('AUTH_DB_DBLINK_DBNAME')} user=${this.configService.get<string>('AUTH_DB_DBLINK_USER')} password=${this.configService.get<string>('AUTH_DB_DBLINK_PASSWORD')}',
           'SELECT "idUser" FROM "User" WHERE "idUser" = ${userId}') AS t("idUser" INT)
       );
      `;
      const res = await this.authDbPool.query(query);
      return res.rows.exists; // Accede a la propiedad 'exists' del primer resultado
    } catch (error) {
      this.logger.error(`DBLINK error checking user ${userId} in Auth DB: ${error.message}`);
      throw new InternalServerErrorException('Failed to verify user existence in authentication database.');
    }
  }

  async onModuleDestroy() {
    await this.authDbPool.end(); // Cierra el pool de conexiones al destruir el módulo
    this.logger.log('DBLINK service pool disconnected.');
  }
}
```

C. Desarrollo de la Lógica del Servicio (src/roles/roles.service.ts) (Actualizado)

Este servicio ya no interactuará con una tabla UserRoleAssignment. Su función será validar la existencia del usuario y publicar eventos de especialización.

1. **Crear el archivo del servicio:**
   Bash
   ```
   nest g service roles --no-spec
   ```

2. **Implementación de RolesService:**
   apps/role-assignment-service/src/roles/roles.service.ts

```TypeScript
import { Injectable, BadRequestException, NotFoundException, Logger } from '@nestjs/common';
import { UserRole } from '../../../packages/common-types/src/index.js'; // Importa el enum UserRole
import { EventPublisherService } from '../events/event.publisher.js'; // Importa el publicador
import { DblinkService } from '../dblink/dblink.service.js'; // Importa el nuevo DblinkService
import { UserSpecializationRequestedEventPayload } from '../../../packages/common-types/src/events/user-events.js'; // Importa el payload del evento

@Injectable()
export class RolesService {
  private readonly logger = new Logger(RolesService.name);

  constructor(
    private dblinkService: DblinkService, // Inyecta DblinkService
    private eventPublisherService: EventPublisherService, // Inyecta el publicador
  ) {}

  /**
   * Orquesta la especialización de un usuario en un rol específico.
   * Publica un evento a QueueT para que el microservicio de dominio correspondiente cree la entrada especializada.
   * @param userId El ID del usuario (cédula).
   * @param role El rol a especializar (PATIENT, DOCTOR, ADMIN).
   * @returns Un mensaje de éxito.
   */
  async specializeUserRole(userId: number, role: UserRole): Promise<string> {
    // 1. Validar que el usuario exista en la base de datos del Auth Service
    const userExists = await this.dblinkService.userExistsInAuthDb(userId);
    if (!userExists) {
      throw new NotFoundException(`User with ID ${userId} not found in the authentication system.`);
    }

    // 2. Validar reglas de negocio de roles (ej. ROOT no puede ser asignado por este servicio)
    // Las reglas de negocio sobre combinaciones de roles (ADMIN no puede ser DOCTOR)
    // se aplicarán en los servicios de dominio al crear la especialización,
    // o en el auth-service al generar el JWT.
    if (role === UserRole.ROOT) {
      throw new BadRequestException('ROOT role cannot be specialized via this service.');
```

```
    }

    // 3. Publicar evento específico en QueueT para el microservicio de dominio
    let eventType: string;
    let payload: UserSpecializationRequestedEventPayload;

    // El payload puede incluir otros datos si el role-assignment-service los conociera
    // y fueran útiles para el servicio consumidor.
    payload = { userId, role, timestamp: new Date().toISOString() };

    switch (role) {
      case UserRole.DOCTOR:
        eventType = 'DoctorSpecializationRequested';
        break;
      case UserRole.ADMIN:
        eventType = 'AdminSpecializationRequested';
        break;
      case UserRole.PATIENT:
        eventType = 'PatientSpecializationRequested';
        break;
      default:
        throw new BadRequestException(`Unsupported role for specialization: ${role}`);
    }

    await this.eventPublisherService.publishEvent(eventType, payload);

    this.logger.log(`Specialization event '${eventType}' published for user ${userId} with role
${role}.`);
    return `Specialization for user ${userId} as ${role} requested.`;
  }
}
```

D. Desarrollo del Controlador (src/roles/roles.controller.ts) (Actualizado)

Este controlador expondrá el *endpoint* HTTP para la especialización de roles.

1. **Crear el archivo del controlador:**
   Bash
   ```
   nest g controller roles --no-spec
   ```

2. **Implementación de RolesController:**
   apps/role-assignment-service/src/roles/roles.controller.ts
   TypeScript
   ```
   import { Controller, Post, Body, UsePipes, ValidationPipe, HttpCode, HttpStatus, Param, ParseIntPipe, BadRequestException } from '@nestjs/common';
   import { RolesService } from './roles.service.js';
   import { ApiTags, ApiOperation, ApiResponse, ApiParam } from '@nestjs/swagger';
   import { UserRole } from '../../../packages/common-types/src/index.js'; // Importa el enum UserRole

   @ApiTags('roles')
   @Controller('roles')
   export class RolesController {
     constructor(private rolesService: RolesService) {}

     @ApiOperation({ summary: 'Solicitar especialización de rol para un usuario' })
     @ApiResponse({ status: 200, description: 'Solicitud de especialización de rol enviada.' })
     @ApiResponse({ status: 400, description: 'Solicitud inválida o rol no soportado.' })
     @ApiResponse({ status: 404, description: 'Usuario no encontrado en el sistema de autenticación.' })
     @ApiParam({ name: 'idUser', type: Number, description: 'ID del usuario (cédula)' })
     @ApiParam({ name: 'role', enum: UserRole, description: 'Rol a especializar (PATIENT, DOCTOR, ADMIN)' })
     @Post(':idUser/:role')
     @HttpCode(HttpStatus.OK)
     @UsePipes(new ValidationPipe({ transform: true }))
     async specializeRole(
       @Param('idUser', ParseIntPipe) idUser: number,
       @Param('role') role: UserRole, // NestJS puede parsear enums de URL
     ): Promise<{ message: string }> {
       // Validar que el rol sea uno de los roles de especialización permitidos por este endpoint
       const specializationRoles: UserRole =;
       if (!specializationRoles.includes(role)) {
         throw new BadRequestException(`Role '${role}' cannot be specialized via this endpoint.`);
   ```

```
  }

  const message = await this.rolesService.specializ```
```typescript
eUserRole(idUser, role);
  return { message };
 }
}
```

E. Configuración de Módulos (Actualizado)

1. **Crear src/roles/roles.module.ts:**
   Bash
   ```
   nest g module roles --no-spec
   ```

2. **Implementación de RolesModule:**
   apps/role-assignment-service/src/roles/roles.module.ts
   TypeScript
   ```
   import { Module } from '@nestjs/common';
   import { RolesController } from './roles.controller.js';
   import { RolesService } from './roles.service.js';
   import { ConfigModule } from '@nestjs/config';
   import { EventPublisherService } from '../events/event.publisher.js';
   import { DblinkService } from '../dblink/dblink.service.js'; // Importa DblinkService

   @Module({
    imports: [ConfigModule], // Importa ConfigModule para usar ConfigService
    controllers:,
    providers:, // Provee todos los servicios
    exports:, // No exporta nada si solo es consumido internamente
   })
   export class RolesModule {}
   ```

3. Actualización de apps/role-assignment-service/src/app.module.ts:
   Asegúrate de que RolesModule y ConfigModule estén importados en el módulo raíz.
   ```

apps/role-assignment-service/src/app.module.ts

TypeScript

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { RolesModule } from './roles/roles.module.js';

@Module({
  imports:,
  controllers:, // Si tienes controladores en AppController, mantenlos
  providers:, // Si tienes servicios en AppService, mantenlos
})
export class AppModule {}
```

F. Configuración de main.ts (Sin cambios)

La configuración de main.ts para el role-assignment-service sigue siendo la misma.

apps/role-assignment-service/src/main.ts

TypeScript

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module.js';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { ValidationPipe } from '@nestjs/common';
import * as dotenv from 'dotenv';

dotenv.config();

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({ transform: true }));
```

```
  const config = new DocumentBuilder()
   .setTitle('Role Assignment Service API')
   .setDescription('API para la orquestación de especialización de roles de usuarios.')
   .setVersion('1.0')
   .addBearerAuth()
   .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

  await app.listen(process.env.PORT |

| 3002);
}
bootstrap().catch((error) => {
  console.error('Error starting the Role Assignment Service:', error);
  process.exit(1);
});
```

III. Integración con el API Gateway

El API Gateway (api-gateway en apps/api-gateway) actuará como el punto de entrada único para todas las solicitudes del frontend.

1. Actualizar apps/api-gateway/.env:
   Añade la URL del role-assignment-service.
   apps/api-gateway/.env
   Fragmento de código
   ```
   PORT=5000
   AUTH_SERVICE_URL=http://localhost:3001
   ROLES_SERVICE_URL=http://localhost:3002 # URL del role-assignment-service
   CORS_ORIGINS=http://localhost:3000 # URL de tu frontend React en desarrollo
   JWT_SECRET="tu_secreto_jwt_muy_seguro_y_largo" # Debe coincidir con el de auth-service
   ```

2. Actualizar apps/api-gateway/src/proxy/proxy.service.ts (Enrutamiento):
   Añade una regla de enrutamiento para las solicitudes dirigidas al

role-assignment-service.
apps/api-gateway/src/proxy/proxy.service.ts (fragmento)
TypeScript

```
import { Injectable, InternalServerErrorException } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class ProxyService {
  constructor(private configService: ConfigService) {}

  getTargetUrl(path: string): string {
    if (path.startsWith('/auth')) {
      return this.configService.get<string>('AUTH_SERVICE_URL');
    }
    if (path.startsWith('/roles')) { // NUEVA REGLA
      return this.configService.get<string>('ROLES_SERVICE_URL');
    }
    // Añade reglas para otros servicios aquí
    // if (path.startsWith('/history')) { return
this.configService.get<string>('HISTORY_SERVICE_URL'); }
    // if (path.startsWith('/appointments')) { return
this.configService.get<string>('APPOINTMENTS_SERVICE_URL'); }
    // if (path.startsWith('/equipment')) { return
this.configService.get<string>('EQUIPMENT_SERVICE_URL'); }
    // if (path.startsWith('/pharmacy')) { return
this.configService.get<string>('PHARMACY_SERVICE_URL'); }

    throw new InternalServerErrorException(`No target service found for path: ${path}`);
  }
}
```

Nota: El proxy.controller.ts de tu API Gateway ya debería estar usando proxy.service.ts para reenviar todas las solicitudes (@All('')).*

IV. Integración con el auth-service

El auth-service es el que inicia la especialización de roles después de crear un

usuario.

1. Actualizar apps/auth-service/.env:
   Asegúrate de que ROLES_SERVICE_URL apunte al nuevo role-assignment-service.
   apps/auth-service/.env
   Fragmento de código
   #... otras variables
   ROLES_SERVICE_URL="http://localhost:3002/roles" # URL base del servicio de roles
   # URLs de los servicios de dominio para obtener roles dinámicamente
   CARDIO_SERVICE_URL="http://localhost:3003" # Ejemplo: URL del CardiologyService
   PATIENT_SERVICE_URL="http://localhost:3004" # Ejemplo: URL del PatientService
   ROOT_USER_ID_DEV=123456789 # ID de cédula del usuario ROOT para desarrollo

2. Actualizar apps/auth-service/src/roles/roles.service.ts:
   Este servicio en el auth-service hará la llamada HTTP al role-assignment-service.
   apps/auth-service/src/roles/roles.service.ts
   TypeScript

```typescript
import { Injectable, Logger, InternalServerErrorException, BadRequestException }
from '@nestjs/common';
import axios from 'axios';
import { ConfigService } from '@nestjs/config';
import { UserRole } from '../../../packages/common-types/src/enums/user-roles.enum.js';

@Injectable()
export class RolesService {
  private readonly logger = new Logger(RolesService.name);
  private readonly rolesServiceBaseUrl: string; // Cambiado a base URL

  constructor(private configService: ConfigService) {
    this.rolesServiceBaseUrl = this.configService.get<string>(
      'ROLES_SERVICE_URL',
      'http://localhost:3002/roles', // Valor por defecto actualizado
    );
  }

  /**
   * Solicita al Role Assignment Service que especialice el rol de un usuario.
```

```
   * @param idUser El ID del usuario (cédula).
   * @param role El rol a especializar (PATIENT, DOCTOR, ADMIN).
   */
  async requestRoleSpecialization(idUser: number, role: UserRole) {
    try {
      // Validar la regla de negocio: ROOT no puede ser asignado por este endpoint
      if (role === UserRole.ROOT) {
        throw new BadRequestException('ROOT role cannot be assigned via this endpoint.');
      }

      const targetUrl = `${this.rolesServiceBaseUrl}/${idUser}/${role}`; // Construye la URL con
idUser y rol
      this.logger.log(`Calling Role Assignment Service at: ${targetUrl}`);

      await axios.post(targetUrl); // Envía la solicitud POST sin body, ya que los datos van en la
URL

      this.logger.log(
        `Requested specialization for user ${idUser} as ${role} via Role Assignment Service.`,
      );
    } catch (error) {
      this.logger.error(
        `Failed to request role specialization for user ${idUser} as ${role}: ${error.message}`,
      );
      if (axios.isAxiosError(error) && error.response) {
        throw new InternalServerErrorException(
          `Role Assignment Service error: ${JSON.stringify(error.response.data)}`,
        );
      }
      throw new InternalServerErrorException(
        'Could not request role specialization in Role Assignment Service.',
      );
    }
  }
}
```

3. Actualizar apps/auth-service/src/auth/auth.controller.ts:
   El endpoint POST /auth/root/create-user/:role ahora llamará a requestRoleSpecialization.
   apps/auth-service/src/auth/auth.controller.ts (fragmento del método

createUserByAdmin)

TypeScript

```
import { Controller, Post, Body, BadRequestException, Param, Get, ForbiddenException, Req, } from '@nestjs/common';
import { AuthService } from './auth.service.js';
import { LoginDto } from '../../dto/login.dto.js';
import { ForgotPasswordDto } from '../../dto/forgot-password.dto.js';
import { ResetPasswordDto } from '../../dto/reset-password.dto.js';
import { CreateUserAdminDto } from '../../dto/create-user-admin.dto.js';
import { UsersService } from '../users/users.service.js';
import { RolesService } from '../roles/roles.service.js'; // Importa RolesService
import { ApiTags, ApiOperation, ApiParam, ApiResponse, ApiBody, } from '@nestjs/swagger';
import { UserRole } from '../../../packages/common-types/src/index.js'; // Importa UserRole

@ApiTags('auth')
@Controller('auth')
export class AuthController {
  constructor(
    private authService: AuthService,
    private usersService: UsersService,
    private rolesService: RolesService, // Inyecta RolesService
  ) {}

  //... (otros endpoints de login y forgot/reset password)

  @ApiOperation({ summary: 'Crear un usuario por un administrador (con rol inicial)' })
  @ApiResponse({ status: 201, description: 'Usuario creado y especialización de rol solicitada.' })
  @ApiResponse({ status: 400, description: 'Solicitud inválida o rol no soportado.' })
  @ApiParam({ name: 'role', enum: UserRole, description: 'Rol a especializar (PATIENT, DOCTOR, ADMIN)' })
  @ApiBody({ type: CreateUserAdminDto })
  @Post('root/create-user/:role')
  async createUserByAdmin(
    @Param('role') role: string,
    @Body() createUserAdminDto: CreateUserAdminDto,
  ) {
    const validSpecializationRoles: UserRole =;
    const upperRole = role.toUpperCase() as UserRole;
```

```
    if (!validSpecializationRoles.includes(upperRole)) {
      throw new BadRequestException(`Invalid role for user creation: ${role}. Supported roles are: ${validSpecializationRoles.join(', ')}`);
    }

    const user = await this.usersService.createUserByAdmin(createUserAdminDto);

    // Llamar al servicio de roles para solicitar la especialización
    await this.rolesService.requestRoleSpecialization(user.idUser, upperRole);

    return { message: `User ${user.idUser} created and specialization as ${upperRole} requested.`, user };
  }
}
```

4. Actualizar apps/auth-service/src/auth/auth.service.ts (validateUser y login):
   Este es un cambio CRÍTICO. Si los roles no se guardan en la DB del auth-service, deben obtenerse dinámicamente en el login consultando los servicios de dominio.
   apps/auth-service/src/auth/auth.service.ts (fragmento)
   TypeScript

```
import { Injectable, UnauthorizedException, NotFoundException, BadRequestException, InternalServerErrorException, Logger } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import * as bcrypt from 'bcryptjs';
import { UsersService } from '../users/users.service.js';
import { User, UserRole } from '../../../packages/common-types/src/index.js'; // Importa UserRole
import { ConfigService } from '@nestjs/config';
import { PrismaService } from '../prisma.service.js';
import { v4 as uuidv4 } from 'uuid';
import { HttpService } from '@nestjs/axios'; // Importa HttpService
import { firstValueFrom } from 'rxjs'; // Para manejar observables de HttpService

@Injectable()
export class AuthService {
  private readonly logger = new Logger(AuthService.name);
  private readonly cardiologyServiceUrl: string; // URL del servicio de Cardiología
  private readonly patientServiceUrl: string; // URL del servicio de Pacientes (Azure)
```

```
  constructor(
    private usersService: UsersService,
    private jwtService: JwtService,
    private configService: ConfigService,
    private prisma: PrismaService,
    private httpService: HttpService, // Inyecta HttpService
    // private eventPublisherService: EventPublisherService, // Descomentar si se usa para emails asíncronos
  ) {
    // Configura las URLs de los servicios de dominio desde variables de entorno
    this.cardiologyServiceUrl = this.configService.get<string>('CARDIO_SERVICE_URL');
    this.patientServiceUrl = this.configService.get<string>('PATIENT_SERVICE_URL');
  }

  /**
   * Determina los roles de un usuario consultando los servicios de dominio.
   * @param idUser ID del usuario.
   * @returns Array de roles del usuario.
   */
  private async getUserRolesFromDomainServices(idUser: number): Promise<UserRole> {
    const roles: UserRole =;
    try {
      // Consultar CardiologyService para roles de Empleado (Medico, Admin)
      // Endpoint de ejemplo: GET /employees/roles/:idUser
      const employeeRolesResponse = await firstValueFrom(
        this.httpService.get(`${this.cardiologyServiceUrl}/employees/roles/${idUser}`),
      );
      if (employeeRolesResponse.data && employeeRolesResponse.data.roles) {
        employeeRolesResponse.data.roles.forEach((role: UserRole) => {
          if (!roles.includes(role)) roles.push(role);
        });
      }

      // Consultar PatientService para rol de Paciente
      // Endpoint de ejemplo: GET /patients/roles/:idUser
      const patientRolesResponse = await firstValueFrom(
        this.httpService.get(`${this.patientServiceUrl}/patients/roles/${idUser}`),
      );
      if (patientRolesResponse.data && patientRolesResponse.data.roles) {
```

```
      patientRolesResponse.data.roles.forEach((role: UserRole) => {
        if (!roles.includes(role)) roles.push(role);
      });
    }

    // Si el usuario es el ROOT temporal (solo en desarrollo)
    if (idUser === Number(this.configService.get<string>('ROOT_USER_ID_DEV'))) {
      if (!roles.includes(UserRole.ROOT)) roles.push(UserRole.ROOT);
    }

  } catch (error) {
    this.logger.error(`Failed to fetch roles for user ${idUser} from domain services:
${error.message}`);
    // Dependiendo de la criticidad, podrías lanzar una excepción o retornar un array vacío
    throw new InternalServerErrorException('Failed to retrieve user roles.');
  }
  return roles;
 }

 async validateUser(email: string, pass: string, expectedRole: UserRole): Promise<any> {
  const user = await this.usersService.findUserByEmail(email);
  if (!user ||!(await bcrypt.compare(pass, user.passwordHash))) {
    throw new UnauthorizedException('Invalid credentials');
  }

  // Obtener roles dinámicamente de los servicios de dominio
  const actualRoles = await this.getUserRolesFromDomainServices(user.idUser);

  // Validar que el usuario tenga el rol esperado para el endpoint de login
  if (!actualRoles.includes(expectedRole)) {
    throw new UnauthorizedException(`User does not have the ${expectedRole} role.`);
  }

  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  const { passwordHash,...result } = user;
  return {...result, roles: actualRoles }; // Retorna el usuario sin hash y con roles
 }

 async login(user: { idUser: number; email: string; roles: UserRole }) {
```

```
    const payload = { email: user.email, sub: user.idUser, roles: user.roles }; // Incluye roles en el payload del JWT
    return {
      access_token: this.jwtService.sign(payload),
    };
  }

  async register(createUserDto: CreateUserAdminDto): Promise<User> {
    // Hashear la contraseña antes de pasarla al UsersService
    const hashedPassword = await bcrypt.hash(
      createUserDto.password,
      Number(this.configService.get<number>('BCRYPT_SALT_ROUNDS') |
```

```
| 10),
);
```

```
    // Preparar los datos para createUser, incluyendo los nuevos campos y el hash
    const userData = {
      idUser: createUserDto.idUser,
      name: createUserDto.name,
      email: createUserDto.email,
      passwordHash: hashedPassword,
      gender: createUserDto.gender,
      address: createUserDto.address,
      birthDate: new Date(createUserDto.birthDate), // Convertir a Date
      telUser: createUserDto.telUser?.map(tel => ({ telephone: BigInt(tel.telephone) })), // Convertir a BigInt
    };

    return this.usersService.createUserByAdmin(userData);
  }

  //... (forgotPassword y resetPassword métodos, sin cambios significativos)
}
```

Nota: He añadido `HttpService` al constructor de `AuthService` y he creado un método `getUserRolesFromDomainServices` para centralizar la lógica de consulta de roles. Las URLs de los servicios de dominio (`CARDIO_SERVICE_URL`, `PATIENT_SERVICE_URL`) deberán configurarse en el `.env` del `auth-service`. Los servicios de dominio deberán exponer endpoints para verificar si un `idUser` tiene un rol específico o para listar sus roles.

5. Actualizar apps/auth-service/src/auth/auth.module.ts:
   Asegúrate de importar HttpModule para HttpService.
   apps/auth-service/src/auth/auth.module.ts
   TypeScript

```
import { Module } from '@nestjs/common';
import { AuthController } from './auth.controller.js';
import { AuthService } from './auth.service.js';
import { UsersModule } from '../users/users.module.js';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { JwtStrategy } from './jwt.strategy.js';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { RolesService } from '../roles/roles.service.js';
import { HttpModule } from '@nestjs/axios'; // Importa HttpModule

@Module({
 imports: [
  UsersModule,
  PassportModule,
  JwtModule.registerAsync({
   imports: [ConfigModule],
   useFactory: (configService: ConfigService) => ({
    secret: configService.get<string>('JWT_SECRET'),
    signOptions: { expiresIn: '60m' },
   }),
   inject:,
  }),
  ConfigModule,
  HttpModule, // Añade HttpModule
 ],
 controllers: [AuthController],
```

```
  providers:, // Asegúrate de que RolesService esté aquí
  exports:,
})
export class AuthModule {}
```

6. Actualizar apps/auth-service/src/users/users.service.ts:
   Eliminar cualquier referencia a roles o UserRoleAssignment, ya que los roles se infieren de los servicios de dominio.
   apps/auth-service/src/users/users.service.ts (fragmento)
   TypeScript

```
import { Injectable, BadRequestException } from '@nestjs/common';
import { PrismaService } from '../prisma.service.js';
import { User, Prisma } from '../../generated/prisma/index.js'; // Elimina UserRole, UserRoleAssignment
import * as bcrypt from 'bcryptjs';

@Injectable()
export class UsersService {
  constructor(private prisma: PrismaService) {}

  async user(idUser: number): Promise<User | null> {
    return await this.prisma.user.findUnique({
      where: { idUser: idUser },
      include: { telUser: true }, // Mantén telUser si es necesario
    });
  }

  async users(params: { /*... */ }): Promise<User> {
    const { skip, take, cursor, where, orderBy } = params;
    return await this.prisma.user.findMany({
      skip,
      take,
      cursor,
      where,
      orderBy,
      include: { telUser: true },
    });
  }
```

```
 async createUserByAdmin(data: {
   idUser: number;
   name: string;
   email: string;
   passwordHash: string;
   gender: string;
   address: string;
   birthDate: Date;
   telUser?: { telephone: bigint };
 }): Promise<User> {
   // Validar que el idUser no exista
   const existingUserById = await this.prisma.user.findUnique({ where: { idUser:
data.idUser } });
   if (existingUserById) {
     throw new BadRequestException(`User with ID ${data.idUser} already exists.`);
   }

   // Validar que el email no exista
   const existingUserByEmail = await this.prisma.user.findUnique({ where: { email:
data.email } });
   if (existingUserByEmail) {
     throw new BadRequestException(`User with email ${data.email} already exists.`);
   }

   return await this.prisma.user.create({
     data: {
       idUser: data.idUser,
       name: data.name,
       email: data.email,
       passwordHash: data.passwordHash,
       gender: data.gender,
       address: data.address,
       birthDate: data.birthDate,
       telUser: data.telUser? {
         create: data.telUser.map(tel => ({ telephone: tel.telephone })),
       } : undefined,
     },
     include: { telUser: true },
   });
```

```
  }

  //... (otros métodos, eliminando cualquier referencia a roles o UserRoleAssignment)
}
```

7. Actualizar apps/auth-service/src/auth/jwt.strategy.ts:
   La estrategia JWT recibirá los roles directamente en el payload del JWT, por lo que no necesita consultar la base de datos para obtenerlos.
   apps/auth-service/src/auth/jwt.strategy.ts
   TypeScript

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { ConfigService } from '@nestjs/config';
import { UsersService } from '../users/users.service.js';
import { User, UserRole } from '../../../packages/common-types/src/index.js'; // Importa UserRole

// Define la interfaz para el payload del JWT
interface JwtPayload {
  email: string;
  sub: number; // idUser
  roles: UserRole; // Roles del usuario (ya vienen en el payload)
}

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    private configService: ConfigService,
    private usersService: UsersService, // Todavía necesario para buscar el usuario por id
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: configService.get<string>('JWT_SECRET'),
    });
  }

  async validate(payload: JwtPayload): Promise<User & { roles: UserRole }> {
    const user = await this.usersService.user(payload.sub); // Busca por idUser
```

```
    if (!user) {
      throw new UnauthorizedException();
    }
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    const { passwordHash,...result } = user; // No retornar la contraseña hasheada
    // Los roles ya vienen en el payload, simplemente los adjuntamos
    return {...(result as User), roles: payload.roles };
  }
}
```

8. Actualizar apps/auth-service/prisma/schema.prisma:
   Eliminar el modelo UserRoleAssignment y cualquier referencia a roles en el modelo User.
   apps/auth-service/prisma/schema.prisma
   Fragmento de código

```
// apps/auth-service/prisma/schema.prisma
generator client {
  provider = "prisma-client-js"
  output   = "../generated/prisma"
}

datasource db {
  provider = "postgresql"
  url      = env("USER_DB_URL")
}

model User {
  idUser            Int               @id @map("idUser") // La cédula como ID principal
  name              String            @db.VarChar(100)
  email             String            @unique
  passwordHash      String
  gender            String
  address           String            @db.VarChar(100)
  birthDate         DateTime
  telUser           TelUser // Relación uno a muchos con TelUser
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt
  passwordResetToken PasswordResetToken? @relation(name:
"UserPasswordResetToken")
```

```
}

model TelUser {
 idUser    Int
 telephone BigInt
 user      User     @relation(fields: [idUser], references: [idUser], onDelete:
Cascade)
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt

 @@id([idUser, telephone])
 @@map("TelUser")
}

model PasswordResetToken {
 id        String   @id @default(uuid())
 token     String   @unique
 userId    Int      @unique
 expiresAt DateTime
 createdAt DateTime @default(now())
 user      User     @relation(name: "UserPasswordResetToken", fields: [userId],
references: [idUser], onDelete: Cascade)
}
```

Después de modificar schema.prisma, ejecuta npx prisma migrate dev --name remove_role_assignment_table y npx prisma generate en apps/auth-service.

V. Reacción de Otros Microservicios (Consumidores de QueueT)

Este es el paso crucial para la consistencia distribuida. Otros microservicios que necesiten saber sobre las especializaciones de roles se suscribirán a los eventos publicados por el role-assignment-service.

1. Actualizar packages/common-types/src/enums/user-roles.enum.ts:
   Asegúrate de que el enum UserRole en tu paquete common-types incluya todos los roles, incluyendo ROOT.

packages/common-types/src/enums/user-roles.enum.ts

```typescript
export enum UserRole {
  PATIENT = 'PATIENT',
  DOCTOR = 'DOCTOR',
  ADMIN = 'ADMIN',
  ROOT = 'ROOT',
}
```

2. Crear Definición de Evento en packages/common-types/src/events/user-events.ts:
   Define las interfaces para los payloads de los eventos de especialización.
   packages/common-types/src/events/user-events.ts

```typescript
import { UserRole } from '../enums/user-roles.enum.js';

export interface UserSpecializationRequestedEventPayload {
  userId: number;
  role: UserRole;
  timestamp: string;
  // Otros datos que el role-assignment-service pueda conocer y sean útiles para el consumidor
}
```

3. Ejemplo: CardiologyService (Creación de Empleado o Medico)
   El CardiologyService (ubicado en AWS) necesita crear una entrada en su tabla Empleado o Medico cuando un usuario es asignado a un rol relevante.
   - **Instalación de Dependencias en CardiologyService:**

     ```bash
     cd apps/cardiology-service
     npm install @tradologics/queuet @nestjs/config
     ```

   - Configuración de Variables de Entorno para Redis:
     apps/cardiology-service/.env

     ```
     #... otras variables
     REDIS_HOST=localhost
     REDIS_PORT=6379
     QUEUE_NAME=hospital_events # Debe coincidir con el nombre de la cola del publicador
     ```

- Creación del EventSubscriberService en CardiologyService:
  Este servicio escuchará los eventos de QueueT.
  apps/cardiology-service/src/events/event.subscriber.ts
  TypeScript

```
import { Injectable, OnModuleInit, OnModuleDestroy, Logger } from '@nestjs/common';
import * as QueueT from '@tradologics/queuet';
import { ConfigService } from '@nestjs/config';
import { UserRole, UserSpecializationRequestedEventPayload } from '../../../packages/common-types/src/index.js';
// import { EmployeesService } from '../employees/employees.service.js'; // Servicio para crear empleados/médicos
// import { PatientsService } from '../patients/patients.service.js'; // Si Cardiología también maneja pacientes locales

@Injectable()
export class EventSubscriberService implements OnModuleInit, OnModuleDestroy {
  private readonly logger = new Logger(EventSubscriberService.name);
  private readonly queueName: string;

  constructor(
    private configService: ConfigService,
    // private employeesService: EmployeesService, // Inyecta el servicio de empleados
    // private patientsService: PatientsService, // Inyecta el servicio de pacientes
  ) {
    this.queueName = this.configService.get<string>('QUEUE_NAME') |
```

| 'hospital_events';
}

```
  async onModuleInit() {
    const redisHost = this.configService.get<string>('REDIS_HOST') |
```

| 'localhost';
const redisPort = this.configService.get('REDIS_PORT') |
| 6379;

```
      QueueT.initialize(this.queueName, { host: redisHost, port: redisPort });
      this.logger.log(`Subscriber initialized for queue: ${this.queueName} on Redis:
${redisHost}:${redisPort}`);

      QueueT.subscribe(this.handleMessage.bind(this));
    }

    private async handleMessage(id: string, data: string) {
      try {
        const message = JSON.parse(data);
        this.logger.log(`Received event: ${message.eventType} with payload:`,
message.payload);

        switch (message.eventType) {
          case 'DoctorSpecializationRequested':
            const doctorPayload: UserSpecializationRequestedEventPayload =
message.payload;
            this.logger.log(`Processing DoctorSpecializationRequested for user
${doctorPayload.userId}`);
            // Lógica para crear la entrada en la tabla Empleado y Medico en la DB de
Cardiología
            // await
this.employeesService.createEmployeeAndDoctor(doctorPayload.userId, /* otros
datos del médico */);
            break;
          case 'AdminSpecializationRequested':
            const adminPayload: UserSpecializationRequestedEventPayload =
message.payload;
            this.logger.log(`Processing AdminSpecializationRequested for user
${adminPayload.userId}`);
            // Lógica para crear la entrada en la tabla Empleado y P.Admin en la DB de
Cardiología
            // await
this.employeesService.createEmployeeAndAdmin(adminPayload.userId, /* otros datos
del administrativo */);
```

```
            break;
          case 'PatientSpecializationRequested':
            const patientPayload: UserSpecializationRequestedEventPayload =
message.payload;
            this.logger.log(`Processing PatientSpecializationRequested for user
${patientPayload.userId}`);
            // Si Cardiología necesita una referencia local de pacientes (aunque el
maestro esté en Azure)
            // await
this.patientsService.createLocalPatientReference(patientPayload.userId, /* datos */);
            break;
          // Otros eventos que Cardiología pueda necesitar procesar (ej.
PrescripcionCreada)
          default:
            this.logger.warn(`Unhandled event type: ${message.eventType}`);
        }

        await QueueT.delete(id); // Eliminar el mensaje una vez procesado exitosamente
      } catch (error) {
        this.logger.error(`Error processing message ${id}:`, error);
        // **Manejo de Errores:** En producción, aquí implementarías lógica para:
        // 1. Reintentar el procesamiento después de un retardo (ej., usando un
mecanismo de reintentos).
        // 2. Mover el mensaje a una "cola de mensajes fallidos" (Dead-Letter Queue -
DLQ) para inspección manual.
        // QueueT no tiene DLQ nativa, tendrías que implementarla manualmente o usar
una biblioteca más robusta para producción.
      }
    }

    async onModuleDestroy() {
      QueueT.disconnect();
      this.logger.log(`Subscriber disconnected.`);
    }
  }
  ```
```

4. Integración del EventSubscriberService en el CardiologyModule:
   apps/cardiology-service/src/cardiology.module.ts (o app.module.ts)
```

```
TypeScript
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { EventSubscriberService } from './events/event.subscriber.js';
// import { EmployeesModule } from './employees/employees.module.js'; // Si tienes un módulo de empleados

@Module({
  imports:,
  controllers:,
  providers:,
})
export class CardiologyModule {}
```

VI. Consideraciones Adicionales y Buenas Prácticas

- **Consistencia de IDs:** El idUser (cédula) es la clave principal para la consistencia entre el auth-service y el role-assignment-service, y luego para los servicios de dominio (Cardiología, Pacientes). Asegúrate de que este ID sea siempre el mismo y se propague correctamente.
- **Manejo de BigInt:** Si TelUser.telephone es BigInt en Prisma, asegúrate de que en tus DTOs y lógica de negocio lo manejes como BigInt de JavaScript (BigInt(value)).
- **Reglas de Negocio de Roles:** La regla "Admin puede ser Paciente PERO NO Médico" (y otras similares) ahora debe ser aplicada en los **servicios de dominio** (ej., CardiologyService al crear Medico o P.Admin) o en el auth-service al generar el JWT. El role-assignment-service solo orquesta la especialización.
- **Manejo de Errores en QueueT:** Para producción, la lógica de handleMessage en los suscriptores debe ser más robusta, incluyendo reintentos con *backoff* exponencial y el envío de mensajes fallidos a una Dead-Letter Queue (DLQ) para evitar la pérdida de mensajes y permitir la depuración.
- **Transacciones Distribuidas (Consistencia Eventual):** Recuerda que el patrón Saga con QueueT garantiza **consistencia eventual**. Esto significa que puede haber un breve período de tiempo en el que un usuario se crea en auth-service pero su especialización de rol aún no se ha reflejado en los servicios de dominio. Para la mayoría de los casos de uso, esto es aceptable.

- **Seguridad de DBLINK:** Las credenciales de DBLINK (user, password) deben ser gestionadas de forma segura (ej., con variables de entorno o un gestor de secretos en producción) y el usuario de la base de datos remota (wally en este caso) debe tener el **mínimo privilegio** necesario (solo SELECT en la tabla User del auth-service).
- **Swagger:** La documentación interactiva de la API del role-assignment-service estará disponible en /api (ej., http://localhost:3002/api en desarrollo).

Con esta guía detallada, tienes una hoja de ruta clara para implementar el role-assignment-service y su integración con el auth-service y el sistema de mensajería QueueT, siguiendo el nuevo modelo de especialización de roles por tablas de dominio.