MANUAL TÉCNICO

Sistema DICRI – Arquitectura, Diseño y Desarrollo Técnico

Versión 1.0 – Noviembre 2025

SISTEMA DICRI

Arquitectura, Diseño y Desarrollo Técnico

Ministerio Público – Prueba Técnica

Autor: Sergio Danilo Pérez Martínez

Fecha: 23/11/2025

MINISTERIO PÚBLICO
Ciencia • Verdad • Justicia

Documento Técnico – Versión 1.0

TABLA DE CONTENIDO

1. Introducción
2. Objetivos del sistema
3. Alcance del desarrollo
4. Tecnologías utilizadas
5. Arquitectura general del sistema
6. Contenerización y despliegue (Docker)
7. Estructura del proyecto
8. Modelo relacional y ER
9. Roles, permisos y flujo institucional
10. Procedimientos almacenados
11. API REST – Diseño y endpoints
12. Seguridad y autenticación (JWT)
13. Pruebas unitarias
14. Pruebas Postman
15. Interfaz de usuario (frontend)
16. Justificación de decisiones técnicas
17. Conclusiones

1. INTRODUCCIÓN

El presente documento describe de manera técnica y formal el desarrollo del Sistema DICRI, una plataforma web diseñada para la gestión integral de expedientes e indicios, incorporando mecanismos de control, trazabilidad operativa, reportería analítica y aplicación de roles institucionales.
El sistema reproduce el flujo de trabajo utilizado en dependencias del Ministerio Público, permitiendo registrar expedientes, administrar indicios asociados, controlar su evolución por estados, y documentar las decisiones tomadas por los diferentes perfiles involucrados (Técnico, Coordinador y Administrador).

El manual presenta la arquitectura general del sistema, el diseño de sus componentes frontend y backend, la estructura y normalización del modelo de datos, así como la implementación de procedimientos almacenados en SQL Server para garantizar integridad, eficiencia y consistencia transaccional.
También se describen los mecanismos de seguridad aplicados—incluyendo autenticación mediante JWT y control de permisos por roles—junto con las estrategias empleadas para desplegar la aplicación en un entorno contenerizado utilizando Docker.

Adicionalmente, el documento incluye la descripción de los módulos de reportería y análisis, los cuales permiten generar resúmenes estadísticos y series temporales basadas en la actividad registrada en el sistema. Estos reportes están orientados a brindar soporte para procesos de supervisión, auditoría y toma de decisiones institucionales.

Finalmente, se documentan las pruebas unitarias y funcionales realizadas, así como las decisiones técnicas adoptadas durante el desarrollo, con el fin de garantizar un sistema robusto, escalable y alineado con las buenas prácticas de ingeniería de software.

2. OBJETIVOS DEL SISTEMA

1. Proveer una plataforma unificada para el registro, administración y seguimiento integral de expedientes, asegurando consistencia en la información y disponibilidad para los distintos perfiles operativos.
2. Permitir la gestión completa de indicios asociados a cada expediente, garantizando orden, correlatividad, integridad y trazabilidad de cada elemento registrado.
3. Implementar un flujo formal y controlado de estados, que incluya registro inicial, revisión, rechazo justificable, aprobación y archivado, manteniendo el historial cronológico de decisiones realizadas por los perfiles autorizados.
4. Garantizar trazabilidad y control institucional mediante:
    - auditoría de acciones relevantes,
    - registro de usuarios que crean y actualizan información,
    - aplicación estricta de permisos según roles (Técnico, Coordinador y Administrador).
5. Proveer herramientas de reportería y análisis que permitan visualizar indicadores operativos, estadísticas por estado y series temporales, facilitando labores de supervisión, seguimiento y toma de decisiones.
6. Establecer una arquitectura escalable, modular y mantenible, basada en separación por capas, API REST, procedimientos almacenados y un front-end desacoplado.
7. Asegurar un entorno portable y reproducible mediante contenedores Docker, permitiendo su despliegue completo con un solo comando, sin dependencias externas ni configuraciones manuales adicionales.

3. ALCANCE DEL DESARROLLO

El desarrollo realizado para el sistema DICRI comprende la implementación de una solución tecnológica integral que cubre los siguientes componentes:

1. Frontend

- Desarrollo completo de la interfaz web utilizando React + TypeScript.
- Implementación de vistas para administración de expedientes, indicios, historial y reportería.
- Validaciones de formularios, navegación por tabs y experiencia de usuario optimizada.
- Consumo de la API mediante servicios centralizados.

2. Backend

- Construcción de una API REST modular con Node.js + Express + TypeScript.
- Estructura en capas (rutas, controladores, servicios y modelo de datos).
- Middleware de autenticación JWT y autorización basada en roles.
- Lógica de negocio unificada a través de servicios y stored procedures.

3. Base de Datos

- Diseño y normalización del modelo relacional en SQL Server.
- Tablas para usuarios, roles, expedientes, indicios, catálogos, estados y auditoria básica.
- Índices, claves foráneas y borrado lógico para garantizar integridad y trazabilidad.
- Implementación de stored procedures para todas las operaciones CRUD y consultas analíticas.

4. Flujos Operativos

- Registro, edición y seguimiento completo de expedientes.
- Gestión correlativa y controlada de indicios asociados.
- Historial cronológico de estados del expediente y registro de motivos de rechazo.
- Permisos diferenciados según rol (Técnico, Coordinador, Administrador).

5. Reportería y Analítica

- Endpoints para generación de indicadores operativos (por estado y rango de fechas).
- Series temporales para análisis de actividad.
- Exportación de información relevante a formatos visuales (PDF).

6. Seguridad

- Autenticación mediante tokens JWT.
- Control de acceso basado en roles desde el backend.
- Validaciones de integridad y sanitización de parámetros de entrada.

7. Pruebas

- Pruebas unitarias base para servicios, controladores y middleware.
- Simulación de solicitudes mediante Postman para verificación funcional.

8. Contenerización y Despliegue

- Orquestación de servicios mediante Docker Compose.
- Contenedores para SQL Server, backend, frontend y servidor web NGINX.
- Script de inicialización automática de la base de datos (db-init).
- Entorno reproducible con un solo comando.

4. TECNOLOGÍAS UTILIZADAS

Frontend

El frontend se desarrolló como una aplicación SPA moderna, priorizando rendimiento, modularidad y mantenibilidad.

- React + Vite — Framework declarativo y compilador ultrarrápido para aplicaciones SPA.
- TypeScript — Tipado estático para reducir errores y mejorar calidad del código.
- Material UI — Biblioteca de componentes accesibles y consistentes para interfaz profesional.
- React Router DOM — Control de navegación cliente-side con rutas protegidas.
- Axios — Cliente HTTP para consumo estructurado de la API.

Backend

El backend expone una API REST modular, escalable y desacoplada.

- Node.js — Entorno de ejecución basado en eventos, ideal para servicios REST.
- Express — Micro-framework minimalista para definir rutas, middlewares y controladores.
- TypeScript — Facilita tipado fuerte, desarrollo por capas y mantenimiento a largo plazo.
- mssql — Driver oficial para SQL Server con soporte para stored procedures.
- Jest + Supertest — Framework de pruebas unitarias e integración para endpoints API.

Base de Datos

Se utilizó SQL Server 2022 por su soporte robusto para transacciones, integridad referencial y stored procedures.

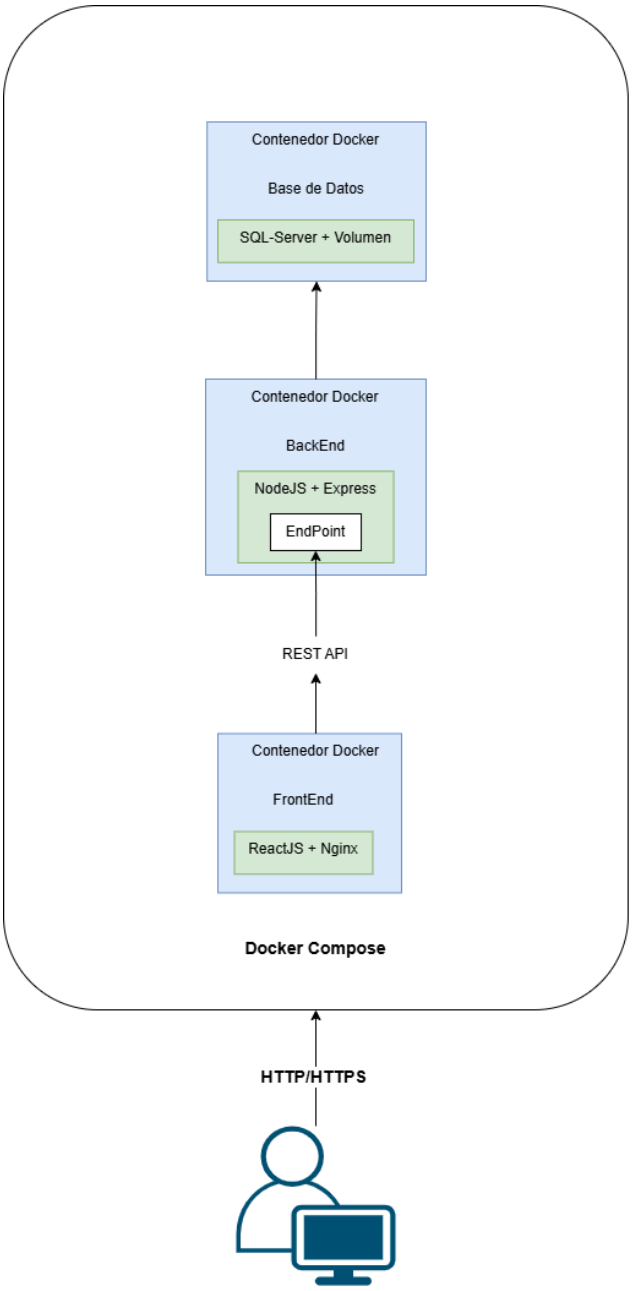
- Microsoft SQL Server 2022 — Motor relacional con alta estabilidad en entornos institucionales.
- T-SQL — Lenguaje para consultas, batch processing y lógica en stored procedures.
- Stored Procedures (SP) — Encapsulan la lógica transaccional crítica (insert/update/listado).
- Índices y constraints — Optimizan rendimiento y garantizan integridad.
- Borrado lógico y trazabilidad — Persistencia del historial y control de auditoría.

Infraestructura y Despliegue

El sistema está diseñado para ser portable, reproducible y fácil de levantar en cualquier entorno evaluador.

- Docker — Empaquetado del entorno completo sin dependencias externas.
- docker-compose — Orquestación de servicios con dependencias y tiempos de arranque controlados.
- Healthchecks — Validación automática del estado del SQL Server previo a inicialización.
- Contenedores independientes:
    - SQL Server — Motor de base de datos.
    - db-init — Ejecuta init.sql solo cuando SQL está listo para recibir conexiones.
    - Backend — API REST en Node.js.
    - Frontend — Aplicación React servida con Nginx.

5. ARQUITECTURA GENERAL DEL SISTEMA



Patrones utilizados

1. Arquitectura en Capas (Layered Architecture)

Separación explícita entre:

- **Capa de Rutas**
- **Capa de Controladores**
- **Capa de Servicios (lógica de negocio)**
- **Capa de Acceso a Datos (SP y consultas)**

Este patrón mejora mantenibilidad, testabilidad y desac acoplamiento.

2. Separación de Responsabilidades (SoC – Separation of Concerns)

Cada módulo cumple una única función:

- Controladores validan inputs y manejan respuestas.
- Servicios contienen la lógica.
- Middlewares aplican seguridad.
- Config maneja ambiente y conexión a BD.

Aumenta claridad y reduce acoplamiento entre módulos.

3. Patron Middleware (Express Middleware Pattern)

Utilizado para:

- Autenticación JWT
- Verificación de roles
- Manejo centralizado de errores
- Logging básico

Permite reutilizar lógica transversal en distintos endpoints.

4. Encapsulamiento de Lógica SQL mediante Stored Procedures

El backend **no contiene SQL hardcodeado**, sino que delega a SP:

- Operaciones CRUD
- Flujo de estados
- Reporterías
- Series temporales

Esto permite centralizar la lógica transaccional, mejorar seguridad y facilitar auditorías.

5. Patrón Repository (Implementación Ligera)

Aunque no se implementó un repositorio completamente aislado, el servicio de base de datos encapsula:

- ejecución de stored procedures
- mapeo de resultados
- manejo de errores

Funciona como un *mini-repositorio* para desacoplar Express de SQL.

6. Patrón DTO (Data Transfer Objects) – versión simplificada

Aplicado en:

- Validación de requests
- Estructuración de respuestas
- Normalización de la API

Evita exponer estructuras internas o inconsistentes.

7. Patrón “State Machine” aplicado al flujo institucional

Los estados del expediente siguen una secuencia controlada:

- Registrado → En revisión → Aprobado / Rechazado → Archivado

Esto implementa un comportamiento similar a una **máquina de estados**, controlado desde backend + SP.

8. Contenerización (Container Pattern)

Cada servicio del sistema es un contenedor aislado:

- SQL Server
- Backend
- Frontend
- db-init

Esto garantiza portabilidad, reproducibilidad y separación de entornos.

6. CONTENERIZACIÓN Y DESPLIEGUE

Servicios del docker-compose:

Servicio	Función
sqlserver	Motor de base de datos
db-init	Ejecuta init.sql esperando healthcheck
backend	API Node/Express
frontend	SPA React

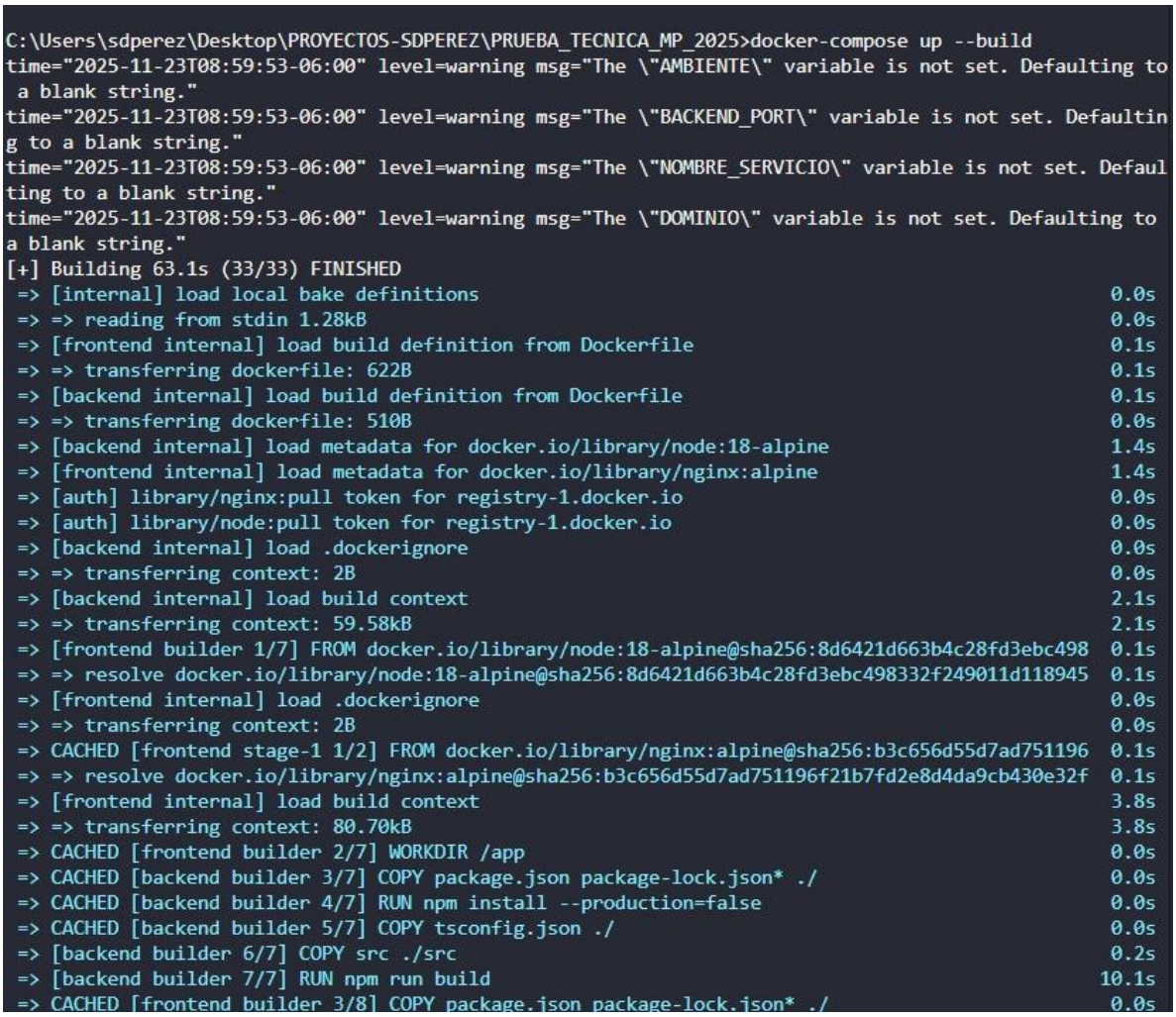
Healthcheck de SQL Server

Asegura que la base de datos **esté completamente lista** antes de ejecutar scripts o iniciar el backend.

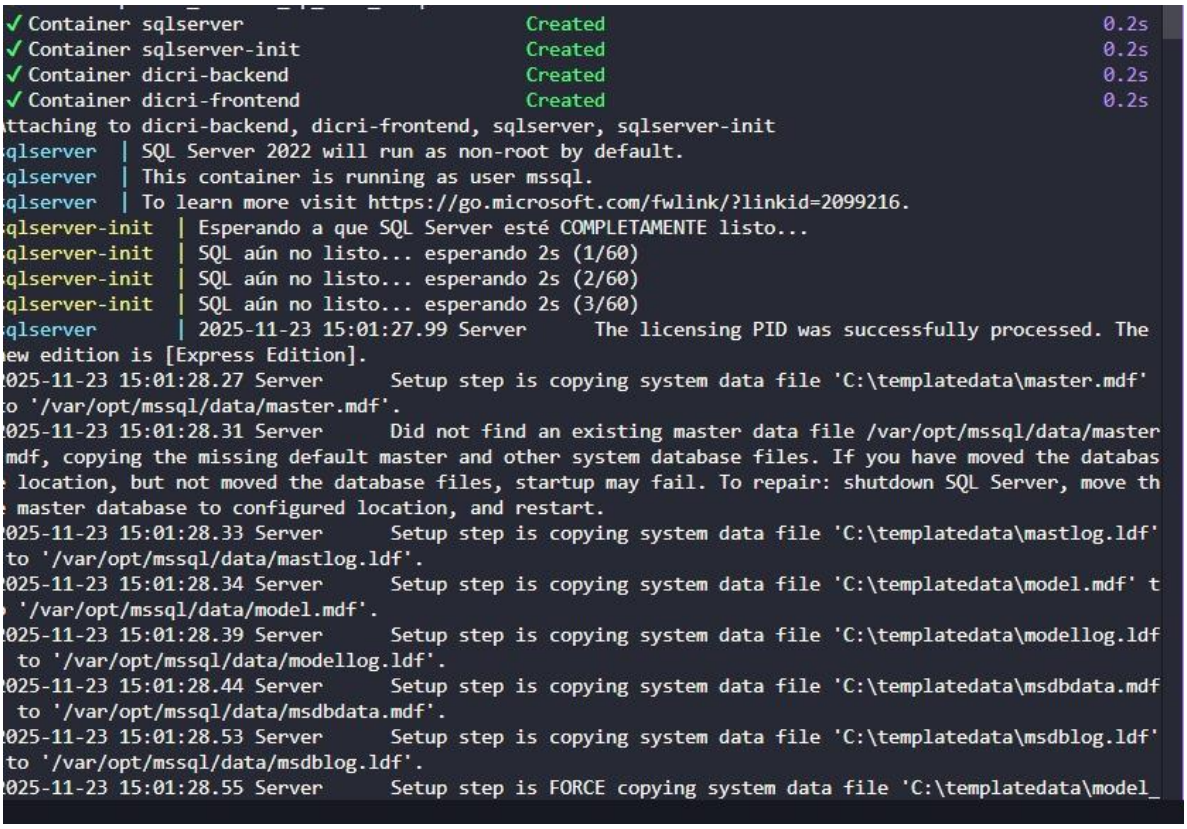
Flujo de arranque

1. sqlserver inicia y se verifica con healthcheck.
2. db-init ejecuta init.sql (creación de tablas, catálogos, SP).
3. backend inicia solo cuando db-init termina.

4. frontend inicia cuando contenedor backend está disponible.



```
C:\Users\sdperez\Desktop\PROYECTOS-SDPEREZ\PRUEBA_TECNICA_MP_2025>docker-compose up --build
time="2025-11-23T08:59:53-06:00" level=warning msg="The \"AMBIENTE\" variable is not set. Defaulting to
 a blank string."
time="2025-11-23T08:59:53-06:00" level=warning msg="The \"BACKEND_PORT\" variable is not set. Defaultin
g to a blank string."
time="2025-11-23T08:59:53-06:00" level=warning msg="The \"NOMBRE_SERVICIO\" variable is not set. Defaul
ting to a blank string."
time="2025-11-23T08:59:53-06:00" level=warning msg="The \"DOMINIO\" variable is not set. Defaulting to
a blank string."
[+] Building 63.1s (33/33) FINISHED
 => [internal] load local bake definitions                                                    0.0s
 => => reading from stdin 1.28kB                                                              0.0s
 => [frontend internal] load build definition from Dockerfile                                 0.1s
 => => transferring dockerfile: 622B                                                          0.1s
 => [backend internal] load build definition from Dockerfile                                  0.1s
 => => transferring dockerfile: 510B                                                          0.0s
 => [backend internal] load metadata for docker.io/library/node:18-alpine                     1.4s
 => [frontend internal] load metadata for docker.io/library/nginx:alpine                      1.4s
 => [auth] library/nginx:pull token for registry-1.docker.io                                  0.0s
 => [auth] library/node:pull token for registry-1.docker.io                                   0.0s
 => [backend internal] load .dockerignore                                                     0.0s
 => => transferring context: 2B                                                               0.0s
 => [backend internal] load build context                                                     2.1s
 => => transferring context: 59.58kB                                                          2.1s
 => [frontend builder 1/7] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498  0.1s
 => => resolve docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945  0.1s
 => [frontend internal] load .dockerignore                                                    0.0s
 => => transferring context: 2B                                                               0.0s
 => CACHED [frontend stage-1 1/2] FROM docker.io/library/nginx:alpine@sha256:b3c656d55d7ad751196  0.1s
 => => resolve docker.io/library/nginx:alpine@sha256:b3c656d55d7ad751196f21b7fd2e8d4da9cb430e32f  0.1s
 => [frontend internal] load build context                                                    3.8s
 => => transferring context: 80.70kB                                                          3.8s
 => CACHED [frontend builder 2/7] WORKDIR /app                                                0.0s
 => CACHED [backend builder 3/7] COPY package.json package-lock.json* ./                      0.0s
 => CACHED [backend builder 4/7] RUN npm install --production=false                           0.0s
 => CACHED [backend builder 5/7] COPY tsconfig.json ./                                        0.0s
 => [backend builder 6/7] COPY src ./src                                                      0.2s
 => [backend builder 7/7] RUN npm run build                                                  10.1s
 => CACHED [frontend builder 3/8] COPY package.json package-lock.json* ./                     0.0s
```



7. ESTRUCTURA DEL PROYECTO (DETALLADA)

Backend

/src/routes

Define endpoints y aplica middlewares.

/src/controllers

Controladores por dominio: expedientes, indicios, usuarios, estados.

/src/services

Implementa lógica de negocio y llamadas a SP:

/src/middlewares

- authMiddleware: verifica JWT.
- rolesMiddleware: valida permisos.
- ErroMiddeleware maneja los errores

```
export default function authMiddleware(
    req: Request,
    res: Response,
    next: NextFunction
) {
    const authHeader = (req.headers["authorization"] ||
        req.headers["Authorization"]) as string | undefined;
    if (!authHeader) {
        res.status(401).json({ message: "Token no proporcionado" });
        return;
    }

    const parts = authHeader.split(" ");
    if (parts.length !== 2 || parts[0] !== "Bearer") {
        res.status(401).json({ message: "Formato de token inválido" });
        return;
    }

    const token = parts[1];
    try {
        const payload = jwt.verify(token as string, JWT_SECRET as string) as any;
        (req as any).user = payload;
        next();
    } catch (err) {
        res.status(401).json({
            message: "Token inválido",
            error: err instanceof Error ? err.message : err,
        });
    }
}
```

```ts
export function permit(...allowed: string[]) {
    const allowedLower = allowed.map((s) => String(s).toLowerCase());
    return (req: Request, res: Response, next: NextFunction) => {
        const user = (req as any).user;
        if (!user) {
            return res.status(401).json({ message: "No autenticado" });
        }
        const roles = extractRolesFromUser(user);
        const ok = roles.some((r) => allowedLower.includes(r));
        if (!ok) return res.status(403).json({ message: "Acceso denegado" });
        return next();
    };
}

export function deny(...denied: string[]) {
    const deniedLower = denied.map((s) => String(s).toLowerCase());
    return (req: Request, res: Response, next: NextFunction) => {
        const user = (req as any).user;
        if (!user) {
            return res.status(401).json({ message: "No autenticado" });
        }
        const roles = extractRolesFromUser(user);
        const blocked = roles.some((r) => deniedLower.includes(r));
        if (blocked) return res.status(403).json({ message: "Acceso denegado" });
        return next();
    };
}
```

```
import { Request, Response, NextFunction } from "express";
import { HttpError, isHttpError } from "../utils/httpError";

function errorHandler(
    err: unknown,
    req: Request,
    res: Response,
    next: NextFunction
) {
    if (res.headersSent) {
        return next(err);
    }

    let status = 500;
    let message = "Error interno del servidor";
    let details: any = undefined;

    if (isHttpError(err)) {
        status = err.statusCode || 500;
        message = err.message || message;
        details = err.details;
    } else if (err instanceof Error) {
        message = err.message;
    }

    console.error(
        `[ErrorHandler] ${req.method} ${req.originalUrl} -> ${status} :`,
        message,
        details || err
    );

    res.status(status).json({
        success: false,
        message,
        ...(details ? { details } : {}),
    });
}

export default errorHandler;
```

/src/config

- db.ts: ConnectionPool
- env.ts: carga de variables

Frontend

/src/pages

- Expedientes

- Detalle
- Indicios
- Reportes

/src/components

- Login
- ProtectedRoutes
- SessionTimeOut
- /src/layout
- Header
- Loyout
- Sidebar


TT_USUARIO_ROL
PK ID_USUARIO_ROL
FK1 ID_USUARIO INT NOT NULL
FK2 ID_ROL INT NOT NULL
ID_USUARIO_REGISTRO INT NOT NULL
FECHA_REGISTRO DATETIME NOT NULL
ESTADO_REGISTRO TINYINT NOT NULL
TC_ROL
PK ID_ROL
NOMBRE VARCHAR(100) NOT NULL
DESCRIPCION VARCHAR(255) NULL
ESTADO_REGISTRO TINYINT NOT NULL
TT_USUARIO
PK ID_USUARIO
USUARIO VARCHAR(50) UNIQUE NOT NULL
CLAVE_HASH VARCHAR(255) NOT NULL
NOMBRES VARCHAR(100) NOT NULL
APELLIDOS VARCHAR(100) NOT NULL
CORREO VARCHAR(100) NOT NULL
TELEFONO VARCHAR(20) NULL
NUMERO_EMPLEADO VARCHAR(20) NULL
ULTIMO_LOGIN DATETIME NULL
FK1 ID_USUARIO_REGISTRO INT NULL
FECHA_REGISTRO DATETIME NULL
FK2 ID_USUARIO_ACTUALIZACION INT NULL
FECHA_ACTUALIZACION DATETIME NULL
ESTADO_REGISTRO TINYINT NOT NULL
TC_DEPARTAMENTO
PK ID_DEPARTAMENTO
NOMBRE VARCHAR(100) NOT NULL
FK1 ID_USUARIO_REGISTRO INT NULL
FECHA_REGISTRO DATETIME NOT NULL
ESTADO_REGISTRO TINYINT NOT NULL
TC_MUNICIPIO
PK ID_MUNICIPIO
FK1 ID_DEPARTAMENTO INT NOT NULL
NOMBRE VARCHAR(100) NOT NULL
FK2 ID_USUARIO_ACTUALIZACION INT NULL
FECHA_REGISTRO DATETIME NOT NULL
ESTADO_REGISTRO TINYINT NOT NULL
TT_EXPEDIENTE
PK ID_EXPEDIENTE
FK1 CODIGO_EXPEDIENTE VARCHAR(50) UNIQUE NOT NULL
DESCRIPCION_CASO VARCHAR(500) NULL
FK2 ID_DEPARTAMENTO INT NULL
FK3 ID_MUNICIPIO INT NULL
DIRECCION VARCHAR(200) NULL
FECHA_HECHO DATETIME NULL
ID_ULTIMO_ESTADO_EXPEDIENTE INT NULL
FK4 ID_USUARIO_REGISTRO INT NOT NULL
FECHA_REGISTRO DATETIME NOT NULL
FK5 ID_USUARIO_ACTUALIZACION INT NULL
FECHA_ACTUALIZACION DATETIME NULL
ESTADO_REGISTRO TINYINT NOT NULL
TT_EXPEDIENTE_ESTADO
PK ID_EXPEDIENTE_ESTADO
FK1 ID_EXPEDIENTE INT NOT NULL
FK2 ID_ESTADO_EXPEDIENTE INT NOT NULL
FK3 ID_COORDINADOR_FISCALIA
MOTIVO_BLOQUEO VARCHAR(500) NULL
FK4 ID_USUARIO_CAMBIO INT NOT NULL
FECHA_CAMBIO DATETIME NOT NULL
ESTADO_REGISTRO TINYINT NOT NULL
TC_ESTADO_EXPEDIENTE
PK ID_ESTADO_EXPEDIENTE
NOMBRE VARCHAR(100) NOT NULL
DESCRIPCION VARCHAR(255) NULL
ID_USUARIO_REGISTRO INT NULL
FECHA_REGISTRO DATETIME NOT NULL
ESTADO_REGISTRO TINYINT NOT NULL
TT_INDICIO
PK ID_INDICIO
FK1 ID_EXPEDIENTE INT NOT NULL
NUMERO_INDICIO INT NOT NULL
DESCRIPCION VARCHAR(500) NOT NULL
FK2 ID_TIPO_INDICIO INT NOT NULL
COLOR VARCHAR(50) NULL
TAMANO VARCHAR(100) NULL
PESO VARCHAR(50) NULL
UBICACION_ENCONTRADA VARCHAR(200) NULL
CARACTERISTICAS VARCHAR(500) NULL
FK3 ID_USUARIO_REGISTRO INT NOT NULL
FECHA_REGISTRO DATETIME NOT NULL
FK4 ID_USUARIO_ACTUALIZACION INT NULL
FECHA_ACTUALIZACION DATETIME NULL
ESTADO_REGISTRO TINYINT NOT NULL DEFAULT
TC_TIPO_INDICIO
PK ID_TIPO_INDICIO
NOMBRE VARCHAR(100) NOT NULL
DESCRIPCION VARCHAR(255) NULL
FK1 ID_USUARIO_REGISTRO INT NULL
FECHA_REGISTRO DATETIME NOT NULL
ESTADO_REGISTRO TINYINT NOT NULL

8. MODELO RELACIONAL Y ER

Entidades clave:

- TT_USUARIO
- TT_USUARIO_ROL
- TC_ROL
- TT_EXPEDIENTE
- TT_INDICIO
- TT_EXPEDIENTE_ESTADO
- Catálogos: departamento, município, estados, tipo de indício, estado expediente

Principios aplicados:

- Normalización 3FN
- Auditoría simple
- Borrado lógico
- Integridad referencial total
- Historial completo de estados

9. ROLES Y PERMISOS

Tabla completa:

Acción	Admin	Técnico	Coordinador
Crear expediente	**NO**	**SI**	**NO**
Editar expediente	**NO**	**SI(Registrado/ Rechazado)**	**NO**
Ver expedientes	**SI (todos)**	**SI (propios)**	**SI (todos)**
Crear indicios	**NO**	**SI**	**NO**

Acción	Admin	Técnico	Coordinador
Editar indicios	**NO**	**SI**	**NO**
Cambiar estado	**NO**	**SI (Revisión)**	**SI**
Ver historial	**SI**	**SI**	**SI**

10. PROCEDIMIENTOS ALMACENADOS

Lista de SP implementados:

1. sp_obtener_todos_expedientes — Listado global de expedientes (incluye nombre del último estado).
2. sp_crear_expediente — Inserta un expediente y registra el estado inicial (Registrado).
3. sp_crear_usuario — Crea un usuario y devuelve su fila.
4. sp_obtener_usuario — Devuelve un usuario por id.
5. sp_obtener_usuarios — Lista usuarios activos.
6. sp_actualizar_usuario — Actualiza campos de un usuario.
7. sp_eliminar_usuario — Borrado lógico de usuario (marca estado_registro = 0).
8. sp_authenticate_usuario — Autentica usuario (devuelve usuario + roles como segundo resultset).
9. sp_update_ultimo_acceso — Actualiza ultimo_acceso de un usuario.
10. sp_obtener_expediente — Detalle de expediente por id (incluye último estado).
11. sp_actualizar_expediente — Actualiza un expediente.
12. sp_eliminar_expediente — Borrado lógico de expediente.
13. sp_obtener_expedientes_por_usuario — Lista expedientes filtrados por id_usuario_registro.
14. sp_crear_indicio — Inserta un indicio vinculado a un expediente.
15. sp_obtener_indicio — Obtiene un indicio por id.
16. sp_obtener_indicios_por_expediente — Lista indicios de un expediente.
17. sp_actualizar_indicio — Actualiza un indicio.
18. sp_eliminar_indicio — Borrado lógico de indicio.
19. sp_crear_expediente_estado — Registra una transición de estado para expediente (actualiza id_ultimo_estado_expediente).
20. sp_obtener_expediente_estado — Obtiene un registro de estado por id.

21. sp_obtener_expediente_estados_por_expediente — Lista historial de estados por expediente.
22. sp_actualizar_expediente_estado — Actualiza un registro de estado del expediente.
23. sp_eliminar_expediente_estado — Borrado lógico de registro de estado.
24. sp_obtener_estado_expediente — Lista catálogos de estados de expediente.
25. sp_obtener_departamentos — Lista departamentos.
26. sp_obtener_municipios — Lista municipios (con filtro opcional por departamento).
27. sp_obtener_tipos_indicio — Lista tipos de indicio (catálogo).
28. sp_report_summary — Reportería: conteos por estado y total de expedientes en rango de fechas.
29. sp_report_timeseries — Reportería: series temporales (transiciones por fecha y creaciones por fecha).

```sql
CREATE OR ALTER PROCEDURE dbo.sp_obtener_todos_expedientes
AS
BEGIN
    SET NOCOUNT ON;
    SELECT
        e.id_expediente,
        e.codigo_expediente,
        e.descripcion,
        e.id_departamento,
        e.id_municipio,
        e.fecha_hecho,
        e.id_usuario_registro,
        e.fecha_registro,
        e.id_usuario_actualizacion,
        e.fecha_actualizacion,
        e.estado_registro,
        e.id_ultimo_estado_expediente,
        est.nombre AS ultimo_estado_nombre
    FROM dbo.TT_EXPEDIENTE e
    LEFT JOIN dbo.TC_ESTADO_EXPEDIENTE est ON est.id_estado_expediente = e.id_ultimo_estado_expediente
    WHERE e.estado_registro = 1
    ORDER BY e.fecha_registro DESC;
END
GO
```

11. API REST – ENDPOINTS

Autenticación

- POST /api/auth/login: Login y obtención de token JWT.

Expedientes

- POST /api/expedientes: Crear un expediente.
- GET /api/expedientes: Listar todos los expedientes (uso: COORDINADOR / lista global).
- GET /api/expedientes/:id: Obtener detalle de un expediente por id.
- GET /api/expedientes/por-usuario/:id: Listar expedientes creados por un usuario (id).
- PUT /api/expedientes/:id: Actualizar expediente por id.
- DELETE /api/expedientes/:id: Borrado lógico de expediente por id.

Indicios

- POST /api/indicios: Crear un indicio.
- GET /api/indicios/:id: Obtener un indicio por id.
- GET /api/indicios/por-expediente/:id: Listar indicios de un expediente.
- PUT /api/indicios/:id: Actualizar indicio por id.
- DELETE /api/indicios/:id: Borrado lógico de indicio por id.

Usuarios

- POST /api/usuarios: Crear usuario.
- GET /api/usuarios: Listar usuarios activos.
- GET /api/usuarios/:id: Obtener usuario por id.
- PUT /api/usuarios/:id: Actualizar usuario por id.
- DELETE /api/usuarios/:id: Borrado lógico de usuario por id.

Estados de expediente (historial / transiciones)

- POST /api/expediente-estados: Crear un registro de estado (transición).
- GET /api/expediente-estados/:id: Obtener registro de estado por id.
- PUT /api/expediente-estados/:id: Actualizar un registro de estado por id.

- DELETE /api/expediente-estados/:id: Borrado lógico de registro de estado por id.
- GET /api/expediente-estados/por-expediente/:id: Listar historial de estados de un expediente.

Catálogos

- GET /api/catalogs/estados: Listar estados de expediente (TC_ESTADO_EXPEDIENTE).
- GET /api/catalogs/departamentos: Listar departamentos.
- GET /api/catalogs/municipios: Listar municipios (acepta filtro opcional id_departamento).
- GET /api/catalogs/tipos-indicio: Listar tipos de indicio.

Reportería

- GET /api/reports/summary: Resumen agregado por estado (filtros: fecha_inicio, fecha_fin, id_estado).
- GET /api/reports/timeseries: Series temporales (filtros: fecha_inicio, fecha_fin, id_estado).

12. SEGURIDAD Y AUTENTICACIÓN

- JWT firmado con clave secreta
- Expiración configurada
- Sesión TimeOut frontend
- Middlewares:
    - ahMiddleware
    - rolesMiddleware

Flujo típico:

Login → JWT → Guardado en localStorage → Uso en Authorization header

```
const JWT_SECRET = config.jwtSecret;

export default function authMiddleware(
    req: Request,
    res: Response,
    next: NextFunction
) {
    const authHeader = (req.headers["authorization"] ||
        req.headers["Authorization"]) as string | undefined;
    if (!authHeader) {
        res.status(401).json({ message: "Token no proporcionado" });
        return;
    }

    const parts = authHeader.split(" ");
    if (parts.length !== 2 || parts[0] !== "Bearer") {
        res.status(401).json({ message: "Formato de token inválido" });
        return;
    }

    const token = parts[1];
    try {
        const payload = jwt.verify(token as string, JWT_SECRET as string) as an
        (req as any).user = payload;
        next();
    } catch (err) {
        res.status(401).json({
            message: "Token inválido",
            error: err instanceof Error ? err.message : err,
        });
    }
}
```

13. PRUEBAS UNITARIAS

Estructura de Jest:

tests/

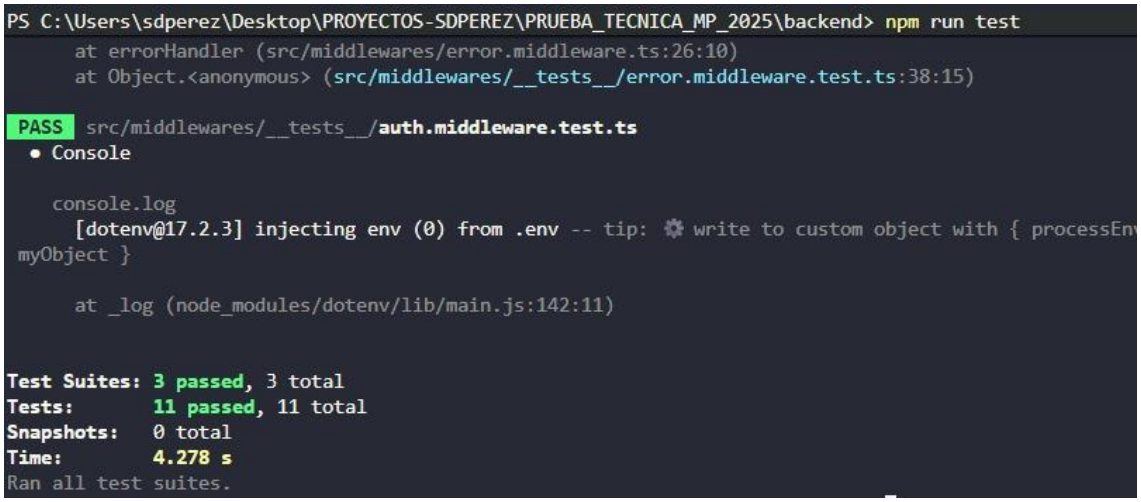
controllers/

services/

Cobertura básica:

- Login

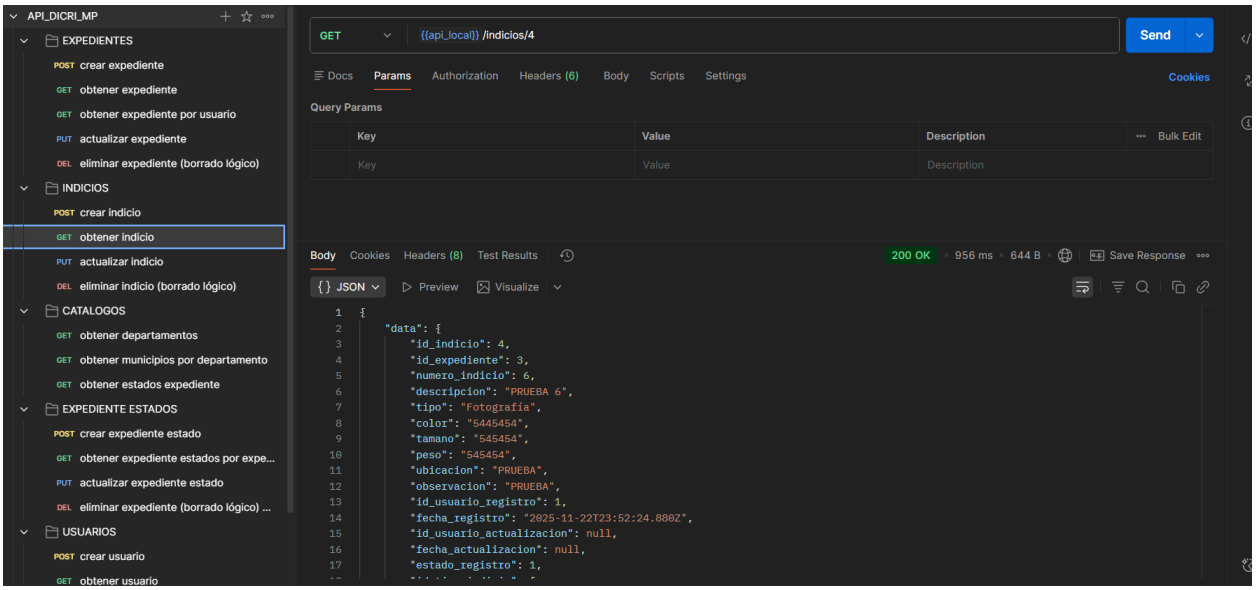
- Listado de expedientes
- Reportes
- Validación de roles



14. PRUEBAS POSTMAN

Incluye:

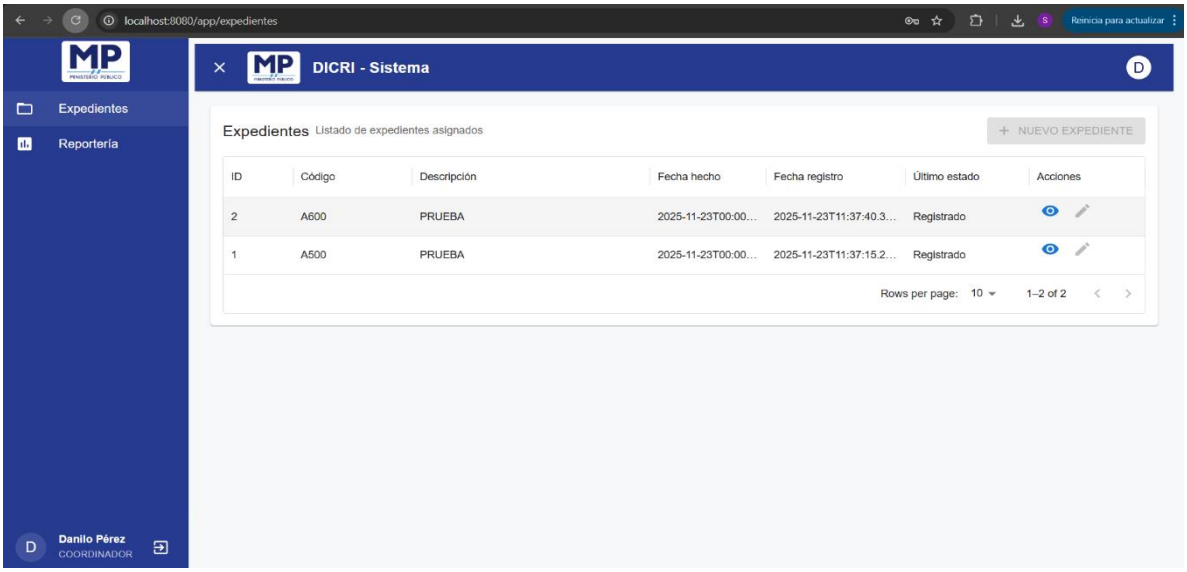
- Login
- CRUD expedientes
- CRUD indicios
- Cambios de estado
- Reportes



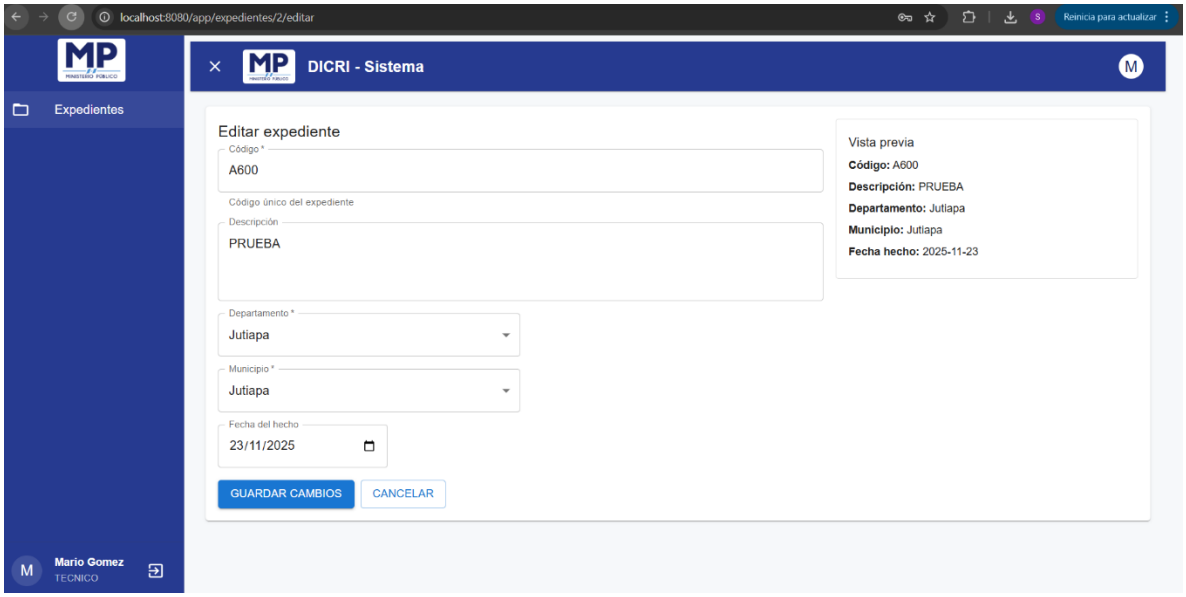
15. INTERFAZ DE USUARIO

Login:

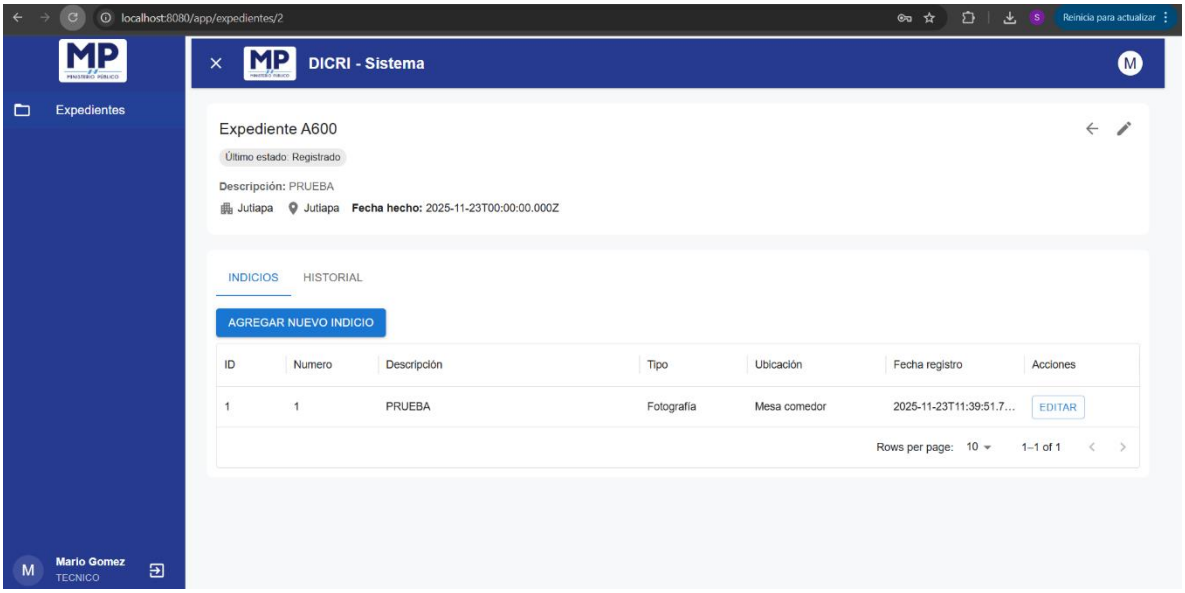
Listado de expedientes:



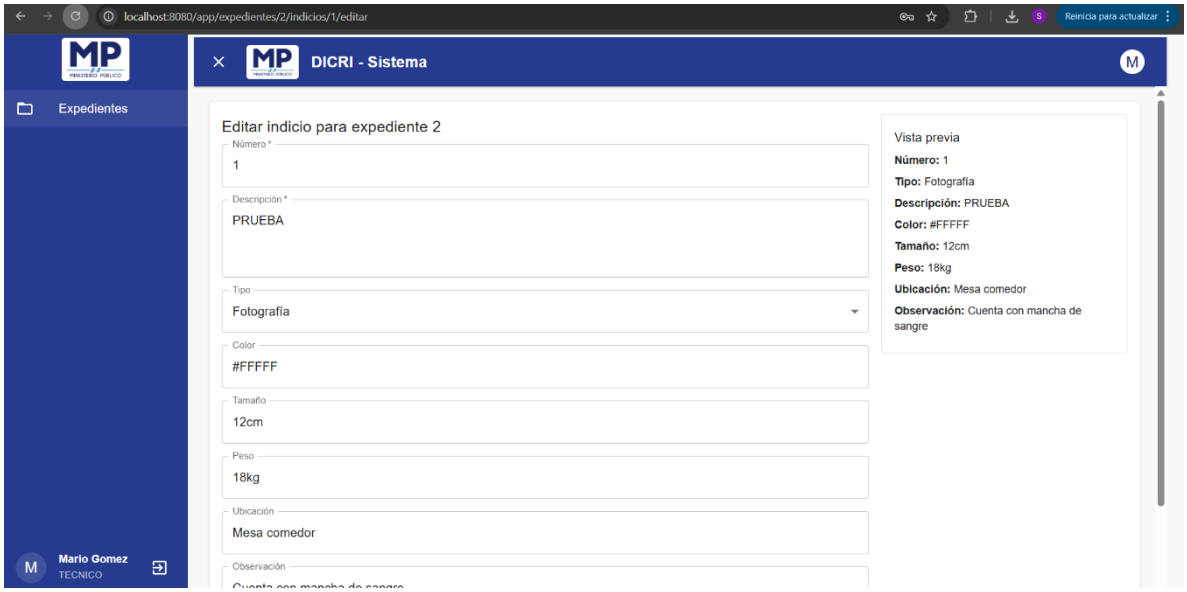
Formulario creación y edición de expediente:



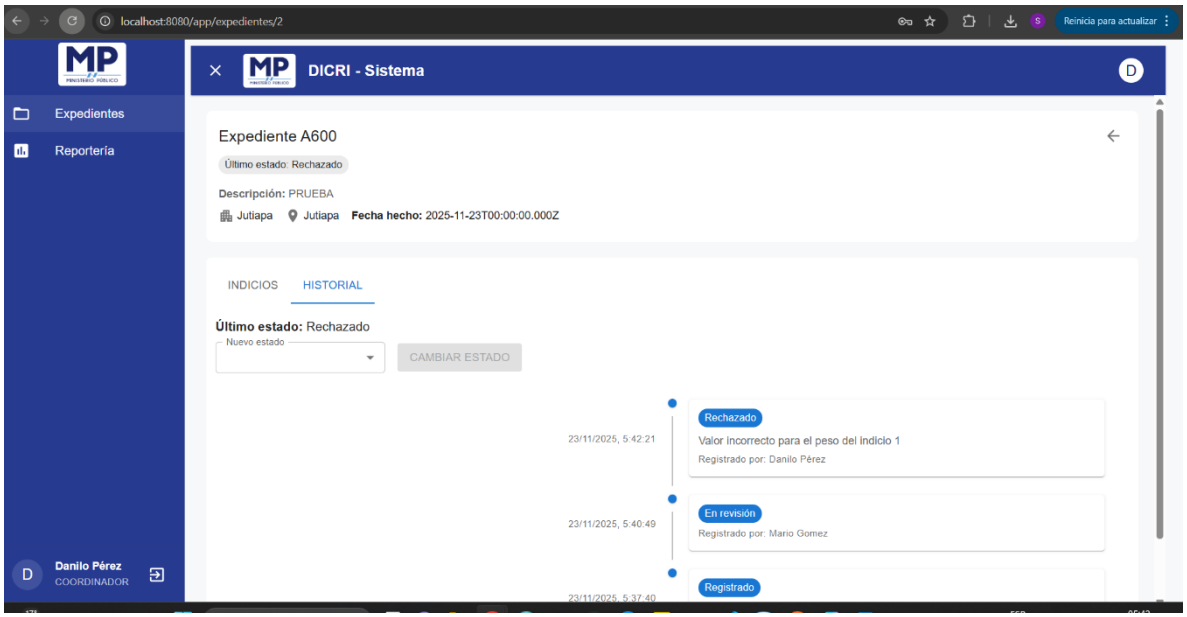
Pantalla detalle de expediente:



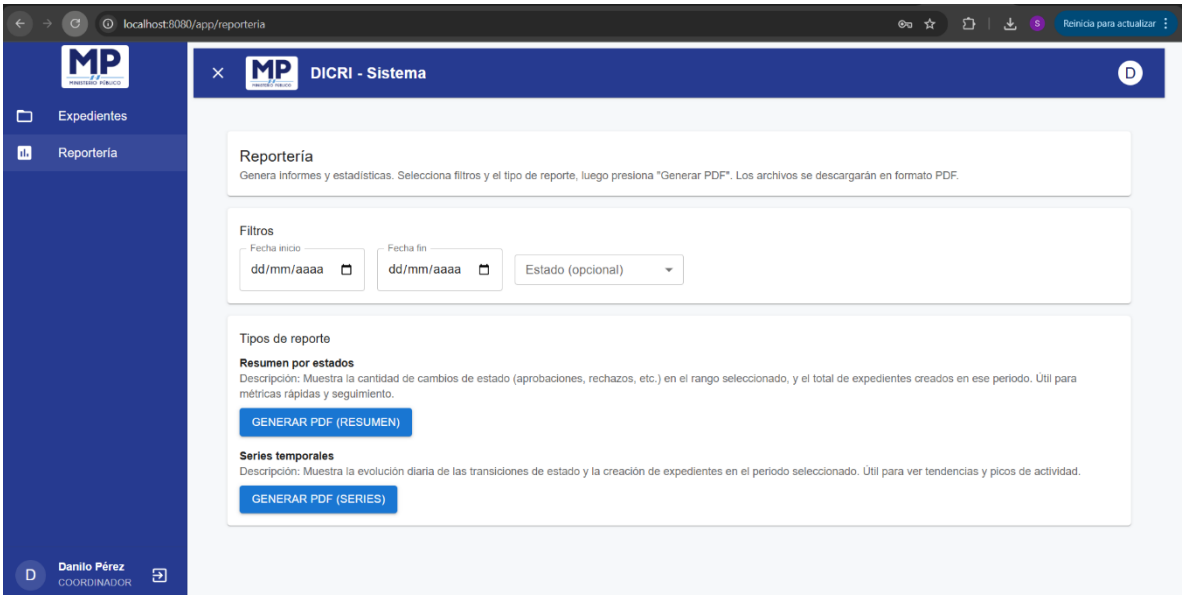
Pantalla creación y edición de indicio:



Pantalla historial de estados:



Pantalla reportes:



16. JUSTIFICACIÓN DE DECISIONES TÉCNICAS

1. Separación clara entre Frontend y Backend

Se optó por un diseño desacoplado (SPA React + API REST Node.js) para:

- mejorar mantenibilidad
- permitir despliegues independientes
- facilitar escalabilidad horizontal
- reducir acoplamiento entre presentación y lógica de negocio

Esto también permite evaluar ambos componentes por separado durante el proceso técnico.

2. Uso de Stored Procedures para encapsular lógica SQL

Los SP se eligieron por razones institucionales y técnicas:

- evitan inyección SQL
- mejoran el rendimiento mediante planes de ejecución reutilizables
- centralizan la lógica transaccional
- simplifican auditoría y mantenimiento de reglas de negocio
- reducen carga de procesamiento en el backend

En entornos del MP, los SP son estándar por temas de seguridad y gobernanza.

3. Uso de Docker para un entorno reproducible y portable

Docker garantiza que:

- la aplicación se ejecute igual en cualquier entorno
- no existan diferencias entre máquinas del evaluador
- el entorno se levante con un solo comando
- SQL Server, backend y frontend estén aislados y controlados

La inclusión de db-init asegura una carga automática y confiable del modelo de datos.

4. Autenticación mediante JWT

JWT fue elegido por:

- simplicidad en entornos sin servicios de identidad institucional
- independencia del backend
- compatibilidad con aplicaciones SPA
- facilidad para incluir roles en el payload

Permite controlar permisos de manera eficiente y flexible.

5. Uso de Tabs para organizar vistas internas

La interfaz utiliza pestañas para presentar:

- indicios asociados
- historial de estados

Esto mejora la usabilidad, reduce saturación visual y facilita flujos operativos.

6. Modal para el cambio de estado

El modal se eligió porque:

- obliga al usuario a confirmar cambios críticos
- permite capturar motivos de rechazo
- evita transiciones accidentales de estado
- se alinea al flujo institucional usado en revisiones reales

7. Sistema estricto de roles (Técnico, Coordinador, Administrador)

El control de roles es fundamental para:

- restringir acciones según función institucional
- mantener trazabilidad del sistema
- asegurar que solo Coordinadores cambien estados
- garantizar orden y responsabilidad operativa

La lógica se centraliza en middleware y SP para coherencia en toda la solución.

8. Estructura modular del backend

Se aplicó separación por capas (routes → controllers → services → data) para:

- mejorar claridad
- evitar duplicación de código
- facilitar pruebas unitarias
- permitir extensión futura del sistema

9. Reporterías optimizadas mediante consultas agregadas

Los endpoints analíticos utilizan:

- agregaciones por estado
- series temporales
- filtros por rango de fechas

Se eligió delegar estas operaciones directamente a SQL Server por eficiencia y menores tiempos de respuesta.

17. **CONCLUSIONES**

El desarrollo del sistema DICRI logra cumplir plenamente con los requisitos funcionales y técnicos planteados en la prueba, proporcionando una arquitectura

sólida, mantenible y alineada con buenas prácticas de ingeniería de software. La solución implementada integra de forma coherente un frontend moderno basado en React, un backend modular con Node.js y Express, y una base de datos SQL Server respaldada por procedimientos almacenados que garantizan integridad, rendimiento y seguridad.

El sistema implementa correctamente el flujo institucional requerido para la gestión de expedientes, incluyendo la administración de indicios, la evolución de estados, la trazabilidad de acciones y la aplicación estricta de roles (Técnico, Coordinador y Administrador).
La utilización de JWT, middleware de autorización y una estructura en capas refuerzan la seguridad y permiten extender la aplicación sin afectar su núcleo.

Asimismo, la contenerización completa mediante Docker asegura portabilidad, reproducibilidad y facilidad de despliegue, permitiendo ejecutar toda la solución con un solo comando y sin dependencias externas. Las pruebas unitarias y la reportería agregada complementan el funcionamiento general, validando la calidad del código y proporcionando herramientas prácticas para análisis operativo.

Finalmente, el diseño adoptado deja abierta la posibilidad de evolucionar el sistema hacia características más avanzadas, tales como:

- integración con sistemas judiciales existentes,
- manejo de archivos multimedia asociados a indicios,
- implementación de cadena de custodia digital,
- reportería estadística avanzada y tableros ejecutivos,
- servicios adicionales de interoperabilidad institucional.

El resultado general es una plataforma robusta, extensible y técnicamente sólida, adecuada para su uso en entornos operativos institucionales y como base para futuras mejoras.