ChatGPT

Informe de mejora del proyecto CRM Odontológico

1. Buenas prácticas de programación

Para mejorar la calidad del código, es fundamental adoptar **buenas prácticas de programación** que faciliten el mantenimiento y escalabilidad. Algunas recomendaciones clave son:

- **Consistencia en estilo y nombres:** Utilizar convenciones uniformes para nombrar archivos, variables y componentes. Por ejemplo, nombrar los componentes React con **PascalCase** y archivos que coincidan con el nombre del componente (actualmente hay componentes en minúsculas con guiones como `calendar-header.jsx`, lo cual convendría renombrar a `CalendarHeader.jsx`). Mantener un idioma coherente en el código (idealmente inglés para nombres de variables y componentes) ayuda a la claridad.

- **Componentes pequeños y enfocados:** Evitar componentes gigantes con demasiada lógica. Cada componente debe encargarse de una cosa específica (mostrar una UI simple, o gestionar cierta lógica, pero no ambas a la vez). Si un componente crece mucho o hace varias tareas, conviene separarlo en subcomponentes o en lógica externa. Esto mejora la legibilidad y reutilización del código. Como señala un recurso, *"cada módulo o componente debe hacer una sola cosa y hacerla bien"* [1].

- **Separación de responsabilidades (Separation of Concerns):** En React es recomendable **separar la lógica de datos de la presentación**. Por ejemplo, las llamadas a APIs deben residir en servicios o hooks especializados, la gestión de estado en hooks o contextos, y la parte visual en componentes puros [2]. Esto significa que el código de **fetch** de datos, manejo de formularios, etc., debe estar aislado de los componentes visuales. Más adelante profundizaremos en este tema con patrones de arquitectura.

- **Uso de hooks personalizados y utilidades:** Identificar lógica repetitiva (por ejemplo, obtener datos de una API, gestionar un formulario, etc.) y extraerla en *custom hooks* o funciones utilitarias. En el proyecto, por ejemplo, varias páginas realizan llamadas a servicios en un `useEffect` similar; esto podría unificarse en un hook (`usePatients`, `useDoctors`, etc.) que provea los datos y estados de carga/error, reduciendo duplicación. Así los componentes de UI permanecen más simples, como sugiere la literatura: *"en un componente de formulario, crea un hook* `useFormValidation` *para la validación y requests API, permitiendo que el componente de UI se concentre en la presentación"* [3].

- **Estandares de código y linting:** Aprovechar la configuración de ESLint (ya presente en el repositorio) para asegurar un estilo de código consistente. Integrar Prettier u otra herramienta de formateo automático puede ayudar a mantener el formato (indentación, comas, comillas) uniforme en todo el proyecto. Un código consistente es más fácil de leer y menos propenso a errores.

- **Eliminación de código muerto y debugging:** Remover fragmentos comentados o console.logs de depuración que ya no sean necesarios. Por ejemplo, hay un `console.log` en `SchedulePage` para el estado inicial de modales; una vez probada la funcionalidad, es mejor eliminarlo para limpiar el código. Cada archivo debe contener solo el código relevante en producción.

- **Comentarios claros y útiles:** Incluir comentarios **solo cuando aporten valor**, explicando el *porqué* de alguna decisión no obvia. Evitar comentarios redundantes que expliquen el *qué* (eso debe desprenderse del código limpio). Un comentario podría justificar una fórmula compleja (ej. cálculo de slots horarios), pero el código también debería ser autoexplicativo con nombres descriptivos.

- **Tipado y validación de props:** Actualmente se usan *JSDoc* para definir tipos de datos (por ejemplo `@typedef AppointmentData` en `DentistColumn`). Esto es positivo, pero se podría ir un paso más allá usando TypeScript para tipar todo el proyecto, o al menos PropTypes en cada componente público, para asegurar que reciben las props correctas. Un tipado estático ayuda a atrapar errores temprano y sirve de documentación del contrato de cada componente.

En resumen, aplicar estas prácticas hará el código más **legible, consistente y fácil de mantener**, sentando las bases para aplicar los principios de arquitectura descritos a continuación.

2. Código limpio (Clean Code)

El concepto de **Código Limpio** se refiere a escribir código claro, sencillo y libre de elementos innecesarios o confusos. En React, esto es especialmente importante porque trabajamos intensivamente con JavaScript en la UI. Un código limpio asegura que el proyecto sea fácil de entender y modificar por cualquier desarrollador [4]. A continuación se listan algunas pautas de Clean Code aplicables al repositorio:

- **Legibilidad ante todo:** Priorizar que el código se lea casi como prosa. Esto implica usar nombres descriptivos para variables, funciones y componentes. Por ejemplo, `appointments` es claro, mientras que nombres genéricos como `data` o `list` dificultan entender de qué lista se trata. Igualmente, en lugar de comentarios excesivos, lograr que la intención del código se deduzca de su estructura y nombres.

- **Evitar números mágicos y valores duplicados:** En el código actual, vemos números "mágicos" como `25`, `+2` o `14` en la lógica de los slots horarios (`DentistColumn.jsx`). Estos deberían reemplazarse por constantes con nombres semánticos. Por ejemplo:

  ```
  const START_HOUR = 8;
  const TOTAL_SLOTS = 24; // 8:00 to 20:00 in half-hour increments
  const LUNCH_SLOT_INDEX = 14; // índice donde comienza el break (15:00)
  ```

  De este modo, si cambia el horario de trabajo o la lógica de compensación, se ajusta la constante en un solo lugar y el código gana claridad.

- **Funciones y métodos concisos:** Siguiendo a Robert C. Martin, una función debería ser corta y hacer solo una tarea. Si en un componente tenemos un método muy largo (por ejemplo, una función que verifica datos, luego formatea fechas y finalmente renderiza JSX), conviene separarla

en funciones más pequeñas o mover parte de esa lógica a un helper. Esto facilita pruebas unitarias y reuse. En React, esto también aplica a los componentes funcionales: si un componente se vuelve demasiado grande, considerar dividirlo.

- **No repetir lógica (evitar code smells):** Diversos *code smells* comunes son código duplicado, componentes fuertemente acoplados o funciones que hacen de todo. En la sección de DRY profundizaremos sobre duplicación. Otro ejemplo de code smell es tener un componente que gestione múltiples cosas: por ejemplo, un componente de formulario que también valida y también envía datos es tres responsabilidades en uno (difícil de mantener). El código limpio sugiere separarlos (un componente para UI del formulario, una función/servicio para la validación, otra para el envío). Esto sigue el principio de responsabilidad única: *"Un componente que renderiza un formulario no debería ser también responsable de manejar el envío del mismo… son dos responsabilidades diferentes y violaría SRP"* [5] . En general, si notamos que un módulo hace varias cosas, es señal de refactorizar.

- **Eliminar lo innecesario:** Código limpio implica eliminar cualquier rastro de código sin uso. Por ejemplo, en el repositorio aparecen duplicados `Sidebar.jsx` y `Topbar.jsx` en dos ubicaciones (`src/components` y `src/components/layout`). Tras decidir cuál versión conservar, se debe eliminar la duplicada para evitar confusión. Igualmente, borrar importaciones no utilizadas, variables definidas pero nunca leídas, etc. Mantener el repositorio libre de "ruido" acelera la comprensión.

- **Formato consistente:** Asegurarse de que la indentación, espacios y saltos de línea sean uniformes. Por ejemplo, en JSX abrir y cerrar etiquetas con la misma indentación mejora la legibilidad. Usar comillas simples vs dobles consistentemente (el linter/formatter puede hacer esto automáticamente). Estos detalles hacen el código más prolijo.

En síntesis, un código limpio **facilita la colaboración y reduce la probabilidad de errores**. Como indica la literatura, en React es vital porque una base de JavaScript limpia hace que todo el sistema sea más comprensible [4] . Adoptando estas prácticas de claridad, estaremos preparados para implementar principios de diseño más avanzados (SOLID, DDD, MVP) de forma efectiva.

3. Aplicación de los principios SOLID

Los principios **SOLID** son cinco guías de diseño que ayudan a crear software mantenible y extensible. A continuación, detallamos cada principio y cómo aplicarlo en el contexto de este proyecto React:

- **S – Principio de Responsabilidad Única (SRP):** Cada componente, clase o módulo debe tener **una única responsabilidad** o propósito. En React esto significa que un componente debería encargarse *solo de una cosa*. Por ejemplo, un componente de lista de pacientes muestra la lista, pero la lógica de filtrado/búsqueda podría estar en otro nivel (en un hook o en un componente superior), y la obtención de datos en un servicio aparte. Actualmente, páginas como `SchedulePage.jsx` manejan lógica de datos (fetch de citas y doctores), estado de modales y renderizado de calendario, todo junto. Aplicando SRP, separaríamos esas tareas: la lógica de datos pasaría a un **presentador** o hook (ver MVP más abajo), y el componente de vista solo presentaría los datos ya preparados. Esto mejora la reutilización (podríamos reutilizar la lógica de citas en otro contexto, como un widget diferente) y el mantenimiento. **Cada módulo o componente debe hacer una sola cosa y hacerla bien** [1] , así evitamos efectos colaterales. Un ejemplo concreto: actualmente se define un sub-componente `DoctorAvatar` dentro de `DoctorsPage.jsx` para mostrar iniciales de doctor con ciertos colores. Si en otro lugar se

necesitara un avatar similar, habría que duplicar esa lógica. Siguiendo SRP, conviene extraerlo como componente reutilizable (`<DoctorAvatar>` independiente), cuya única razón de cambio sea cambiar la forma de mostrar avatares, no la lógica de la página entera.

- **O – Principio de Abierto/Cerrado (OCP):** Una entidad de software debe estar **abierta para su extensión pero cerrada para su modificación**. En la práctica, esto significa diseñar el código de forma que agregar una nueva funcionalidad requiera agregar nuevo código, no editar el existente en múltiples lugares. ¿Cómo se aplica en este proyecto? Por ejemplo, supongamos que en el futuro se quiere añadir un nuevo rol (además de pacientes y doctores). Siguiendo OCP, deberíamos poder agregar nuevos componentes de vista y quizás nuevos servicios para ese rol sin tener que modificar la estructura básica de la aplicación (ruteo, layout común, etc.). Esto se logra centralizando la configuración y usando estructuras genéricas. Un caso concreto: la barra lateral (Sidebar) lista secciones como Home, Pacientes, Doctores, Calendario, etc. Si está hardcodeada, agregar una sección nueva implicaría editar el componente. Mejor sería que `Sidebar` construya su menú a partir de una lista de opciones (configuración). Así, para extender el menú se añade un ítem a la lista (extensión) en vez de tocar el código de `Sidebar` (modificación). Igualmente, conviene utilizar componentes genéricos y composición: por ejemplo, tener un componente genérico `Modal` (ya lo hay) que se pueda extender pasando distintas props, en vez de crear modales específicos con estructura duplicada. Resumiendo, el sistema debe permitir **crecer** (nuevas vistas, nuevos tipos de dato) causando el mínimo impacto en el código existente.

- **L – Principio de Sustitución de Liskov (LSP):** Este principio (originalmente sobre herencia) indica que las clases derivadas deben poder usarse como la clase base sin romper la aplicación. En React, donde no se usa herencia de componentes comúnmente, LSP se traduce a asegurarnos de que **las abstracciones se cumplan**. Por ejemplo, si definimos una interfaz o contrato para un servicio (digamos una interfaz `AppointmentService` con métodos `fetchAppointments`, `createAppointment`, etc.), cualquier implementación de esa interfaz (ya sea un servicio real que llama a API REST, o un servicio simulado de datos locales) debería poder sustituirse y la aplicación seguir funcionando. LSP nos guía a diseñar componentes y funciones con **expectativas claras**: un componente con ciertas props debe comportarse correctamente con cualquier valor válido de esas props. En la práctica, esto anima a evitar suposiciones frágiles en el código. Un ejemplo sencillo: un componente de `InputTexto` reusable no debería asumir que siempre recibe un valor no vacío; debe funcionar igual con valor vacío (posible en un formulario recién abierto). Cumplir LSP es más relevante en código de lógica (por ejemplo, nuestras funciones de servicio deberían poder usar diferentes fuentes de datos sin cambiar su contrato). Aunque React no use mucha herencia, podemos pensar en LSP al utilizar *polimorfismo por composición*: por ejemplo, un componente `<List items={...} renderer={ItemComponent} />` debería aceptar cualquier `ItemComponent` que cumpla la interfaz esperada (p. ej., un componente que reciba la data del item como prop). Si diseñamos esas abstracciones correctamente, podemos intercambiar implementaciones sin problemas.

- **I – Principio de Segregación de Interfaces (ISP):** Indica que es mejor tener **interfaces específicas y pequeñas** en lugar de interfaces monolíticas. En nuestro contexto, significa no forzar a un componente o módulo a depender de props o funcionalidades que no usa. Por ejemplo, evitar pasar un objeto enorme de propiedades a un componente cuando este solo necesita dos o tres de ellas; es preferible pasar solo las necesarias o dividir el componente en varios más simples. En los servicios, en lugar de una sola clase `ApiService` que tenga métodos para pacientes, doctores, citas, etc., podemos tener servicios separados (`PatientService`, `DoctorService`, etc.), segregando las interfaces por dominio. De hecho, el repositorio ya apunta hacia eso con archivos separados, lo cual es bueno. Aplicando ISP,

podríamos definir pequeñas interfaces o contratos para ciertas funcionalidades: por ejemplo, una interfaz `ICalendarView` que exponga solo las operaciones que la vista de calendario necesita (siguiente día, anterior día, cambiar vista semanal/diaria). Así, la vista de calendario no depende de más de lo necesario. En React functional, ISP también se refleja en la creación de **hooks especializados**: mejor tener un hook `usePatientData()` que retorna justo lo pertinente a pacientes, y otro `useDoctorData()` para doctores, que uno genérico que mezcle lógica de todos (lo cual se volvería difícil de mantener). En resumen, **divide las interfaces grandes en partes más específicas**, de modo que cada parte del código conozca solo lo que necesita y nada más.

- **D – Principio de Inversión de Dependencias (DIP):** Este principio sugiere que los módulos de alto nivel no deben depender de módulos de bajo nivel directamente, sino de abstracciones. Es decir, **invertir las dependencias** definiendo interfaces abstractas y haciendo que las implementaciones concretas las satisfagan. En nuestra aplicación, esto se traduce a desacoplar los componentes/logic de la fuente de datos concreta. Actualmente, las páginas llaman directamente a funciones que internamente usan `axios` (`getAppointments` en `api.js`). Para aplicar DIP, podríamos definir por ejemplo una abstracción `AppointmentRepository` o `IAppointmentService` que exponga métodos como `fetchAll()`, `create(app)`, `update(app)`. La página o su presentador consumirían esa abstracción, sin importar si debajo usa Axios, Fetch API u otra cosa. Luego proveeríamos una implementación concreta (por ejemplo `RestAppointmentService` que implementa esa interfaz usando llamadas HTTP). Esta inversión permite, por ejemplo, cambiar de una API REST a otra fuente (o datos mock) **sin cambiar el código de la página**, solo proporcionando otra implementación de la interfaz. También facilita pruebas unitarias: se puede inyectar una implementación falsa que devuelva datos predecibles para verificar la lógica. Como menciona un experto: *"siguiendo principios SOLID, además de la clase de flujo principal, tendrías interfaces que definen lo que deben hacer sus dependencias (fetchers, submitters), e inyectas esas dependencias pasando instancias al constructor"* [6]. En nuestro caso, el "flujo principal" podría ser un **presentador** (ver MVP) y las dependencias inyectadas los servicios de datos. En general, DIP nos anima a programar contra **interfaces** y no contra detalles concretos. Aunque JavaScript no tiene interfaces formales, podemos lograrlo mediante patrones de diseño (pasar funciones como props, usar contextos para proveer implementaciones, etc.). Un ejemplo práctico: definir una interfaz para el servicio de calendario:

```
// Definición de contrato (puede ser JSDoc typedef o TypeScript
interface)
/**
 * @interface IAppointmentService
 * Métodos para gestionar citas
 */
class IAppointmentService {
  getAllAppointments() {}
  createAppointment(data) {}
  // ...
}
```

Y hacer que nuestro servicio real la implemente:

```
class AppointmentServiceAPI /* implements IAppointmentService */ {
  async getAllAppointments() {
    return await axios.get('/api/appointments');
  }
  // ...
}
```

Luego en el presentador (o donde corresponda) usamos `appointmentService = new AppointmentServiceAPI()` pero referenciado como `IAppointmentService`. Esto suena abstracto, pero el beneficio es poder sustituir fácilmente `AppointmentServiceAPI` por, digamos, `AppointmentServiceMock` que devuelve datos de prueba, sin que el resto de la app lo note. De hecho, un desarrollador recomienda *"mover la mayor cantidad de lógica posible fuera de los componentes React y a clases independientes... para poder probarla en aislamiento"* [7] – DIP facilita esto al permitirnos desconectar la lógica de detalles como llamadas de red.

En resumen, aplicar SOLID en todo el repositorio fortalecerá la arquitectura: **componentes más simples (SRP), sistema extensible (OCP), contratos claros (LSP, ISP) y bajo acoplamiento mediante abstracciones (DIP)**. Esto nos prepara para reorganizar el proyecto con DDD y MVP, manteniendo estos principios como guías.

4. Aplicación de Domain-Driven Design (DDD)

El **Diseño Guiado por el Dominio (DDD)** propone estructurar el código según las *áreas de negocio o dominio* de la aplicación, reflejando las reglas y conceptos del mundo real que maneja. En este CRM odontológico, podemos identificar dominios claros: **Pacientes, Doctores, Citas (Agenda)**, y posiblemente otros como administración, facturación, etc. Aplicar DDD en el frontend implica:

- **Agrupar por contexto de negocio:** En lugar de organizar solo por tipos técnicos (componentes, páginas, servicios separados), conviene reorganizar el proyecto agrupando todo lo relacionado a un mismo dominio funcional. Por ejemplo, todo lo referente a pacientes (lista de pacientes, detalle de paciente, formulario de nuevo paciente, servicio de pacientes, componentes de paciente como `PatientCard`) estaría bajo una carpeta o módulo común "patients". Lo mismo para doctores, citas, etc. Esto facilita encontrar y modificar funcionalidades completas. Organizar archivos por **feature/domain** está alineado con DDD y promueve modularidad [8].

- **Modelo de dominio explícito:** Definir las **entidades y valores del dominio** de forma explícita en el código. Por ejemplo, una entidad **Paciente** con sus atributos (nombre, edad, historial, etc.), **Doctor** (nombre, especialidad, etc.), **Cita** (fecha/hora inicio, fin, paciente, doctor, tratamiento, estado, etc.). En JavaScript podemos representarlo con clases o simplemente con estructuras de objetos bien definidas (idealmente usando TypeScript para tiparlas). Actualmente se usan objetos JSON para estas entidades, pero podríamos llevarlo un paso más allá: por ejemplo, crear una clase `Appointment` con métodos que encapsulen lógica de negocio como calcular duración, formatear hora, verificar solapamiento con otra cita, etc. Si no se quieren clases, al menos centralizar esas funciones de negocio en módulos de lógica dentro del dominio correspondiente (por ejemplo, un helper `appointmentUtils.js` con funciones como `calculateSlots(appointment)`).

- **Reglas de negocio en el frontend:** Aunque muchas reglas de negocio se manejan en el backend, en la capa de presentación también hay lógica de dominio que debe respetar

conceptos del negocio. Por ejemplo, la lógica de “una cita ocupa múltiples timeslots” o “un doctor no está disponible después de cierta hora” son reglas del dominio de agenda. DDD sugiere que este tipo de lógica se implemente de forma coherente con el lenguaje del dominio. En la práctica, podríamos crear en el dominio de citas una función o método que dado `startTime` y `endTime` calcule cuántos slots ocupa la cita y cuáles índices de slot abarca. De hecho, en el código actual vemos que `DentistColumn.jsx` calcula `startSlot` y `endSlot` para cada cita y rellena un array de slots. Esa lógica pertenece claramente al *dominio de agenda*. Para mejorarla:

- Se podría mover ese cálculo a un **servicio de dominio** (por ejemplo, `AppointmentService` o un helper en `appointmentService.js` podría tomar la cita con hora de inicio/fin y devolverla enriquecida con `startSlot` y `endSlot`). De esta manera, el componente de vista no calcula nada, solo presenta slots ya preparados.

- Alternativamente, se puede encapsular en un componente de más alto nivel: por ejemplo, un componente `<AppointmentCard>` que reciba `appointment` con inicio y fin, y sea capaz de renderizarse con la altura o número de celdas adecuado. Este componente interno manejaría la regla de cuántos pixels o celdas de alto representa la cita. Así se aísla la lógica en un lugar y no se duplica en varios.

- **Capas claras dentro de cada dominio:** DDD propone separar las capas de **Dominio, Aplicación, Infraestructura y Presentación**. En el frontend podemos mapear:

- *Dominio:* las entidades (datos) y lógica pura de negocio (funciones que operan sobre esos datos, sin preocuparse de UI ni de detalles de infraestructura).
- *Aplicación:* la orquestación de casos de uso; en nuestro contexto serán los **Presenters** o controladores que coordinan acciones entre la UI, los servicios y el dominio.
- *Infraestructura:* detalles técnicos como llamadas HTTP, almacenamiento, etc. (aquí serían nuestros archivos de servicios que llaman al backend con Axios).
- *Presentación:* la interfaz de usuario (componentes React) que muestra datos y captura eventos. Idealmente, esta capa es la más “tonta” y no contiene lógica de negocio, solo lógica de presentación (mostrar u ocultar elementos, manejar interacciones inmediatas del usuario). *“La UI es la parte más tonta de la aplicación. Su lógica se limita a crear una interfaz visual llamativa para el usuario”* [9] .

Reorganizando el proyecto según DDD, cada dominio tendría dentro de sí sus subcapas: por ejemplo, el dominio **appointments** podría tener: - Un **modelo** o definición de la entidad Appointment. - Un **servicio de infraestructura** (API) para obtener/guardar citas (`appointmentService.js` actual). - Un **presentador/controlador** que implementa casos de uso de agenda (por ejemplo, lógica de agendar una cita nueva, eliminar cita, cambiar de vista día/semana). - Componentes de **presentación específicos** de ese dominio (por ejemplo, el calendario de citas, tarjeta de cita, columna de doctor, etc.).

Esta separación hace que, si en el futuro se expande la funcionalidad de *agenda* (por ejemplo, soportar eventos recurrentes, o distintos tipos de cita), todo esté confinado en su módulo de dominio sin afectar otros módulos como pacientes o doctores. Cada **bounded context** (contexto delimitado) mantiene su lógica aislada.

Por último, DDD también promueve usar el **lenguaje ubicuo** del dominio: esto significa nombrar las cosas en el código tal como se habla en el negocio. En el código ya se hace bastante bien (se usan nombres como *Appointment, Patient, Dentist/Doctor, etc.*). Mantener esa terminología consistente y evitar

términos genéricos (como usar “item” cuando realmente es “cita”) ayuda a que cualquier miembro del equipo (incluso no desarrollador) entienda qué hace cada parte.

En síntesis, aplicando DDD reorganizaremos el repositorio para **alinearlo con las funcionalidades del mundo real** (pacientes, doctores, agenda), agrupando código relacionado y aislando la lógica de cada dominio. Esto mejora la escalabilidad (añadir un nuevo dominio o modificar uno existente es más fácil), la **navegabilidad** del proyecto (encuentras todo sobre "citas" en un mismo lugar) y reduce el riesgo de mezclar conceptos de distintos dominios incorrectamente.

5. Aplicación del principio DRY (Don't Repeat Yourself)

El principio **DRY** indica “No te repitas”: busca eliminar la duplicación de código y de lógica. En el análisis del repositorio se identificaron varias áreas donde se repite código que podríamos refactorizar para cumplir DRY. La duplicación es uno de los problemas más comunes en aplicaciones React, y seguir el principio DRY resulta en un código más limpio y fácil de mantener [10] [11].

Algunas acciones concretas para aplicar DRY en este proyecto:

- **Unificar componentes duplicados:** Actualmente existen componentes duplicados con el mismo propósito, por ejemplo dos `Sidebar` y dos `Topbar` en distintas rutas (`src/components/Sidebar.jsx` y `src/components/layout/Sidebar.jsx`). Esto viola DRY, ya que cualquier cambio en la barra lateral habría que hacerlo en dos archivos. La mejora aquí es **quedarnos con una única implementación de Sidebar y Topbar** (preferiblemente la más completa, probablemente la ubicada en `components/layout/`). Esa versión única servirá de componente común de navegación en toda la app. Se debe eliminar la versión no utilizada para no repetir código. Igual con cualquier otro componente redundante.

- **Factorizar lógica común en funciones/hooks compartidos:** Por ejemplo, el código de las páginas de listado (pacientes, doctores) seguramente es muy parecido: mantienen un estado de lista, usan un efecto para fetch inicial, manejan loading/error, y tienen funciones para abrir modales de creación/edición. En lugar de repetir ese patrón en `PatientsPage.jsx` y `DoctorsPage.jsx`, podríamos extraer esa lógica común. Una opción es crear un **hook reutilizable** para listas con CRUD: por ejemplo `useListData(service)` que dado un servicio (pacientes o doctores) maneje internamente `items, loading, error, addItem, updateItem, deleteItem`. Cada página concreta llamaría a `useListData(patientService)` o `useListData(doctorService)` y obtendría todo ya resuelto. Así no repetimos la misma estructura de useEffect y handlers en ambas páginas. Incluso sin generalizar tanto, podríamos al menos tener un `usePatientsData` y `useDoctorsData` separados pero muy limpios, en lugar de incrustar la lógica en el componente.

- **Reutilizar componentes de presentación:** Si notas que estás creando múltiples componentes muy similares, es el momento de refactorizar. Un ejemplo: en la vista de calendario, probablemente hay un componente para cita diaria (`DailySchedule`) y otro para semanal (`WeeklySchedule`) con estructuras parecidas. En lugar de tener dos componentes duplicados manejando lógica similar, podríamos tener un solo componente de Calendario que cambie su presentación según el modo (día/semana) o componentes más pequeños reutilizables en ambas vistas (por ejemplo, ambos usan `CalendarHeader` para el encabezado, ambos usan `TimeColumn` para las horas). **No duplicar JSX o estilos**: mejor abstraer componentes

comunes. El código ya tiene un `CalendarHeader.jsx` y otros subcomponentes; asegurémonos de usarlos en ambas vistas para no repetir código de cabecera.

- **Lógica de cálculo centralizada:** Relativo al calendario, la lógica de asignar citas a múltiples *timeslots* no debe duplicarse en varias partes. Actualmente parece estar concentrada en `DentistColumn.jsx` (lo cual es bueno). Si en otro componente se necesitara calcular bloques horarios (por ejemplo, para un posible "vista semanal" multi-columna), en vez de reescribir ese cálculo, hay que reutilizarlo. Podríamos extraerlo a una función utilitaria `assignAppointmentsToSlots(appointments)` que devuelva la estructura de slots con citas posicionadas. Así, tanto la vista diaria como la semanal podrían usar la misma función, pasando la lista de citas relevante. **Cada trozo de lógica, idealmente, debe vivir en un solo lugar del código**. Esto garantiza que si hay que cambiar esa lógica, se cambia en un punto y automáticamente se refleja en todas las partes de la app que la usan.

- **Compartir estado en lugar de duplicarlo:** Si dos componentes necesitan acceder a la misma información, evitar que cada uno la obtenga por su cuenta (duplicación de llamadas) o la maneje separadamente (pudiendo causar inconsistencias). Por ejemplo, supongamos que la barra superior muestra el número de pacientes y la página de pacientes lista la tabla de pacientes. Sería tentador que cada uno llame a `getPatients`, duplicando la petición. Aplicando DRY, sería mejor tener un único origen de verdad: quizá un contexto global de pacientes o levantar el estado al ancestro común (si existiera). En este proyecto, podría no hacer falta un contexto global, pero es algo a tener en cuenta: **no duplicar fuentes de datos**. Un caso concreto: se ve en `SchedulePage` que hace `getAppointments()` y `getDoctors()`. La página de doctores también hace `getDoctors()`. Si en el futuro se navega del calendario a la lista de doctores, se estaría llamando dos veces lo mismo. Una mejora podría ser usar React Context para proveer la lista de doctores a toda la app una sola vez (siempre y cuando la lista no sea enorme). O al menos, almacenar en un estado global (por ejemplo con Zustand o Redux si aplica) la lista de doctores descargada para reusarla. Con esto evitamos llamadas redundantes y mantenemos una única copia de la información (DRY a nivel de datos).

- **Principio de una sola fuente de verdad:** Este es un corolario de DRY: cualquier dato o lógica debe existir en un solo lugar. Si ves el mismo literal o constante usados en varios sitios, define una constante compartida. Ej: si el valor "Estado: pendiente" aparece en varias vistas, mejor tener un enum o objeto central `STATUS = { PENDING: 'pendiente', ... }`. Así se evita que una corrección ortográfica o un cambio de texto haya que hacerlo en muchos archivos.

Como bien se resume: *"No duplicar código siempre que sea posible… Si estás creando múltiples componentes muy similares, es hora de extraer funcionalidad común en un componente compartido… siguiendo DRY tendrás aplicaciones React más limpias y fáciles de mantener"* [10] [11]. Implementar DRY en todo el repositorio eliminará redundancias, reduciendo el tamaño del código y evitando bugs (pues la lógica unificada en un solo lugar disminuye el riesgo de actualizaciones olvidadas en copias). La aplicación será más fácil de refactorizar y extender porque sabremos exactamente dónde modificar cada comportamiento.

6. Aplicación del patrón Modelo-Vista-Presentador (MVP)

El patrón **MVP (Model-View-Presenter)** es una variación de la clásica separación de capas (deriva de MVC) orientada a interfaces de usuario. Adopta una división triádica: - **Modelo (Model):** los datos o el dominio (en nuestro caso serían los datos de negocio: pacientes, doctores, citas, y las reglas asociadas – mucho de esto ya lo cubre DDD). - **Vista (View):** la interfaz de usuario, es decir, los componentes React

que renderizan la información y capturan eventos de usuario (clics, entradas de formulario). - **Presentador (Presenter):** la capa intermedia que actúa como **mediador** entre la Vista y el Modelo. El Presentador contiene la lógica de presentación: maneja eventos de la vista, solicita/actualiza datos del modelo (por medio de servicios) y decide qué le pasa a la vista (por ejemplo: "mostrar loading", "navegar a tal página", "enseñar mensaje de error").

En React, implementar MVP requiere mover la mayor parte de la lógica que hoy está en los componentes hacia **clases o módulos de Presentador**. Ahora mismo, muchos componentes hacen peticiones a la API, procesan datos y controlan estado (modales, etc.) además de renderizar JSX. Bajo MVP, **la vista debería ser (casi) tonta**: *"en general, nuestras vistas no deben manejar lógica de negocio; solo pueden manejar lógica puramente de UI (presentación)"* [12] . El Presentador, en cambio, sí contiene lógica de negocio de la capa de aplicación: sabe qué hacer cuando el usuario interactúa.

¿Cómo aplicar MVP en este proyecto?

- **Crear presentadores para cada vista importante:** Por ejemplo, podríamos tener un `PatientsPresenter`, `DoctorsPresenter`, `SchedulePresenter`, etc., cada uno encargado de la lógica de esa pantalla. Estos presentadores podrían ser **clases** tradicionales con métodos, o de forma más idiomática en React, **hooks personalizados** que encapsulen la lógica (estado + funciones) de la vista. Ambos enfoques son válidos:
- *Enfoque con clases:* Crear una clase JS que mantenga el estado necesario (podría internamente usar observables o simplemente exponer métodos que devuelven datos). La vista React crearía una instancia del presentador y la usaría para invocar acciones o leer datos. Este enfoque requiere más infraestructura para ligar la clase con la reactividad de React, pero es parecido a como se hace MVP en otras plataformas (ej: Android).

- *Enfoque con hooks (funcional):* Es práctico en React aprovechar hooks para implementar el presentador. Un **hook presentador** puede usar `useState`, `useEffect` y otras APIs React internamente para manejar estado y efectos, y exponer métodos para acciones. La vista simplemente llama al hook y obtiene todo lo necesario. Este método encaja bien con la naturaleza funcional de React.

- **Mover la lógica de las páginas a los presentadores:** Tomemos por ejemplo `SchedulePage.jsx`. Hoy este componente realiza:

- Petición inicial de datos de citas y doctores (`useEffect` con `getAppointments()` y `getDoctors()`).
- Manejo de estados locales: lista de citas, lista de doctores, doctor seleccionado, fecha actual, tipo de vista (día/semana), estados booleanos para mostrar/ocultar modales, etc.
- Definición de handlers para acciones: abrir modal nueva cita, abrir detalle, confirmar borrado, etc.
- Renderizado de la estructura de calendario (incluyendo el `CalendarView` con sus subcomponentes).

En MVP, separaríamos estas responsabilidades. Crearíamos, por ejemplo, `useSchedulePresenter` que internamente haga el fetch de datos, guarde los estados y provea funciones para abrir/cerrar modales y demás lógica. Así el componente de vista `SchedulePage` quedaría mucho más limpio, solo ocupándose de la disposición de los componentes. **Ejemplo de implementación con hook presentador:**

```
// Presentador (hook) para la agenda
function useSchedulePresenter() {
  const [date, setDate] = useState(new Date());
  const [viewType, setViewType] = useState('day');
  const [doctors, setDoctors] = useState([]);
  const [appointments, setAppointments] = useState([]);
  const [selectedDoctor, setSelectedDoctor] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  // estados para modales
  const [showNew, setShowNew] = useState(false);
  const [showDetail, setShowDetail] = useState(false);
  // ... otros estados necesarios

  useEffect(() => {
    // Cargar doctores y citas iniciales
    const loadData = async () => {
      setLoading(true);
      try {
        const [apps, docs] = await Promise.all([
          appointmentService.getAll(),
          doctorService.getAll()
        ]);
        setAppointments(apps);
        setDoctors(docs);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false);
      }
    };
    loadData();
  }, []);

  // Actions / handlers:
  const createAppointment = async (data) => {
    try {
      const newApp = await appointmentService.create(data);
      setAppointments(prev => [...prev, newApp]);
      setShowNew(false);
    } catch (err) {
      setError(err);
    }
  };
  const selectAppointment = (app) => {
    // e.g., open detail modal for app
    setSelectedDoctor(app.doctor);
    setShowDetail(true);
  };
  // ... más métodos como updateAppointment, deleteAppointment, etc.
```

```
  return {
    date, setDate,
    viewType, setViewType,
    doctors, appointments,
    loading, error,
    showNew, setShowNew,
    showDetail, setShowDetail,
    createAppointment, selectAppointment,
    // ... etc.
  };
}
```

```
// Vista (SchedulePage.jsx) usando el presentador
function SchedulePage() {
  const navigate = useNavigate();
  const {
    date, setDate,
    viewType, setViewType,
    doctors, appointments,
    loading, error,
    showNew, setShowNew,
    showDetail, setShowDetail,
    createAppointment, selectAppointment
  } = useSchedulePresenter();

  if (loading) return <Spinner/>;         // Ejemplo: mostrar spinner
mientras carga
  if (error) return <Error message={error.message}/>;

  return (
    <>
      <CalendarHeader
        date={date}
        viewType={viewType}
        onChangeDate={setDate}
        onChangeView={setViewType}
      />
      <CalendarView
        viewType={viewType}
        date={date}
        doctors={doctors}
        appointments={appointments}
        onSelectAppointment={selectAppointment}
      />
      {showNew && (
        <Modal onClose={() => setShowNew(false)}>
          <AppointmentForm onSubmit={createAppointment} onCancel={()=>
setShowNew(false)}/>
        </Modal>
```

```
      )}
      {showDetail && (
        <Modal onClose={() => setShowDetail(false)}>
          <AppointmentDetail appointment={/* detalle de cita actual */} />
        </Modal>
      )}
      <Button onClick={()=> setShowNew(true)}>Nueva cita</Button>
    </>
  );
}
```

En el ejemplo anterior, `useSchedulePresenter` maneja toda la lógica de datos y eventos. La vista `SchedulePage` solo se suscribe a los datos (mediante las variables devueltas) y define cómo mostrarlos (por ejemplo, qué componentes usar para calendario, cómo estructurar los modales). Nótese que la vista no sabe nada de cómo se obtienen los datos o qué pasa al crear una cita; solo llama `createAppointment` del presentador. De esta forma, **el Presentador controla qué ocurre en la vista, pero no el cómo**: por ejemplo, el presentador puede indicar "hay un error, debe mostrarse", y la vista decide *cómo* mostrar ese error (un componente <Error>, un alert, etc.) – cumpliendo la separación de MVP.

- **Comunicación Vista-Presentador:** En MVP clásico, la vista y el presentador se comunican a través de interfaces (la vista expone una interfaz que el presentador invoca para actualizar UI, y el presentador una que la vista usa para notificar eventos). En React podemos simplificar esto: la *vista* reacciona a cambios de estado (por ej., `error` no es null entonces renderiza mensaje) en lugar de tener métodos explícitos como `showError()`. El presentador, al actualizar el estado que comparte con la vista (por ejemplo `setError`), ya provoca ese cambio. Para eventos de usuario, la vista simplemente llama funciones del presentador (como `createAppointment`). Esta comunicación es suficiente y sigue el espíritu MVP: la vista no está tomando decisiones de negocio, solo pasa el evento al presentador.

- **No todas las vistas necesitan presentador:** Si una vista (página o componente) es muy simple y no contiene lógica de negocio, no hace falta crear un presentador "de relleno". Por ejemplo, supongamos que `HomePage` solo muestra un dashboard estático con botones de navegación; no tendría lógica más allá de JSX, entonces no necesita un presentador (sería un caso de *Passive View* trivial). En el proyecto, el layout general (Sidebar + Topbar + Outlet de contenido) tampoco requiere presentador propio ya que solo organiza la interfaz. Debemos aplicar MVP donde aporte valor: en aquellas vistas con interacciones complejas o manejo de datos significativo.

- **Facilitar pruebas unitarias:** Un gran beneficio de MVP es que los presentadores (si los diseñamos sin dependencias de la vista o de detalles de React) pueden probarse en aislamiento. Podemos instanciar un presentador, inyectarle servicios simulados (siguiendo DIP) y verificar que su lógica funciona (por ejemplo, que al llamar a `createAppointment` actualiza la lista de citas interna correctamente, maneja errores, etc.). La vista se vuelve tan simple que podríamos probarla con pruebas de snapshot o de integración sabiendo que la lógica ya está testeada en el presentador. Esto aumenta la confiabilidad del sistema.

En resumen, implementar MVP en todo el repositorio implicará **separar la lógica de cada página en un módulo/hook de Presentador** y dejar que los componentes de Vista sean declarativos. Esto resulta en componentes React más simples, más cercanos a "plantillas" o contenedores de UI, y una capa de presentadores donde concentramos la lógica (fácil de mantener y cambiar). La consecuencia positiva es

menos duplicación de lógica entre componentes (porque la centralizamos en el presentador) y un cumplimiento más estricto de SRP en los componentes. Esta refactorización sienta las bases para una arquitectura limpia: la **Presentación** separada del **Dominio/Aplicación**, tal como recomienda DDD. De hecho, muchos principios SOLID quedan naturalmente satisfechos: por ejemplo, SRP (vista y presentador tienen responsabilidades distintas y únicas) y DIP (podemos inyectar servicios al presentador fácilmente para testear, etc.), como vimos antes.

7. Estructura de carpetas clara y escalable

Con las mejoras de arquitectura propuestas (DDD + MVP + componentes reutilizables), es importante reorganizar el proyecto con una **estructura de carpetas** que refleje esa separación. A continuación se sugiere una estructura escalable, separando presentadores, vistas, componentes reutilizables y componentes de estilo base, manteniendo también la agrupación por dominios:

```
src/
├── domains/  (Módulos por contexto de negocio)
│   ├── appointments/        (Dominio de Citas/Calendario)
│   │   ├── model/           (Modelos o entidades de negocio, p.ej.
Appointment.js, tal vez Value Objects)
│   │   ├── services/        (Servicios de infraestructura, p.ej.
appointmentService.js para llamadas API)
│   │   ├── presenter/       (Lógica de presentación, p.ej.
useSchedulePresenter.js)
│   │   ├── views/           (Componentes de vista/page, p.ej.
SchedulePage.jsx)
│   │   └── components/      (Componentes específicos de este dominio,
reutilizables dentro de él)
│   │       ├── CalendarView.jsx
│   │       ├── CalendarHeader.jsx
│   │       ├── DoctorColumn.jsx
│   │       ├── AppointmentCard.jsx
│   │       ├── TimeSlot.jsx
│   │       └── ... (otros, e.g. NotAvailableArea.jsx, BreakTime.jsx)
│   ├── patients/            (Dominio de Pacientes)
│   │   ├── model/           (p.ej. Patient.js)
│   │   ├── services/        (p.ej. patientService.js)
│   │   ├── presenter/       (p.ej. usePatientsPresenter.js)
│   │   ├── views/           (PatientsPage.jsx, PatientDetailPage.jsx,
PatientNewPage.jsx)
│   │   └── components/      (Componentes UI específicos de pacientes)
│   │       ├── PatientCard.jsx
│   │       ├── PatientForm.jsx
│   │       └── Odontogram.jsx  (si es específico de paciente)
│   ├── doctors/             (Dominio de Doctores)
│   │   ├── model/           (p.ej. Doctor.js)
│   │   ├── services/        (doctorService.js)
│   │   ├── presenter/       (useDoctorsPresenter.js)
│   │   ├── views/           (DoctorsPage.jsx)
│   │   └── components/      (UI de doctores)
```

```
│   │       ├── DoctorForm.jsx
│   │       └── DoctorList.jsx (si hubiese lista separada de page)
│   └── ... (otros dominios futuros)
│
├── components/  (Componentes compartidos a nivel global)
│   ├── layout/        (Componentes de diseño presentes en todas las vistas)
│   │   ├── Sidebar.jsx
│   │   └── Topbar.jsx
│   ├── ui/            (Componentes de estilo base reutilizables en toda la
app)
│   │   ├── Button.jsx
│   │   ├── Card.jsx
│   │   ├── Modal.jsx
│   │   └── ... (otros átomos/moléculas: Input.jsx, Avatar.jsx, etc.)
│   └── common/        (Otros componentes compartidos que no sean puro
estilo,
│                       p.ej. quizás un componente Avatar genérico, o
componentes multi-dominio)
│
├── hooks/            (Hooks reutilizables generales, si los hay)
│   └── useApi.js     (por ejemplo un hook genérico para llamadas fetch con
cancelación, etc.)
│
├── utils/            (Funciones utilitarias puras)
│   └── dateUtils.js, classNames.js, etc.
│
├── App.jsx           (Configuración de rutas y layout general)
└── index.jsx         (punto de entrada de ReactDOM)
```

Explicación de la estructura propuesta:

- **domains/**: Agrupa el código por **contexto de dominio**. Cada subcarpeta (appointments, patients, doctors, ...) contiene todo lo necesario para esa parte de la aplicación. Esto sigue DDD. Dentro de cada dominio:
- **model/**: define modelos o tipos del dominio. Si usáramos TypeScript, aquí estarían las interfaces o tipos para las entidades, o clases de dominio. En JS plain, pueden ser archivos con definiciones JSDoc o simplemente conceptual.
- **services/**: contiene la comunicación con APIs u otras infraestructuras para ese dominio. En el código actual, los archivos `patientService.js`, `doctorService.js`, etc., entrarían aquí. Así mantenemos separados los detalles de fetching de datos.
- **presenter/**: aquí pondríamos los presentadores o lógica de aplicación para las vistas de ese dominio. Pueden ser archivos de hooks (p.ej. `useSchedulePresenter.js`) o clases. La idea es que cada página principal tenga su presentador. También podríamos tener presentadores por sub-caso de uso si fuese complejo (por ejemplo, un `PatientDetailPresenter` distinto de `PatientsListPresenter` si se prefiere separar).
- **views/**: componentes React que representan páginas o secciones de interfaz del dominio. Por ejemplo `SchedulePage.jsx`, `PatientDetailPage.jsx`, etc. Estos componentes usarán los presentadores correspondientes. Podemos mapear estas vistas a rutas en `App.jsx`.

- **components/**: componentes *visuales* específicos del dominio, que pueden ser usados por las vistas de ese dominio. Aquí irían aquellos que no son globales porque solo tienen sentido en ese contexto. Por ejemplo, en *appointments/components* incluimos los componentes del calendario (CalendarView, AppointmentCard, etc.) ya que solo se usan en la funcionalidad de agenda. Similarmente, en *patients/components* podría estar `PatientForm` o la representación del odontograma si solo se usa en la vista de pacientes. Esta separación permite que los **componentes reutilizables dentro de un dominio** estén juntos (aplicando DRY dentro del dominio) sin ensuciar el espacio global de componentes.

- **components/layout/**: aquí ubicamos componentes comunes de **layout** que aparecen en (casi) todas las páginas, en este caso la barra lateral de navegación (Sidebar) y la barra superior (Topbar). Estos actúan como contenedores de la estructura general de la app. Podemos implementar un componente `<MainLayout>` que renderice `<Sidebar/>`, `<Topbar/>` y un `<Outlet>` para el contenido de cada página. React Router nos permite definir un layout por defecto para rutas, así que podríamos mover la lógica de App.jsx allí (ver apartado de rutas abajo). Tener `layout/Sidebar.jsx` y `layout/Topbar.jsx` en este directorio clarifica que son parte del esqueleto de la aplicación. Al ser comunes, deben ser altamente reutilizables y **única fuente** (no duplicados).

- **components/ui/**: este directorio contendrá los **componentes de estilo base** o de interfaz genéricos (a veces llamados *átomos* y *moléculas* en diseño atómico). Aquí ponemos cosas como:

  - Botones genéricos (`Button.jsx`): por ejemplo un botón estilizado según el tema de la app, que acepte props para variante (primario, secundario) y tamaño.
  - Tarjetas o paneles (`Card.jsx`): un contenedor con cierto estilo (sombra, padding) usado para agrupar contenido en distintas vistas.
  - Modal genérico (`Modal.jsx`): ya existe uno, para cuadros de diálogo reutilizables.
  - Inputs básicos, selectores, etc., si fuera necesario crear propios, también aquí.

Estos componentes **no están ligados a ningún dominio**; son piezas de UI reutilizables en cualquier parte de la aplicación. Constituyen una especie de "mini librería de diseño" interna. Más adelante, al hablar de la librería de estilos, reforzaremos su importancia. Al centralizar estos componentes base aquí, se asegura consistencia visual (todas las partes de la app usan el mismo botón, las mismas tarjetas, etc.).

- **components/common/**: este es opcional, pero puede servir para componentes compartidos que no son exactamente "átomos" UI ni pertenecen a un solo dominio. Por ejemplo, podría existir un componente `Avatar.jsx` genérico que muestre las iniciales de una persona con un fondo de color (como el DoctorAvatar actual, pero generalizado). Podríamos ubicarlo en `ui/` si es parte de estilo base, o en `common/` si es más complejo. Otra posibilidad: un componente `ConfirmDialog` genérico para confirmaciones de eliminación (si lo implementamos, usado tanto en pacientes, doctores, etc.), iría en common. En resumen, *common* agrupa componentes reutilizables de nivel medio (ni tan básicos como un botón, ni tan específicos de un dominio).

- **hooks/**: aquí se pueden poner hooks reutilizables que no pertenezcan a un dominio específico. Por ejemplo, `useApi.js` actual (que configura axios) podría moverse aquí. O un hook `useWindowDimensions` si hubiera, etc. Si no hay hooks globales, este directorio no es obligatorio.

- **utils/**: utilidades puras sin React, p.ej. funciones de formateo de fecha (dateUtils), función `classNames` para concatenar clases, etc., como ya existen. Mantenerlas en utils es correcto.

- **App.jsx y rutas:** Con esta estructura, `App.jsx` puede simplificarse. Se puede configurar React Router para usar un layout por defecto. Por ejemplo:

```
<Routes>
  <Route element={<MainLayout/>}>  {/* MainLayout contiene Sidebar + Topbar + Outlet */}
    <Route path="/" element={<HomePage/>}/>
    <Route path="/patients" element={<PatientsPage/>}/>
    ... etc
  </Route>
</Routes>
```

  De ese modo, `MainLayout.jsx` (que vive en components/layout/) tendría el `<Sidebar/>` y `<Topbar/>`. Esto limpia App.jsx. Es una buena práctica separar la definición de rutas de la definición del layout visual. El archivo App.jsx quedaría principalmente como orquestador de rutas.

La estructura propuesta asegura **separación de responsabilidades por carpeta**: los presentadores están separados de las vistas, los componentes de UI genéricos separados de los específicos, y todo organizado por dominio para escalabilidad. Si mañana se agrega una nueva sección (por ejemplo, *Reportes*), simplemente se crea `src/domains/reports/` con sus subcarpetas y se añade la ruta correspondiente, sin afectar el resto del proyecto.

Veamos específicamente los casos que el usuario preguntó sobre organización:

- **Componentes comunes (barra lateral, barra superior):** Como indicado, ubicarlos en `components/layout/`. Ambas barras se usan en todas las vistas, por lo que deben ser únicas e importadas por el layout principal. Internamente, pueden estar diseñadas con componentes de la librería de estilos (por ejemplo, la barra superior podría usar componentes `Button` de ui/ para sus botones, etc.). Es útil que `Sidebar` genere su menú a partir de una lista de rutas central (así, agregar una nueva sección es fácil – OCP aplicado). Podríamos tener un archivo `routesConfig.js` donde definimos las secciones y sus íconos, y `Sidebar` mapea sobre eso.

- **Componentes dentro de cada vista (caso pestaña de calendario):** En el calendario (agenda de citas) tenemos varias piezas: tarjetas de cita, columna de doctor, columna de horas, áreas de no disponible, encabezado de calendario, etc. La sugerencia es que *cada una de estas piezas sea un componente reutilizable dentro del dominio de citas*. En la estructura propuesta, esos componentes están en `domains/appointments/components/`. Por ejemplo:

- `CalendarHeader.jsx`: componente que renderiza la cabecera del calendario (contiene quizás los días de la semana en vista semanal, o la fecha en vista diaria, y controles para navegar entre fechas). Este componente se usa dentro de `SchedulePage` o `CalendarView`, y es reutilizable si hay varias vistas de calendario (día vs semana).
- `TimeColumn.jsx`: ya existe, representa la columna de horarios. Es un componente puramente presentacional (lista de horas). Debe permanecer simple y reutilizable.

- `DoctorColumn.jsx`: representa la columna de un doctor con sus citas en ciertas horas. Este componente actualmente maneja la distribución de citas en slots y renderiza subcomponentes `Appointment` y quizás `BreakTime` y `NotAvailableArea`. Podría mantenerse así, aunque es bastante lógico. Alternativamente, podríamos dividirlo: un componente `AppointmentCard` separado (para mostrar la cita con un determinado estilo), un componente `BreakTime` (ya existe) para la franja de descanso, y uno para "no disponible". El `DoctorColumn` entonces solo itera sobre sus slots y decide qué componente insertar en cada uno (cita, break, etc.). Esto ya sucede en el código actual en alguna medida. Lo importante es que toda esta familia de componentes quede agrupada y separada de otros dominios. Ningún componente de Pacientes o Doctores debería preocuparse de cómo se renderiza una cita; solo el dominio de Appointments lo sabe.

- **Lógica de múltiples timeslots:** Respecto a cómo hacer que una cita ocupe múltiples slots, hay dos enfoques: uno, como el actual, dividir la cita en segmentos (un segmento inicial y varios continuos) en el array de slots; otro enfoque es usar CSS para que un componente `<AppointmentCard duration={n}>` tenga un estilo `height: n * slotHeight`. El primero ya está implementado con isStart/!isStart. Si mantenemos esa técnica, encapsulemos esa lógica en una función o dentro de `DoctorColumn`, de modo que si se reutiliza `DoctorColumn` en otro contexto (por ejemplo, en vista semanal con múltiples doctores columnas), podamos pasarle las citas y que funcione igual. **Idealmente, la lógica de ocupar múltiples slots estaría en el presentador de citas**: es decir, al preparar la data para la vista, calcular `startSlot` / `endSlot` de cada cita según la duración. De hecho, en el typedef de `AppointmentData` ya figuran `startSlot` y `endSlot`, lo cual sugiere que el cálculo podría venir del backend o lo calculó antes. Si no, podemos calcularlo en el presentador una vez, y pasarle a la vista citas ya con esas propiedades. Así la vista no tiene que deducir duración en minutos, simplemente usa los slots pre-calculados. En cualquier caso, ese cálculo no debe duplicarse: debe haber un único lugar (un helper o presentador) que establezca que, por ejemplo, una cita de 1 hora ocupa 2 slots de 30min. Esto cumple DRY y SRP.

- **Encabezado del horario como componente:** Sí, como se indicó, `CalendarHeader` es un componente propio. En la vista semanal, `CalendarHeader` podría mostrar los días de la semana; en vista diaria, quizás solo muestra el día actual con flechas. Podemos hacer que `CalendarHeader` sea parametrizable (por ejemplo, prop `viewType`). En el código actual hay `calendar-header.jsx` que seguramente cumple esta función. Lo importante es que esté desacoplado de la página: la página le pasa la fecha actual y qué hacer cuando se cambia, y el header encapsula la presentación de esos controles.

Resumiendo, la estructura de carpetas propuesta separa claramente: - **Presentadores** (en carpeta presenter de cada dominio) – lógica de cada caso de uso. - **Vistas** (en views de cada dominio) – componentes de página que usan presentadores y componentes de UI. - **Componentes reutilizables de dominio** (en components de cada dominio) – trozos de UI específicos pero usados en varias partes dentro de ese dominio. - **Componentes comunes (cross-domain):** layout (para estructura general) y ui (base visual). - **Estilos base:** centralizados en componentes ui/ y eventualmente archivos de estilo global (como tailwind.css si aplica).

Esta arquitectura de carpetas es **escalable**: nuevos dominios se añaden fácilmente; nuevos componentes UI base se añaden sin interferir con lógica de negocio; presentadores y vistas crecen de la mano dentro de su dominio. También mejora la **navegación del desarrollador** por el proyecto: si estoy trabajando en funcionalidad de calendario, miro todo dentro de `domains/appointments`; si modifico el diseño general, voy a `components/layout` o `components/ui`. Cada cosa en su lugar.

8. Librería de estilos y componentes visuales reutilizables

Para recrear el diseño visual profesional y moderno como el mostrado en **Zendenta** (referencia: fikristudio.com), es recomendable apoyarse en una **librería de estilos** consistente. Dos buenas opciones mencionadas son **Tailwind CSS** y **Chakra UI**:

- **Tailwind CSS:** Es una librería de utilidades CSS que permite aplicar estilos mediante clases predefinidas (p. ej., `bg-gray-50`, `text-xl`, `p-4`, etc.). Ventajas:
- Permite implementar fielmente un diseño personalizado como Zendenta, ya que brinda mucho control granular. Podemos definir en la configuración de Tailwind el palette de colores de la clínica, fuentes, tamaños de componente, etc.
- Ya está configurado en el proyecto (existe `tailwind.config.js` y se usan clases como `bg-blue-100 text-blue-700` en el código). Podemos **extender la configuración** para incluir los colores corporativos de Zendenta (por ejemplo, si usan un verde menta y azul, añadirlos como `primary` y `secondary`).
- Con Tailwind, crearíamos una base de estilos consistente usando clases utilitarias, y podríamos componer componentes reutilizables fácilmente. Por ejemplo, definir que todos los cards tengan `rounded-xl shadow-md bg-white` mediante un componente `<Card>` que aplique esas clases. Luego, usar `<Card>` en todas las vistas garantiza el mismo estilo.

- Tailwind también facilita el diseño **responsive** con sus prefijos (`sm:`, `md:`), importante si la app debe verse bien en diferentes tamaños de pantalla.

- **Chakra UI:** Es una librería de **componentes React pre-construidos** con un sistema de diseño configurable. Sus puntos fuertes:

- Ofrece componentes listos para usar como botones, inputs, modales, tooltips, barras laterales, etc., con un estilo base atractivo y accesibilidad por defecto.
- Es altamente *themeable*: se puede configurar un tema para que los componentes de Chakra adopten colores y fuentes deseados. Por ejemplo, se puede ajustar el tema para que el color `teal` de Chakra sea el verde/azul de la identidad Zendenta, y luego usar componentes `<Button colorScheme="teal">` etc. que automáticamente se verán consistentes.
- Chakra aplica muchos principios de diseño consistentes, lo que podría acelerar la implementación de una interfaz coherente sin escribir CSS manualmente.
- Además, Chakra utiliza un sistema de props de estilo (similar a Tailwind en flexibilidad, pero a través de props en vez de clases) y provee utilidades responsivas, manejo de grids, etc.

Dado que el proyecto ya usa Tailwind, una estrategia viable es **continuar con Tailwind** para fine-tuning de estilos, complementándolo quizás con alguna librería de componentes "headless" o minimalista si se desea (por ejemplo, usar **Headless UI** de Tailwind Labs para modales, menús accesibles, etc., o **Radix UI**). Sin embargo, incorporar Chakra UI podría implicar un cambio mayor, ya que habría que sustituir algunos componentes existentes por los de Chakra (por ejemplo, reemplazar nuestro Modal y Button con los de Chakra). No es imposible y Chakra se integra bien con Vite/React, pero debe evaluarse el esfuerzo.

Independientemente de la elección, necesitamos crear una **base de componentes visuales reutilizables** que garanticen un diseño uniforme en todas las secciones. Ya identificamos en `components/ui/` los candidatos: - Botones, tarjetas, modales, inputs, etc. - Podríamos añadir: **Barra de navegación lateral** estilizada acorde al diseño (si no usamos Chakra's Drawer, la implementamos con Tailwind), - **Tablas o listas**: por ejemplo, la lista de pacientes podría beneficiarse de un componente de tabla estandarizado (con cierto estilo de cabecera, filas zebra, etc.). - **Componente Avatar**: muy útil

para mostrar iniciales de usuarios (doctores, pacientes). Podemos generalizar lo que hace `DoctorAvatar` para que reciba un nombre y a partir de la primera letra decida un color de fondo consistente (como ya hace). Este Avatar se usaría en la lista de pacientes, lista de doctores, etc., proporcionando identidad visual (por ejemplo, un círculo con iniciales y colores pastel como en Zendenta). - **Sistema de íconos:** Zendenta seguramente usa íconos para menú, estados, etc. Podríamos integrar una librería de íconos (Heroicons combina bien con Tailwind, Chakra también tiene su set de iconos). Asegurarse de usar los mismos íconos en toda la app (evitar iconos dispares) es parte de la consistencia de estilo.

Con Tailwind, se pueden definir componentes base usando clases aplicadas a elementos: Por ejemplo:

```
// Button.jsx (con Tailwind classes)
export default function Button({ variant = "primary", ...props }) {
  const base = "px-4 py-2 font-semibold rounded flex items-center justify-
center";
  const variants = {
    primary: base + " bg-blue-600 text-white hover:bg-blue-700",
    secondary: base + " bg-gray-100 text-gray-800 hover:bg-gray-200",
    danger: base + " bg-red-600 text-white hover:bg-red-700",
  };
  return <button className={variants[variant]} {...props} />;
}
```

Con este patrón, cada botón de la app usa este componente en vez de escribir clases cada vez, logrando DRY en estilos. Similarmente, un `Card.jsx` podría aplicar clases de borde, sombra y padding.

Si optásemos por Chakra, podríamos definir un `ChakraProvider` con un tema personalizado. Chakra ya tiene componentes <Button>, <Card> (como Box con sombra), etc. Reemplazaríamos gradualmente nuestros `<button className="...">` por `<Button>` de Chakra con el colorScheme definido. Chakra también tiene componentes de layout como `<Grid>`, `<Stack>`, que pueden simplificar la escritura de HTML/CSS. Dado que el diseño Zendenta es tipo dashboard administrativo (según la descripción, seguramente con sidebar, headers, tarjetas de métricas, calendario interactivo), Chakra cubre esos patrones (tiene componentes de calendario? No directamente, pero se puede componer con DatePickers externos).

En cuanto a **emular el diseño de Zendenta**: - Analizar su paleta de colores: probablemente blancos, grises claros, un color primario suave (quizá verde azulado) para resaltar, y acentos para estados (por ejemplo, en nuestro código se ven status 'pink', 'green', 'blue', 'yellow' para las citas). Podemos armonizar esos colores con la paleta global para que todos los componentes (botones, badges de estado, etc.) sigan un mismo estilo. - Tipografías: asegurar usar una fuente agradable y consistente (Tailwind por defecto viene con una, Chakra con otra, podemos ajustarlo). - Bordes y sombras: el diseño moderno suele usar bordes redondeados y sombras sutiles en tarjetas. Definir esas características en los componentes base (por ejemplo, nuestra `Card` siempre tendrá `rounded-lg` y `shadow-sm`). - Espacios y layouts: Tailwind nos permite establecer un sistema de espaciado (p. ej., usar múltiplos de 4px) que aplicaremos en padding/margin para uniformidad. Chakra ya tiene spacing scale predefinida (4, 8, 12px etc.). - Estado oscuro/claro: Zendenta probablemente solo tiene tema claro, pero si se quisiera, tanto Tailwind como Chakra permiten implementar un dark mode fácilmente (Chakra con su color mode, Tailwind con clases dark:).

La idea es crear una suerte de **Design System interno**. Esto lo logramos definiendo esos componentes en `components/ui/` y quizás un documento de referencia de estilos (podríamos incluir en la documentación del proyecto una guía de uso de componentes comunes, para que todos los desarrolladores usen los mismos en vez de reinventar estilos en cada vista).

Conclusión de estilos: Si el equipo está cómodo con **TailwindCSS**, continuar con él y potenciarlo es buena opción para reflejar píxel a píxel el diseño deseado. Si prefieren velocidad en montar interfaz y buenas prácticas accesibles out-of-the-box, **Chakra UI** brindaría componentes listos (aunque ajustar detalles a un diseño específico a veces requiere sobreescribir algunos estilos de Chakra). Incluso es posible combinarlos: usar Tailwind para utilidades rápidas y Chakra para componentes complejos, pero normalmente se elige uno principal para no mezclar demasiado. En cualquier caso, establecer una **biblioteca de componentes visuales reutilizables** garantizará que todas las pestañas de la aplicación tengan una apariencia unificada y de calidad profesional.

9. Guía de pasos para refactorizar el proyecto (MVP + DDD + DRY)

Finalmente, proponemos una **hoja de ruta** paso a paso para refactorizar todo el proyecto siguiendo la arquitectura sugerida. Estos pasos ayudarán a realizar la transformación de manera controlada y verificable:

1. **Preparación y análisis inicial:**
2. *Auditar el código existente* identificando duplicaciones y dependencias. Listar qué componentes/archivos están duplicados (como Sidebar, Topbar) y qué partes del código concentran demasiada lógica (por ejemplo, páginas como SchedulePage, DoctorsPage).

3. *Configurar entorno de refactorización:* Asegurarse de tener tests (si existen) o preparar un plan de pruebas manual para las funcionalidades principales antes de empezar, de modo que después de cada cambio puedas verificar que todo sigue funcionando. También es útil instalar Prettier/eslint en el editor para ir corrigiendo formato sobre la marcha.

4. **Reestructurar las carpetas por dominios:**
   Crea la nueva estructura de directorios dentro de `src/` tal como se propuso (domains, etc.). Mueve los archivos existentes a su nueva ubicación:

5. Por ejemplo, crea `src/domains/appointments/{views,components,presenter,services}` y mueve `SchedulePage.jsx` a `views/`, mueve los componentes de calendario (`calendar-header.jsx`, `calendar-view.jsx`, etc.) a `components/`, mueve `appointmentService.js` a `services/`, etc.
6. Haz lo mismo con pacientes y doctores: `PatientsPage.jsx` y otros van a `domains/patients/views`, sus componentes (PatientForm, PatientList, PatientCard) van a `domains/patients/components`, el servicio a `services/`.
7. Actualiza los imports en cada archivo movido. Por ejemplo, si `PatientsPage.jsx` importaba algo desde `"../components/common/Button"`, ahora la ruta puede ser `"../../../components/ui/Button"` según la nueva ubicación. Utiliza buscar/reemplazar global cuidadosamente para ajustar rutas de import.

8. Verifica después de mover cada conjunto que la app sigue compilando y corriendo (aunque todavía no hay cambios de lógica, solo movidos). Realiza commits intermedios en el control de versiones para aislar la reorganización de archivos (esto facilita revertir si algo rompe inesperadamente).

9. **Unificar componentes comunes (layout):**

10. Decide qué versión de Sidebar y Topbar conservar. Seguramente la de `components/layout` es más completa (tiene lógica de resaltar activo, etc.). Elimina la duplicada en `components/Sidebar.jsx` y corrige cualquier importación que apuntara a la antigua (ajústalas a la nueva ubicación si es necesario). Ahora hay solo una Sidebar y Topbar.
11. Crea un componente `MainLayout.jsx` en `components/layout/` que renderice `<Sidebar/>` y `<Topbar/>` con la estructura adecuada (por ejemplo, con contenedores flex). Usa un `<Outlet/>` de React Router para representar las vistas internas. Alternativamente, puedes seguir como estaba en App, pero envolver en un componente tiene ventajas de claridad.
12. Actualiza `App.jsx` para usar este layout. Por ejemplo:

```
<Routes>
  <Route element={<MainLayout/>}>
    <Route path="/" element={<HomePage/>}/>
    <Route path="/patients" element={<PatientsPage/>}/>
    ... etc.
  </Route>
</Routes>
```

13. Comprueba en el navegador que la navegación sigue funcionando y que la barra lateral y superior aparecen correctamente en todas las páginas.

14. **Implementar Presentadores (lógica de negocio fuera de las vistas):**
Por cada página principal, crea su presentador:

15. Para páginas de lista (Pacientes, Doctores): puede ser un hook como `usePatientsPresenter` y `useDoctorsPresenter` que maneje la carga inicial de datos (useEffect llamando al servicio), el estado `loading/error`, y funciones `createX, updateX, deleteX` manejando llamadas al servicio y actualizando estado. Internamente, aplicará la lógica específica (por ejemplo, quizás ordenar la lista, etc.).
16. Para la página de Agenda (SchedulePage): implementa `useSchedulePresenter` como en el ejemplo de la sección MVP. Mueve al presentador toda la lógica de estado y efectos: fetching de citas/doctores, estado de fecha actual, tipo de vista (día/semana), manejo de modales.
17. Asegúrate de inyectar en los presentadores los servicios necesarios. Puedes importarlos directamente (acoplamiento directo) o pasarlos como parámetros al hook (por ejemplo `usePatientsPresenter(patientService)`), lo cual te permite inyectar mocks en tests. Según la complejidad, decide un enfoque.
18. Cada presentador (hook) debe **devolver los datos y funciones necesarios** para que la vista funcione, y esconder los detalles internos. Por ejemplo, `usePatientsPresenter` podría devolver `{ patients, loading, error, openNewModal, selectedPatient, selectPatient, savePatient, removePatient }`. La vista entonces usará estos valores.

19. Testea cada presentador de forma aislada (si es hook, quizás mediante componentes de ejemplo o simplemente mirando en React DevTools el estado) antes de conectarlo. También contempla edge cases: ¿qué pasa si la API falla? El presentador debe asignar `error` y la vista luego mostrar algo.

20. **Refactorizar las Vistas para usar los Presentadores:**
    Ahora modifica las páginas (views) para que utilicen los presentadores en lugar de su lógica interna:

21. En `PatientsPage.jsx`, en lugar de tener sus propios `useState` y `useEffect` llamando a API, sustituye por algo como:

    ```
    const { patients, loading, error, openNewModal, ... } =
    usePatientsPresenter();
    ```

    y adapta el JSX: por ejemplo, donde antes había `patients.map(...)` sigue igual, pero ya proviene del hook. Donde había un botón "Agregar Paciente" con onClick que seteaba un estado local `setShowModal(true)`, ahora debe llamar `openNewModal()` del presentador (que probablemente setea un estado que controla mostrar modal).
22. En `SchedulePage.jsx`, usar `useSchedulePresenter()` como en el ejemplo anterior. Quitar toda la lógica de useEffect y state que ya está en el hook. La página quedará principalmente como un contenedor de `<CalendarHeader>`, `<CalendarView>` y modales, usando datos y handlers del presentador.
23. Ajustar la inyección de datos a subcomponentes: por ejemplo, `CalendarView` necesita `doctors, appointments, viewType, date` y algún handler `onSelectAppointment`. Ahora esos valores vienen del presentador. Pasarlos correctamente vía props. Asegurarse de que `CalendarView` ya no intenta por su cuenta manejar estado (si tenía alguno, intentar simplificarlo para que reciba todo por props).

24. Verificar que en este proceso no rompemos nada visual. Puede ser útil hacerlo incremental: conectar solo algunos aspectos, probar, luego seguir. Por ejemplo, primero conectar la lista de patients sin tocar el modal; ver que carga bien. Luego incorporar la lógica de abrir/cerrar modal desde presentador, etc.

25. **Aplicar SOLID en la nueva estructura:**
    Con presentadores y dominios en su lugar, revisar si cada uno cumple SRP y demás:

26. Asegurarse de que cada presentador tiene una sola responsabilidad. Si `SchedulePresenter` está muy grande manejando tanto citas como doctores, podrías optar por separarlo en sub-presentadores (pero probablemente esté bien junto porque son casos de uso relacionados).
27. Revisar DIP: definir *interfaces conceptuales* para servicios. Quizás crea archivos de tipos o JSDoc definiendo la interfaz esperada de, por ejemplo, un servicio de pacientes (métodos getAll, create, etc.). No es código ejecutable pero sirve de documentación. Considera modificar los presentadores para depender de abstracciones mínimas: por ejemplo, en vez de importar axios directamente en el presentador, que siga usando `patientService.create()` etc., y que internamente ese servicio use axios. Así si mañana se cambia axios por fetch, el presentador ni se entera.
28. Comprobar que no hay *code smells* como funciones muy largas en presentadores: si `useSchedulePresenter` quedó enorme, quizás dividir en hooks más pequeños (por ejemplo, un hook aparte `useModalState` para gestionar estados de modales genéricamente podría ser útil). Mantener el código limpio.

29. Chequear LSP/ISP: básicamente verificar que los componentes y presentadores usan solo lo necesario. Por ejemplo, si `PatientForm` es usado tanto en modal de nuevo paciente como en edición, que reciba props genéricas (initialData, onSubmit) para ser reutilizable en ambos casos,

en lugar de estar acoplado a uno. Eso respeta ISP y evita duplicar formularios para casos similares.

30. **Eliminar duplicaciones restantes (DRY final):**
    Después de reorganizar, es momento de buscar duplicación residual:

31. **Componentes duplicados funcionalmente:** Por ejemplo, ¿existe aún `Odontogram.jsx` tanto en `components/` raíz como en `components/patient/`? En la lista observé que había `components/Odontogram.jsx` y también uno dentro de patient. Si es el mismo, unificarlos. Quizás uno era prototipo y otro final. Decide cuál guardar y ajústalo.
32. **Hooks o funciones repetidas:** Si notas que `usePatientsPresenter` y `useDoctorsPresenter` comparten mucho código (ej: ambos tienen lógica de paginación, o ambos tienen lógica de filtrado de lista), considera factorizar esa parte común en un hook utilitario en `hooks/`. Por ejemplo, un `useListPresenter(service)` como mencionamos. Solo haz esto si realmente ves duplicación significativa y después de que todo funcione (refactorizar en seco primero, luego abstraer lo común).
33. **Estilos repetidos:** Recorre los componentes de UI buscando patrones repetidos de clases Tailwind. Por ejemplo, si muchos componentes usan la misma combinación de clases para contenedores, ese es un candidato a componente `<Card>` o a clase compuesta (Tailwind permite extender clases en la configuración). De igual forma, si hay colores hardcodeados en varias hojas (ej: muchos `text-blue-500`), considera añadir `theme.colors.primary` en Tailwind config y usar `text-primary` en su lugar. Centraliza definiciones para no repetir valores mágicos.
34. **Strings duplicados:** Mensajes de texto, títulos, etc. Si encuentras el mismo string en varios componentes (p.ej. "¿Seguro que deseas eliminar?" en modales distintos), podrías centralizarlos en un fichero de constantes de texto o simplemente en un solo componente de confirmación reutilizable.

35. Realiza pruebas manuales exhaustivas tras estos cambios para asegurarte de que nada se rompió durante la eliminación de duplicados. Idealmente, cada vez que quitas algo repetido, pruebas las partes relacionadas.

36. **Integrar o mejorar la librería de estilos:**
    Con la estructura y lógica ya refactorizadas, puedes abordar la parte visual para que luzca como el diseño esperado:

37. **Definir theme y estilos globales:** Si usas Tailwind, personaliza `tailwind.config.js` (colores, fonts). Si usas Chakra, configura el `ChakraProvider` con un theme custom. Aplica esos estilos base en tu app (por ejemplo, envolver la app en `<ChakraProvider theme={theme}>` si Chakra, o importar los archivos CSS de Tailwind).
38. **Refactorizar componentes UI para consistencia:** Revisa cada componente en `components/ui/` y asegúrate que están siendo utilizados en lugar de elementos HTML brutos en las vistas. Por ejemplo, reemplaza `<button className="...">` en todas las páginas por `<Button variant="...">` de tu librería. Haz lo mismo con modales: en `SchedulePage` usábamos un `<Modal>` custom, mantenlo pero asegúrate que su estilo coincide con el theme. Introduce nuevos componentes base si notaste su necesidad (por ejemplo, un `<Input>` común con estilos).
39. **Layout responsive:** Ajusta los componentes de layout (Sidebar, etc.) para que sean responsive si se requiere. Tal vez la sidebar se oculta en móvil y se abre como menú (en ese caso podrías usar

un componente Drawer de Chakra, o con Tailwind añadir clases para ocultar en sm y mostrar un botón menú).

40. **Detalles visuales de Zendenta:** Incorpora iconografía similar (quizá Zendenta usa iconos lineales modernos; puedes usar Heroicons que son similares). Añade esos iconos en Sidebar items, botones de acción, etc., para enriquecer la UI. Aplica colores suaves de fondo en elementos como cards (por ejemplo, en Zendenta las tarjetas quizás tienen fondo blanco con sombra y esquinas redondeadas; asegúrate de replicarlo).

41. **Probar la app con ojo de diseño:** comparar con las capturas de Zendenta. Ver si los espacios, alineaciones y colores coinciden aproximadamente. Ajustar clases Tailwind o props de Chakra hasta lograrlo.

42. **Verificación y pruebas finales:**

43. *Pruebas funcionales:* Navega por toda la aplicación comprobando que cada funcionalidad aún opera correctamente tras la refactorización: listar pacientes, añadir uno nuevo, editar, borrar; lo mismo con doctores; en la agenda, cambiar entre vista diaria/semanal (si existe), añadir cita, ver detalle, eliminar cita, etc. Presta atención a posibles errores en consola que indiquen props incorrectas o estados no manejados.
44. *Pruebas de código:* Si cuentas con tests unitarios o quieres agregar algunos, este es el momento. Prueba los presentadores con escenarios simulados (por ejemplo, mockear `appointmentService` para que `getAll` devuelva X citas y ver que `useSchedulePresenter` pone ese resultado en `appointments`). Prueba componentes críticos visualmente (snapshot tests o simplemente manual).
45. *Código limpio post-refactor:* Recorre brevemente el repo asegurándote de que no quedaron TODOs, console.logs, imports sin usar. Ejecuta la herramienta de lint: debería estar prácticamente libre de warnings ahora.
46. *Documentación:* Actualiza el README.md si es necesario para reflejar la nueva estructura (por ejemplo, indicando dónde añadir nuevos componentes, cómo está organizado el código). Puede ser útil para nuevos desarrolladores en el proyecto.
47. *Commit y code review:* Sube los cambios en etapas si es posible y haz que otro integrante del equipo revise el diff. La revisión de código ayudará a detectar cualquier cosita pasada por alto (por ejemplo, “olvidaste borrar el viejo file X” o “esta función ya no se usa”).

Siguiendo esta guía paso a paso, podrás refactorizar el proyecto de forma sistemática. El resultado será un CRM odontológico con una arquitectura sólida (MVP + DDD), un código más limpio y mantenible (SOLID, DRY, Clean Code aplicado) y una interfaz unificada y profesional gracias a la librería de estilos. Cada paso aproxima el código a las **mejores prácticas de la industria**, facilitando futuras extensiones del sistema y la incorporación de nuevos desarrolladores al equipo, que encontrarán un proyecto bien organizado y comprensible. ¡Manos a la obra con la refactorización! [8] [10]

---

[1] [4] [5] [10] [11] React la guía máxima de buenas prácticas jamás concebida, Principios SOLID, El acrónimo STUPID, Clean Code y Code Smell - DEV Community
https://dev.to/dennysjmarquez/react-la-guia-maxima-de-buenas-practicas-jamas-concebida-principios-solid-el-acronimo-stupid-clean-code-y-code-smell-53l7

[2] [3] [8] Principles for Building a High-Quality React Front-End | by Mykhailo (Michael) Hrynkevych | Medium
https://medium.com/@hrynkevych/principles-for-building-a-high-quality-react-front-end-86ef37122117

6 7 javascript - How to apply solid Principles and clean code in react in a component with too much logic - Stack Overflow
https://stackoverflow.com/questions/73508849/how-to-apply-solid-principles-and-clean-code-in-react-in-a-component-with-too-mu

9 Clean React with Feature-based Domain Driven Design - DEV Community
https://dev.to/tabernerojerry/clean-react-with-feature-based-domain-driven-design-1e07

12 Let’s talk about React and MVP. Trying new approach to our React... | by Gabriel Teles | PDVend Engineering | Medium
https://medium.com/pdvend-engineering/lets-talk-about-react-and-mvp-67ae35b8968c