Blueprint Arquitectónico: CRM B2B SaaS para Agencias y Freelancers

El desarrollo de un sistema de administración de clientes (CRM) para el sector B2B, específicamente orientado a agencias y freelancers, requiere una base técnica que trascienda la mera funcionalidad de una aplicación web estándar. La arquitectura debe abordar desafíos críticos de seguridad, integridad de datos financieros y aislamiento multi-inquilino desde el primer día. En un entorno donde la confianza del cliente es el activo más valioso, el diseño del sistema debe garantizar que los datos de una organización permanezcan inaccesibles para otras, incluso ante fallos imprevistos en las capas superiores de la aplicación.[1] La elección de Next.js como marco de trabajo front-end y servidor, en conjunto con Supabase como plataforma de persistencia y lógica de base de datos basada en PostgreSQL, permite implementar patrones de diseño de alto nivel que aseguran la escalabilidad y la robustez del producto final.[3]

1. Modelado de Datos: Estructura de Entidad-Relación y Auditoría

La arquitectura de datos de un CRM B2B debe estructurarse jerárquicamente, estableciendo límites claros para el aislamiento de los datos. El modelo propuesto no solo organiza la información, sino que define la lógica de negocio fundamental que permitirá al sistema escalar desde unos pocos usuarios hasta miles de organizaciones concurrentes. El núcleo de este diseño es la organización (Tenant), que actúa como el contenedor supremo de toda la información.[1]

1.1 El Esquema de Base de Datos Ideal

Para un CRM de agencias, la jerarquía de datos recomendada es: Organización -> Cliente -> Contactos -> Proyectos -> Facturas. Cada una de estas entidades debe diseñarse con integridad referencial estricta y tipos de datos que minimicen la deuda técnica futura.

Entidad	Descripción y Atributos Críticos	Tipos de Datos PostgreSQL	Propósito de Integridad
Organizations	Inquilino principal del sistema. Almacena configuración global, marca	id (UUID PK), name (TEXT), settings (JSONB)	Raíz del multi-tenancy.[1]

	blanca y suscripción.		
Clients	Empresas o individuos con los que la agencia hace negocios.	id (UUID PK), org_id (UUID FK), name (TEXT)	Aislamiento por organización.[2]
Contacts	Personas específicas asociadas a un cliente. Relación 1:N.	id (UUID PK), client_id (UUID FK), email (TEXT)	Trazabilidad de comunicación.[5]
Projects	Unidades de trabajo específicas vinculadas a un cliente.	id (UUID PK), client_id (UUID FK), budget (NUMERIC)	Centro de la gestión operativa.[6]
Invoices	Documentos financieros generados a partir de proyectos o tareas.	id (UUID PK), project_id (UUID FK), total (NUMERIC)	Integridad financiera exacta.[7]

En este modelo, el uso de UUID (Universally Unique Identifiers) en lugar de enteros autoincrementales es fundamental. Los UUID v4 (aleatorios) o v7 (basados en tiempo) previenen ataques de enumeración de recursos y facilitan la combinación de datos o la migración entre sistemas sin colisiones de claves.[8] Para los campos financieros como el presupuesto de proyectos o los montos de facturación, es imperativo utilizar el tipo NUMERIC(precision, scale). PostgreSQL garantiza que los cálculos con este tipo de datos sean exactos, a diferencia de los tipos de punto flotante (REAL o DOUBLE PRECISION) que pueden introducir errores de redondeo acumulativos, lo cual es inaceptable en contextos contables o de auditoría legal.[7]

La relación entre estas entidades se mantiene mediante claves foráneas (Foreign Keys) con restricciones de borrado adecuadas. Por ejemplo, al eliminar una organización, se debe decidir si se realiza un ON DELETE CASCADE para limpiar todos los registros asociados o si se prefiere un bloqueo preventivo para asegurar que no se pierdan datos históricos accidentalmente.[2] La arquitectura debe favorecer la desnormalización estratégica del organization_id en tablas críticas para simplificar las consultas de seguridad (RLS), un punto

que se analizará en profundidad en la sección de multi-tenancy.[11]

1.2 Implementación del Historial de Cambios (Audit Logs)

La trazabilidad de los datos no es solo una característica de cumplimiento, sino una herramienta de depuración y soporte esencial. Existen tres métodos principales para implementar logs de auditoría en PostgreSQL: registro nativo, replicación lógica y disparadores (triggers).[12] Para un CRM en Supabase, el método basado en disparadores es el más robusto debido a su naturaleza síncrona y su capacidad para capturar el contexto del usuario autenticado en el momento de la transacción.[13]

La estrategia recomendada utiliza una tabla centralizada de auditoría que aprovecha el tipo de dato JSONB. Este tipo permite almacenar el estado anterior (OLD) y el nuevo (NEW) de una fila de forma estructurada sin necesidad de que la tabla de auditoría cambie cada vez que se modifica el esquema de las tablas de negocio.[15]

SQL

```
CREATE SCHEMA audit;

CREATE TABLE audit.logged_changes (
  id BIGSERIAL PRIMARY KEY,
  table_name TEXT NOT NULL,
  record_id UUID NOT NULL,
  action TEXT NOT NULL, -- INSERT, UPDATE, DELETE
  old_data JSONB,
  new_data JSONB,
  changed_by UUID REFERENCES auth.users(id),
  changed_at TIMESTAMPTZ DEFAULT now()
);

CREATE INDEX idx_audit_record ON audit.logged_changes (table_name, record_id);
```

Para poblar esta tabla de manera automática, se implementa una función PL/pgSQL que se dispara después de cada operación de modificación de datos. Un aspecto crítico para un Arquitecto Senior es la detección inteligente de cambios. No tiene sentido registrar una actualización si los datos no han cambiado realmente. El uso de operadores de diferencia de JSONB (old_data - new_data) permite identificar exactamente qué columnas fueron modificadas, optimizando el espacio de almacenamiento y la claridad de los logs.[15] El volumen de estos registros puede crecer exponencialmente; por lo tanto, es vital utilizar BIGSERIAL

para la clave primaria y planificar estrategias de particionamiento de tablas por tiempo (por ejemplo, particiones mensuales) para mantener el rendimiento de las consultas a largo plazo.[15]

2. Seguridad y Multi-tenancy: El Poder de Row Level Security (RLS)

El mayor riesgo en una aplicación B2B SaaS es la fuga de datos entre inquilinos. Un error en el front-end que accidentalmente pida el ID de un cliente de otra organización jamás debería devolver datos.[1] En la arquitectura tradicional, esto se manejaba añadiendo cláusulas WHERE organization_id =? en cada consulta del código de la aplicación. Sin embargo, este enfoque es propenso a errores humanos. Supabase resuelve esto desplazando la responsabilidad de la seguridad a la base de datos mediante Row Level Security (RLS).[2]

2.1 Estrategia de Implementación de RLS

RLS funciona como un filtro automático e invisible que PostgreSQL aplica a cada consulta antes de que el motor de ejecución recupere los datos de los discos. Para implementar una arquitectura multi-inquilino hermética, el sistema debe identificar primero al usuario que realiza la petición a través de su JWT (JSON Web Token) y luego validar su acceso a la organización correspondiente.[2]

Existen dos patrones principales para verificar la pertenencia a una organización en RLS:

1. **Join Dinámico:** La política verifica en tiempo real si el auth.uid() del usuario existe en una tabla de membresías para la organización del registro solicitado.[18]
2. **Custom Claims (JWT):** El ID de la organización se incluye como un metadato en el token del usuario. Esto permite un acceso instantáneo sin necesidad de consultas adicionales a la base de datos.[2]

Método	Rendimiento	Complejidad de Implementación	Recomendación
Subconsultas IN/EXISTS	Medio (requiere índices óptimos)	Baja	Ideal para inicios rápidos y MVPs.[18]
JWT App Metadata	Muy Alto (lectura de memoria)	Media	Recomendado para escalabilidad masiva.[19]

Seguridad por Schema	Alto	Muy Alta (gestión de migraciones)	Solo para requisitos de aislamiento extremo.

2.2 Patrón SQL Estándar para Aislamiento Total

Para asegurar que un usuario solo pueda interactuar con los datos de su propia organización, se habilita RLS en todas las tablas y se definen políticas específicas. A continuación se presenta el patrón SQL definitivo que garantiza que ni siquiera un fallo masivo en el código Next.js pueda exponer datos ajenos:

SQL

```sql
-- Habilitar RLS en la tabla de proyectos
ALTER TABLE public.projects ENABLE ROW LEVEL SECURITY;

-- Política para permitir solo la lectura de proyectos de la propia organización
CREATE POLICY "Users can only view their organization's projects"
ON public.projects
FOR SELECT
TO authenticated
USING (
  organization_id IN (
    SELECT organization_id
    FROM public.memberships
    WHERE user_id = auth.uid()
  )
);

-- Política para inserción (asegura que el organization_id sea correcto)
CREATE POLICY "Users can only create projects in their organization"
ON public.projects
FOR INSERT
TO authenticated
WITH CHECK (
  organization_id IN (
    SELECT organization_id
    FROM public.memberships
    WHERE user_id = auth.uid()
  )
```

);

Este patrón utiliza la cláusula USING para filtrar lo que el usuario puede ver y WITH CHECK para validar lo que el usuario intenta insertar o modificar.[2] Es una práctica de seguridad crítica que el organization_id sea inmutable o esté estrictamente controlado tras la creación del registro para evitar ataques de "secuestro de inquilinos", donde un usuario malintencionado intenta mover un registro de una organización a otra cambiando el ID de la clave foránea.[11]

2.3 Optimización del Rendimiento en RLS

Un sistema de RLS mal diseñado puede destruir el rendimiento de la aplicación. Cuando PostgreSQL tiene que ejecutar una subconsulta para cada fila en una tabla de un millón de registros, la latencia se dispara. Para prevenir esto, es obligatorio indexar todas las columnas referenciadas en las políticas de RLS, especialmente user_id y organization_id.[2]

Además, el uso de funciones con el atributo SECURITY DEFINER permite ejecutar verificaciones de permisos con privilegios elevados de forma controlada, evitando recursiones infinitas en las políticas de RLS.[18] Un truco avanzado de optimización consiste en envolver las funciones de verificación en sentencias SELECT. Esto permite al optimizador de PostgreSQL ejecutar la función una sola vez por consulta (initPlan) en lugar de una vez por fila, reduciendo el tiempo de ejecución de segundos a milisegundos en conjuntos de datos grandes.[18]

3. Del MVP al Feature Creep: Enfoque en el Valor Real

En el desarrollo de un CRM B2B SaaS, el mayor peligro no es la competencia, sino la complejidad innecesaria. El "Feature Creep" consume recursos preciosos y confunde a los usuarios iniciales. Un MVP exitoso debe resolver un problema específico de manera excepcional antes de expandirse.[21]

3.1 Definición de un MVP Estricto

El MVP para un CRM de agencias y freelancers no debe ser una versión "pobre" del producto final, sino la versión más pequeña que genere un retorno de inversión (ROI) claro para el usuario. El objetivo es automatizar los procesos manuales que consumen el 36% del tiempo administrativo de un freelancer.[5]

Para un CRM de gestión, las funcionalidades CRUD (Crear, Leer, Actualizar, Borrar) son la infraestructura básica, no la propuesta de valor. El valor reside en cómo el sistema orquestar esos datos para mejorar el flujo de caja y la eficiencia operativa.[6]

3.2 Las 4 Funcionalidades Nucleares del CRM

Para que una herramienta de gestión sea indispensable desde el día uno, debe implementar

estas cuatro capacidades:

1. **Pipeline de Proyectos con Seguimiento Visual:** Las agencias necesitan ver dónde se atasca el trabajo. Un tablero Kanban o una lista de hitos que permita visualizar el estado de cada proyecto (ej. "En Espera de Feedback", "En Desarrollo", "Completado") es vital para la gestión de la capacidad y los recursos.[5]
2. **Ciclo Integrado de Tiempo y Facturación:** La desconexión entre el trabajo realizado y el cobro es la principal causa de fallos en pequeñas agencias. El MVP debe permitir registrar horas o entregables y convertirlos automáticamente en una factura profesional. Este flujo elimina errores de entrada de datos y acelera los ciclos de pago.[6]
3. **Portal de Cliente de Marca Blanca (White-label):** Un CRM B2B no es solo para el equipo interno; es la interfaz de la agencia con el mundo. Un portal donde los clientes puedan revisar propuestas, firmar contratos electrónicamente y pagar facturas bajo el dominio y colores de la agencia proyecta una profesionalidad que justifica tarifas más altas.[24]
4. **Automatización de Seguimiento y Recordatorios:** El software debe trabajar mientras el freelancer duerme. Esto incluye recordatorios automáticos de facturas impagadas, alertas de seguimiento para propuestas que el cliente ha abierto pero no ha firmado, y notificaciones de hitos de proyecto.[5]

Funcionalidad	Valor para el Negocio	Impacto en el Usuario
Pipeline Visual	Visibilidad de ingresos futuros	Reducción de la ansiedad operativa.[6]
Invoicing Pipeline	Mejora drástica del flujo de caja	Menos tiempo en tareas administrativas.[6]
White-label Portal	Aumento de la retención de clientes	Percepción de valor y confianza superior.[29]
Smart Automations	Escalabilidad sin aumentar personal	Eliminación de tareas repetitivas y olvidos.[5]

Al centrarse en estas cuatro áreas, el sistema se convierte en el "sistema operativo" de la agencia, pasando de ser una base de datos pasiva a un motor de crecimiento activo.[5]

4. Deuda Técnica Común y Estrategias de Mitigación

Un Arquitecto Senior debe anticipar los errores que los desarrolladores menos experimentados cometen habitualmente. En el contexto de Next.js y Supabase, estos errores

suelen manifestarse en la gestión de la integridad de los datos y en la arquitectura del estado de la aplicación.

4.1 El Error de las Operaciones no Atómicas

Un error clásico es realizar múltiples mutaciones de base de datos de forma secuencial en el código del servidor (Next.js Server Actions) sin un envoltorio transaccional. Por ejemplo, al crear un nuevo cliente, se intenta insertar el registro del cliente y luego un contacto inicial. Si la segunda operación falla, el sistema queda con un cliente "fantasma" sin contacto asociado.[31]

Prevención: La lógica de negocio compleja que involucra múltiples tablas debe encapsularse en Funciones de PostgreSQL (RPC). Las funciones de base de datos son atómicas por definición: o todo el bloque de código entre BEGIN y END se ejecuta correctamente, o se realiza un ROLLBACK total. Esto garantiza que la base de datos nunca quede en un estado inconsistente.[31]

$$Consistencia = \text{Operaciones Exitosas} \vee \text{Estado Original}$$

La atomicidad es el primer pilar de las propiedades ACID (Atomicity, Consistency, Isolation, Durability) que un CRM B2B no puede comprometer.[31]

4.2 Mala Gestión de Estados Globales y RSC

Con la llegada del App Router en Next.js, muchos desarrolladores intentan aplicar patrones de estado global (como Redux o Zustand) a datos que deberían vivir en el servidor. Esto provoca problemas de hidratación, aumenta el tiempo de carga debido a paquetes de JavaScript masivos y rompe la capacidad de compartir URLs con filtros aplicados (por ejemplo, una búsqueda específica de facturas vencidas).[3]

Prevención: Utilizar React Server Components (RSC) para la obtención de datos y la URL para el estado de navegación. La mayoría de los datos de un CRM son de lectura intensiva. Al recuperar los datos directamente en el servidor y pasarlos a los componentes, se elimina la necesidad de estados de carga complejos en el cliente y se mejora el SEO y la seguridad.[4] Para filtros y paginación, se debe utilizar URLSearchParams. Esto no solo es más ligero, sino que permite que los usuarios guarden marcadores o compartan vistas específicas del CRM de forma natural.[34]

4.3 Falta de Validación en el Backend

Confiar únicamente en la validación del formulario en el front-end es una negligencia de seguridad. Un atacante puede enviar peticiones directamente a los puntos finales de la API o Server Actions saltándose cualquier control visual. Sin una validación estricta en el servidor,

se pueden introducir datos corruptos, ataques de inyección o valores financieros inválidos.[36]

Prevención: Implementar una capa de validación de esquemas con librerías como Zod en el punto de entrada de cada Server Action. Zod permite definir el contrato de datos esperado y rechazar cualquier petición que no lo cumpla antes de que toque la base de datos. Esta práctica proporciona además seguridad de tipos (Type Safety) en todo el ciclo de vida de la petición.[36]

TypeScript

```
// Ejemplo de validación robusta en Server Action
const InvoiceSchema = z.object({
  client_id: z.string().uuid(),
  amount: z.number().positive(),
  due_date: z.date().min(new Date())
});

export async function createInvoice(formData: FormData) {
  const parsed = InvoiceSchema.safeParse(Object.fromEntries(formData));
  if (!parsed.success) return { errors: parsed.error.flatten() };

  // Solo aquí llamamos a Supabase
  const { data, error } = await supabase.rpc('create_invoice_atomic', parsed.data);
}
```

5. Escalabilidad y Futuro: Preparando el Sistema para el Crecimiento

Una arquitectura escalable no es solo la que soporta muchos usuarios, sino la que permite al equipo de desarrollo iterar sin romper el sistema. En el contexto de Supabase, esto implica una gestión cuidadosa de los recursos de la base de datos y la infraestructura serverless.

5.1 Gestión de Recursos y Conexiones

A medida que el CRM crece, el número de conexiones simultáneas a PostgreSQL puede convertirse en un cuello de botella. El uso de Serverless Functions (como los Server Actions de Next.js o las Edge Functions de Supabase) tiende a abrir y cerrar conexiones rápidamente. Un Arquitecto Senior debe configurar un Pooler de Conexiones (como Supavisor) para gestionar estas conexiones de manera eficiente, evitando el agotamiento de la memoria del

servidor de base de datos.[40]

5.2 Estrategias de Caching y Rendimiento de UI

Para dashboards de CRM que muestran datos complejos, el uso de "Streaming SSR" es vital. En lugar de esperar a que todas las consultas (clientes, ingresos del mes, tareas pendientes) terminen para enviar la página al usuario, Next.js permite enviar el marco de la página inmediatamente y "transmitir" los datos de cada widget a medida que están listos.[3] Esto reduce la latencia percibida y mejora significativamente la experiencia del usuario en dispositivos móviles o conexiones lentas.

Componente del Dashboard	Estrategia de Carga	Justificación Técnica
KPIs de Ingresos	Streaming con Suspense	Cálculo intensivo, no debe bloquear el renderizado inicial.[3]
Lista de Tareas	RSC con Revalidación por Tag	Datos que cambian frecuentemente tras acciones del usuario.[41]
Gráficos de Pipeline	Client Component con Lazy Loading	Bibliotecas de visualización pesadas que solo necesitan ejecutarse en el navegador.[3]
Configuración de Perfil	Estático con Revalidación bajo Demanda	Datos que rara vez cambian, optimizados para entrega ultra-rápida.

5.3 Mantenimiento de la Integridad a Largo Plazo

Finalmente, la escalabilidad se ve amenazada por la degradación de la calidad de los datos. La implementación de pruebas automatizadas que validen las políticas de RLS es obligatoria. Un Arquitecto Senior no confía en que las políticas funcionen por siempre; utiliza herramientas de testing para simular diferentes roles de usuario y asegurar que las barreras multi-inquilino permanezcan infranqueables a pesar de nuevas migraciones o cambios en el código.[1]

La arquitectura aquí descrita no es solo un conjunto de tecnologías, sino una filosofía de desarrollo que prioriza la seguridad y la integridad. Al construir sobre los hombros de

PostgreSQL y Next.js, y aplicar patrones de RLS avanzados y transacciones atómicas, se crea un sistema capaz de sostener el crecimiento de una agencia desde su primer cliente hasta su madurez como empresa líder en el sector B2B SaaS.[22]

Obras citadas

1. Enforcing Row Level Security in Supabase: A Deep Dive into ..., fecha de acceso: febrero 14, 2026, https://dev.to/blackie360/-enforcing-row-level-security-in-supabase-a-deep-dive-into-lockins-multi-tenant-architecture-4hd2
2. Supabase Row Level Security (RLS): Complete Guide (2026) - DesignRevision, fecha de acceso: febrero 14, 2026, https://designrevision.com/blog/supabase-row-level-security
3. Next.js SaaS Dashboard Development: Scalability & Best Practices - Ksolves, fecha de acceso: febrero 14, 2026, https://www.ksolves.com/blog/next-js/best-practices-for-saas-dashboards
4. Client vs Server Components in Next.js 15 — A Deep Dive (With Real Examples) - Medium, fecha de acceso: febrero 14, 2026, https://medium.com/@technoharsh21/client-vs-server-components-in-next-js-15-a-deep-dive-with-real-examples-f40af6c9d12e
5. Best CRM Software for Freelancers in 2026 - Plutio, fecha de acceso: febrero 14, 2026, https://www.plutio.com/solutions/freelancers/crm
6. Project management for freelancers: best tools and AI features for 2026, fecha de acceso: febrero 14, 2026, https://monday.com/blog/project-management/project-management-for-freelancers/
7. Postgres Decimal data types - Neon Docs, fecha de acceso: febrero 14, 2026, https://neon.com/docs/data-types/decimal
8. Understanding PostgreSQL data types for data optmization - pgroll, fecha de acceso: febrero 14, 2026, https://pgroll.com/blog/understanding-postgresql-data-types
9. Postgres Auditing in 150 lines of SQL - Supabase, fecha de acceso: febrero 14, 2026, https://supabase.com/blog/postgres-audit
10. Efficient PostgreSQL Database Design and Data Types | by Java Jedi | Medium, fecha de acceso: febrero 14, 2026, https://java-jedi.medium.com/efficient-postgresql-database-design-and-data-types-d7b50311fe17
11. multi-tenant backend - tenant id in every table or join from linked tables : r/Supabase, fecha de acceso: febrero 14, 2026, https://www.reddit.com/r/Supabase/comments/1ku8c7l/multitenant_backend_tenant_id_in_every_table_or/
12. Postgres Audit Logging Guide - Bytebase, fecha de acceso: febrero 14, 2026, https://www.bytebase.com/blog/postgres-audit-logging/
13. Supabase CDC Options: Triggers, Webhooks, Realtime Compared - Stacksync, fecha de acceso: febrero 14, 2026,

https://www.stacksync.com/blog/supabase-cdc-options-triggers-webhooks-realtime-compared
14. Creating and Using Database Triggers in Supabase - Sakib Rahman, fecha de acceso: febrero 14, 2026, https://rsakib.com/blogs/creating-using-database-triggers-supabase
15. Let's Build Production-Ready Audit Logs in PostgreSQL | by Sehban ..., fecha de acceso: febrero 14, 2026, https://medium.com/@sehban.alam/lets-build-production-ready-audit-logs-in-postgresql-7125481713d8
16. How to Build PostgreSQL Triggers for Audit - OneUptime, fecha de acceso: febrero 14, 2026, https://oneuptime.com/blog/post/2026-01-30-postgresql-triggers-audit/view
17. Authorization via Row Level Security | Supabase Features, fecha de acceso: febrero 14, 2026, https://supabase.com/features/row-level-security
18. Troubleshooting | RLS Performance and Best Practices - Supabase Docs, fecha de acceso: febrero 14, 2026, https://supabase.com/docs/guides/troubleshooting/rls-performance-and-best-practices-Z5Jjwv
19. supabase-community/supabase-custom-claims: How to ... - GitHub, fecha de acceso: febrero 14, 2026, https://github.com/supabase-community/supabase-custom-claims
20. Optimizing RLS Performance with Supabase(postgres) | by AntStack Inc. - Medium, fecha de acceso: febrero 14, 2026, https://medium.com/@antstack/optimizing-rls-performance-with-supabase-postgres-fa4e2b6e196d
21. SaaS MVP Development: A Complete Guide for 2025 - Talentica Software, fecha de acceso: febrero 14, 2026, https://www.talentica.com/blogs/saas-mvp-development/
22. Minimum Viable Product (MVP): The Complete Guide for 2026 - Monday.com, fecha de acceso: febrero 14, 2026, https://monday.com/blog/rnd/mvp-in-project-management/
23. What is an MVP? Everything you need to know about the minimum viable product, fecha de acceso: febrero 14, 2026, https://agence-scroll.com/en/blog/what-is-an-mvp-minimum-viable-product
24. Top 9 CRM with Client Portal Features for Digital Agencies - OneSuite, fecha de acceso: febrero 14, 2026, https://onesuite.io/blog/crm-with-client-portal/
25. Why Startups Should Use the MVP Approach for CRM and ERP Development - Corpsoft.io, fecha de acceso: febrero 14, 2026, https://corpsoft.io/2025/01/10/how-to-ensure-successful-crm-and-erp-implementation-with-mvp/
26. Best CRM Features for Freelancers - ClientManager, fecha de acceso: febrero 14, 2026, https://www.clientmanager.io/blog/best-crm-features-for-freelancers
27. Best 6 CRM for Freelancers to Manage Clients Like a Pro - WP ERP, fecha de acceso: febrero 14, 2026, https://wperp.com/168435/crm-for-freelancers/
28. White-Label Client Portal: Empower Your Clients, Strengthen Your Brand -

Vendasta, fecha de acceso: febrero 14, 2026, https://www.vendasta.com/blog/white-label-client-portal/
29. What is white label client portal software? Level up your B2B SaaS brand | Moxo, fecha de acceso: febrero 14, 2026, https://www.moxo.com/blog/white-label-client-portal-software
30. Best CRM for B2B SaaS, fecha de acceso: febrero 14, 2026, https://www.folk.app/articles/best-crm-b2b-saas
31. Data Integrity First: Mastering Transactions in Supabase SQL for Reliable Applications, fecha de acceso: febrero 14, 2026, https://dev.to/damasosanoja/data-integrity-first-mastering-transactions-in-supabase-sql-for-reliable-applications-2dbb
32. Architecture Review: Node.js API vs. SvelteKit Server Actions for multi-table inserts (Supabase) : r/sveltejs - Reddit, fecha de acceso: febrero 14, 2026, https://www.reddit.com/r/sveltejs/comments/1qj5njm/architecture_review_nodejs_api_vs_sveltekit/
33. Should I go with server actions , RPC calls or prisma for relational inserts? - Reddit, fecha de acceso: febrero 14, 2026, https://www.reddit.com/r/Supabase/comments/1med36o/should_i_go_with_server_actions_rpc_calls_or/
34. Best Approaches for Managing Global State in Next.js Apps Without Overhead? - Reddit, fecha de acceso: febrero 14, 2026, https://www.reddit.com/r/nextjs/comments/1n1fvb9/best_approaches_for_managing_global_state_in/
35. Getting Started: Server and Client Components - Next.js, fecha de acceso: febrero 14, 2026, https://nextjs.org/docs/app/getting-started/server-and-client-components
36. Next.js form validation on the client and server with Zod - DEV Community, fecha de acceso: febrero 14, 2026, https://dev.to/bookercodes/nextjs-form-validation-on-the-client-and-server-with-zod-lbc
37. Data Validation in the Next.js Supabase Turbo kit - MakerKit, fecha de acceso: febrero 14, 2026, https://makerkit.dev/docs/next-supabase-turbo/security/data-validation
38. How to Handle Forms in Next.js with Server Actions and Zod for Validation - freeCodeCamp, fecha de acceso: febrero 14, 2026, https://www.freecodecamp.org/news/handling-forms-nextjs-server-actions-zod/
39. Type-Safe Server Actions in Next.js with Zod, fecha de acceso: febrero 14, 2026, https://yournextstore.com/blog/typesafe-server-actions-zod-nextjs
40. Introducing: Postgres Best Practices - Supabase, fecha de acceso: febrero 14, 2026, https://supabase.com/blog/postgres-best-practices-for-ai-agents
41. Server Actions and Mutations - Data Fetching - Next.js, fecha de acceso: febrero 14, 2026, https://nextjs.org/docs/13/app/building-your-application/data-fetching/server-actions-and-mutations