Arquitectura del Proyecto

"SKIPUR"

Integrantes:

Chavez Oscullo Klever Enrique

Guacan Rivera Alexander David

Trejo Duque Alex Fernando

Fecha: 2025-07-08

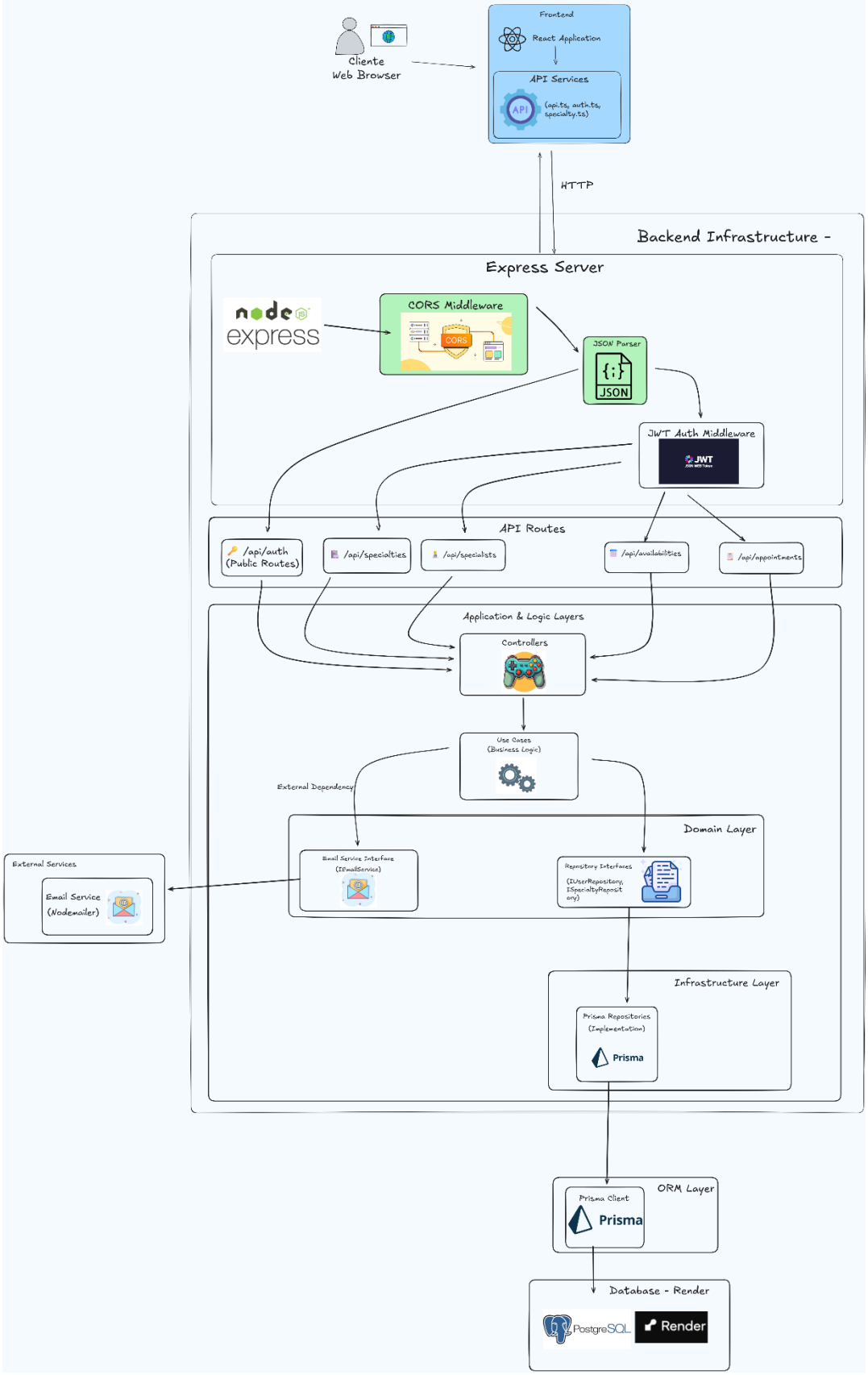
Arquitectura del Sistema SKIPUR

El presente documento detalla la arquitectura de software diseñada para el sistema de agendamiento de citas de la Fundación SKIPUR. La solución ha sido concebida siguiendo los principios de la Clean Architecture (Arquitectura Limpia), un paradigma que promueve la separación de responsabilidades, la mantenibilidad y la escalabilidad del sistema a largo plazo. El objetivo principal es aislar la lógica de negocio fundamental de los detalles de implementación, como la base de datos o el framework web, permitiendo así una mayor flexibilidad y facilidad para realizar pruebas.

SKIPUR

Fundación Carlitos

Realizado por Grupo 7



Explicación de cada Capa Arquitectónica

La arquitectura del sistema se descompone en varias capas lógicas y físicas, cada una con una responsabilidad claramente definida, tal como se visualiza en el diagrama arquitectónico.

1. **Capa de Cliente (Client & Frontend):**
   Esta es la capa de presentación con la que el usuario final interactúa directamente. Se compone de un navegador web estándar que ejecuta una aplicación de página única (SPA) desarrollada con **React**. Su principal responsabilidad es renderizar la interfaz de usuario, gestionar el estado local y comunicarse con el backend a través de un servicio de API centralizado. Este servicio encapsula todas las llamadas HTTP (vía HTTPS para garantizar la seguridad) a los endpoints del servidor, manejando las peticiones y respuestas de forma asíncrona.

2. **Capa de Servidor Backend (Backend Infrastructure):**
   Esta capa representa el núcleo del sistema y está construida sobre el runtime de **Node.js** y el framework **Express.js**. Una petición entrante atraviesa un pipeline de middlewares antes de llegar a la lógica de negocio. Primero, el middleware de **CORS** se asegura de que solo los dominios autorizados (como el del frontend) puedan realizar peticiones. A continuación, el **JSON Parser** transforma el cuerpo de las peticiones entrantes en formato JSON a objetos JavaScript utilizables. Subsecuentemente, para las rutas protegidas, el **JWT Auth Middleware** intercepta la petición, valida el token de autenticación presente en los encabezados y, si es válido, extrae la información del usuario (ID y rol) para que esté disponible en las capas subsiguientes.

3. **Capa de Aplicación y Lógica (Application & Logic Layers):**
   Una vez que la petición es validada y enrutada, ingresa al corazón lógico del sistema. Esta sección está meticulosamente organizada siguiendo la Clean Architecture:

   - **Controllers:** Son el primer punto de entrada después del enrutador. Su única función es recibir la petición HTTP, validar los datos de entrada (parámetros, cuerpo) y orquestar la ejecución del caso de uso correspondiente, sin contener lógica de negocio.

   - **Use Cases (Casos de Uso):** Representan las acciones de negocio específicas que el sistema puede realizar (ej. "Registrar un Especialista", "Reservar una Cita"). Orquestan el flujo de datos y llaman a las interfaces del dominio para interactuar con la persistencia u otros servicios. Son completamente agnósticos a Express y a la base de datos.

   - **Domain Layer (Capa de Dominio):** Es el núcleo más interno y puro de la arquitectura. Contiene las **Interfaces de Repositorio y Servicios** (ej. IUserRepository, IEmailService), que actúan como contratos que definen *qué* se puede hacer, pero no *cómo*. También alberga las entidades y reglas de negocio fundamentales. Esta capa no tiene dependencias externas.

   - **Infrastructure Layer (Implementación de Repositorios):** Aquí es donde los contratos del dominio cobran vida. Las clases como PrismaSpecialistRepository **implementan** las interfaces del dominio utilizando una tecnología concreta, en este caso, **Prisma**.

4. **Capa de ORM y Base de Datos:**
    - **ORM Layer (Prisma Client):** Actúa como un puente seguro y tipado entre la lógica de la aplicación y la base de datos. Traduce las llamadas a métodos en JavaScript a consultas SQL optimizadas.
    - **Database (Render/PostgreSQL):** La capa de persistencia final. Se trata de una base de datos relacional **PostgreSQL**, alojada en la plataforma en la nube **Render**, que garantiza la integridad, disponibilidad y seguridad de los datos.
5. **Servicios Externos:**
   Para funcionalidades que no son parte del núcleo del negocio, como el envío de notificaciones, el sistema se comunica con servicios externos a través de interfaces. Actualmente, se ha integrado un **Email Service** cuya implementación concreta utiliza **Nodemailer** para enviar correos transaccionales, como las notificaciones de bienvenida.

Funcionalidades y Resumen de Operaciones CRUD

Desde la perspectiva del usuario, el sistema ofrece un conjunto de funcionalidades diseñadas para facilitar la gestión y agendamiento de citas. El acceso a estas funcionalidades está regulado por un sistema de roles. Las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) se aplican sobre las principales entidades del sistema:

- Especialidades: Los ADMIN tienen control total (CRUD completo). Todos los usuarios autenticados (CLIENTE, ESPECIALISTA, ADMIN) pueden leer la lista de especialidades activas.
- Especialistas: La gestión completa es exclusiva de los ADMIN. Esto incluye la creación de sus perfiles de usuario, actualización de datos y desactivación (borrado lógico).
- Disponibilidad Horaria: Tanto ADMIN como ESPECIALISTA pueden crear, actualizar y eliminar bloques de horario. Un especialista solo puede gestionar su propia disponibilidad, mientras que un administrador puede gestionar la de cualquiera.
- Citas (Appointments): Los CLIENTE son los únicos que pueden iniciar el proceso de creación (reserva) de una cita. También pueden consultar sus propias citas. La actualización y cancelación son flujos de negocio más complejos que se manejarán con endpoints específicos.

Arquitectura Básica: Flujo de "Registrar un Nuevo Especialista"

Para ilustrar cómo estas capas colaboran, se detalla el flujo completo de una de las operaciones más representativas: la creación de un nuevo especialista por parte de un administrador.

El proceso se inicia cuando un administrador, autenticado en la plataforma frontend, completa un formulario con los datos del nuevo especialista y hace clic en "Guardar". En consecuencia, el frontend dispara una petición POST al endpoint /api/specialists.

1. Recepción y Middlewares (Capa de Servidor): La petición llega al servidor Express.js. Pasa a través de los middlewares de CORS y JSON Parser. Inmediatamente después, el authMiddleware intercepta la petición. Extrae el token JWT del encabezado Authorization, lo verifica con la clave secreta y confirma que el rol contenido en el token es ADMIN. Si la autenticación y autorización son exitosas, adjunta los datos del usuario administrador a la petición y pasa el control al siguiente nivel.

2. Enrutamiento y Controlador (Capa de Aplicación): El enrutador de Express, definido en specialist.routes.ts, dirige la petición al método create de la clase SpecialistController. Este controlador, cuya única responsabilidad es la orquestación, valida los datos del cuerpo de la petición (req.body) usando un esquema de Zod (CreateSpecialistSchema) para asegurar que todos los campos requeridos estén presentes y tengan el formato correcto.

3. Ejecución del Caso de Uso (Capa de Aplicación): El controlador invoca el método execute de la instancia CreateSpecialistUseCase. Este caso de uso es el cerebro de la operación. Primero, llama al método findByEmail de la interfaz IUserRepository para asegurarse de que no exista otro usuario con el mismo correo. A continuación, genera una contraseña aleatoria y segura y la hashea utilizando bcrypt. Con todos los datos listos, invoca el método create de la interfaz ISpecialistRepository, pasándole la información del nuevo especialista y la contraseña hasheada.

4. Interacción con la Infraestructura y Servicios Externos: En este punto, la inversión de dependencias entra en acción. El caso de uso no sabe qué base de datos o servicio de correo se está utilizando, solo conoce los contratos.

    a. La llamada a ISpecialistRepository.create es resuelta por la clase PrismaSpecialistRepository. Esta clase utiliza el cliente de Prisma para construir y ejecutar una consulta INSERT en la base de datos PostgreSQL, creando el registro del nuevo User con rol ESPECIALISTA y su perfil asociado en la tabla Specialist.

    b. Una vez que el repositorio confirma la creación exitosa del usuario, el caso de uso CreateSpecialistUseCase procede a llamar al método sendSpecialistWelcomeEmail de la interfaz IEmailService.

    c. Esta llamada es resuelta por la clase NodemailerService, la cual se conecta al servidor SMTP configurado (Gmail) y envía el correo de bienvenida con la contraseña autogenerada al nuevo especialista.

5. Retorno y Respuesta: El flujo regresa hacia arriba. El caso de uso devuelve el objeto del nuevo especialista (sin la contraseña) al controlador. El SpecialistController finalmente construye una respuesta HTTP exitosa con un código de estado 201 Created y el cuerpo en formato JSON, enviándola de vuelta al frontend. El administrador recibe una confirmación visual, y el nuevo especialista recibe sus credenciales por correo, completando el ciclo de manera exitosa y segura.