Ingeniería Cortex HIVE

Arquitectura de plataforma y decisiones de diseño

Neftali Yagua

Julio 2026

Índice

Capítulo 1

Prefacio

Este documento consolida la **arquitectura de ingeniería** de Cortex/HIVE: la plataforma multi-tenant ensamblada por plugins Python, con interfaz administrativa declarativa (estilo Filament) y contratos REST estables.

Está dirigido a desarrolladores, arquitectos y agentes de automatización que necesitan entender **qué es la plataforma**, **cómo se divide el código** y **qué decisiones están acordadas** frente a lo que aún está en diseño o scaffold.

Autor: Neftali Yagua
Revisión: Julio 2026
Documentación en línea: https://docs.cortex-ia.com.co/

Capítulo 2

Resumen ejecutivo

Cortex/HIVE separa de forma explícita dos familias de código:

- **Dependiente:** módulos internos del framework (autenticación, tenant, capa IO, UI declarativa, configuración). Sin ellos, ninguna aplicación Cortex puede operar.
- **Independiente:** plugins de dominio de negocio (reservas, pagos, contabilidad, ventas POS, tienda e-commerce, etc.) que consumen la plataforma vía SPI y REST, sin reimplementar auth ni shell gráfico.

La REST API (`/api/v1`) es la **fuente de verdad** del negocio. MCP, webhooks y la UI declarativa son fachadas sobre esa capa. El prototipo histórico UnoSport Club validó multi-panel y CUS; **no** define el dominio de los módulos de negocio.

Estado	Significado en este documento
Implementado	Código presente en el monorepo y usable en desarrollo
Diseño	ADR aceptado; implementación pendiente o parcial
Scaffold	Plugin o endpoint de ejemplo con fixtures en memoria

Capítulo 3

Alcance

Incluye: taxonomía Dependiente/Independiente, ciclo de vida del plugin (discovery, hooks DIP, bootstrap vs runtime), stack HIVE, multi-panel, Cortex Panel (CUS), panel admin, módulo IO, autenticación, MCP, persistencia multi-tenant, dominio booking genérico y NFRs con Redis.

Excluye: procedimientos operativos de despliegue, guías paso a paso de contribución y el plan de distribución pip (ADR 002, propuesto), que se documentan en el sitio MkDocs.

Este PDF reproduce una **selección pedagógica** de ADRs (001–011, 018–019). El catálogo completo (001–020) está en el repositorio y en https://docs.cortex-ia.com.co/adr/.

Capítulo 4

Arquitectura Dependiente vs Independiente

La pizarra de producto define **dónde vive el código**, no solo la distinción genérica entre «core» y «plugins».

Columna	Qué es	Dónde vive
Dependiente	Módulos internos del framework — la plataforma no funciona sin ellos	`framework/cortex_framework/*`, `packages/cortex-panel-*`, `framework/web*`
Independiente	Dominios de negocio en plugins pip o sideload	`plugins/*/cortex_plugin_*`

Los plugins consumen módulos Dependiente; no reimplementan OAuth, panel shell, widgets base ni capa IO.

4.1 Mapa general del sistema

Leyenda: Visión holística Cortex/HIVE — actores, capas, plugins, datos. **Estado:** Parcial (ver leyendas por ADR).

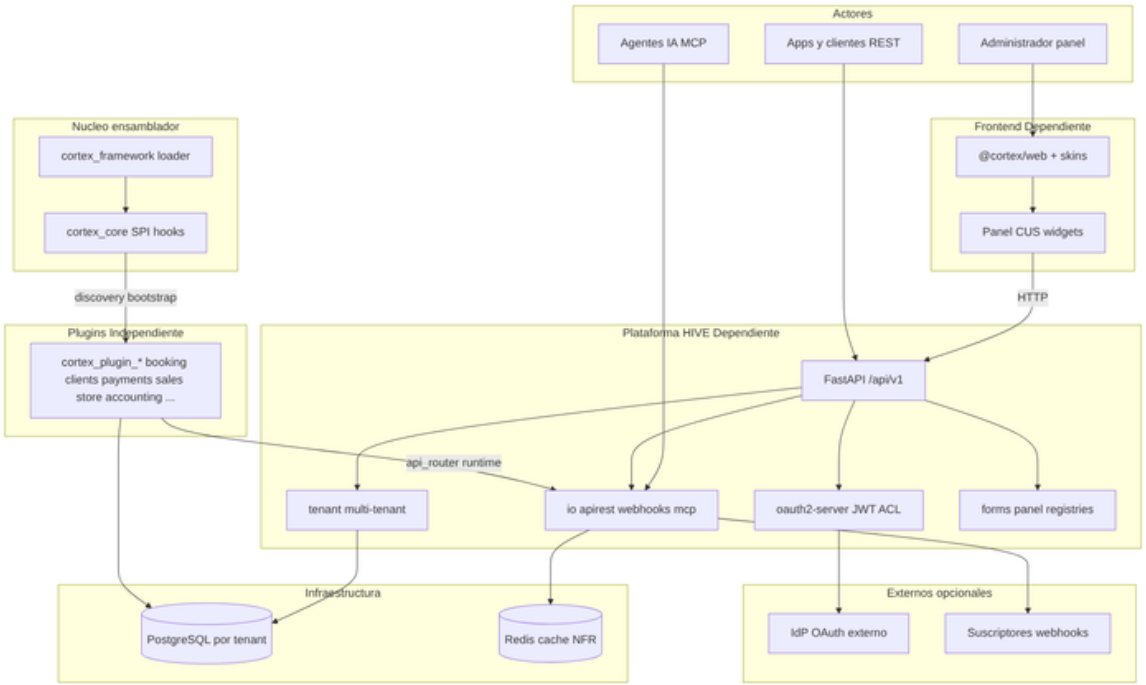

Figura 4.1: Diagrama 1

4.2 Módulos Dependiente

Módulo	Responsabilidad	Estado
oauth2-server	OAuth2, ACL, permisos, JWT (interno por defecto; IdP externo opcional)	Diseño — ADR 005
tenant	Multi-tenant, `X-Tenant-Id`, PostgreSQL por tenant	Parcial
io	apirest, webhooks, mcp, apirest2mcp	Parcial — ADR 010
ui / panel	Forms, componentes, dashboards, skins, multi-panel CUS	Implementado
observability	Logs, trazas, métricas	Diseño
cms	Contenido gestionado para apps host	Diseño
reportes	Informes transversales sobre datos de plugins	Diseño
integraciones	Conectores vía submódulo `io`	Diseño
configuración	Settings tenant, panel Control	Implementado

4.3 Plugins Independiente (dominio)

Dominio	Plugin	Notas
Reserva	`cortex_plugin_booking`	ADR 008 — Resource / Slot / Booking
Pagos	`cortex_plugin_payments`	
Facturación	`cortex_plugin_billing`	
Contabilidad	`cortex_plugin_accounting`	PoC más avanzado
Descuentos	`cortex_plugin_discounts`	
Tarifas	`cortex_plugin_pricing`	
Clientes	`cortex_plugin_clients`	

Dominio	Plugin	Notas
Ventas (POS)	`cortex_plugin_sales`	Cajeros, apertura/cierre, mostrador
Tienda (e-commerce)	`cortex_plugin_store`	Catálogo, carrito, checkout
Eventos	`cortex_plugin_events`	
Suscripciones	`cortex_plugin_subscriptions`	
Bot / IA	Consumen `io.mcp`	Sin duplicar reglas de negocio

Ventas y Tienda son plugins distintos. Comparten integraciones (pagos, clientes, tarifas) exclusivamente vía REST.

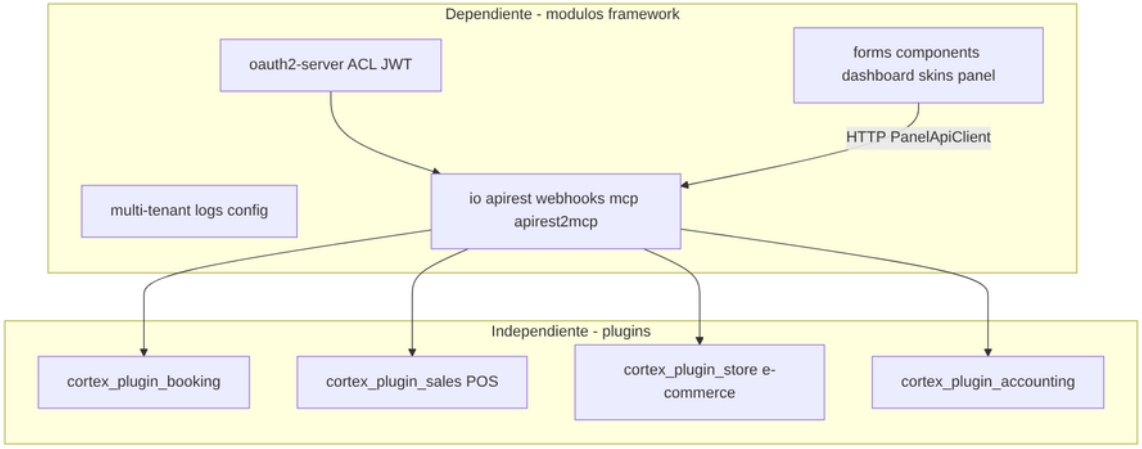

Figura 4.2: Diagrama 2

Leyenda: Taxonomía Dependiente vs Independiente. La UI solo habla HTTP con `io.apirest`. Estado: parcial — registries y apirest implementados; oauth2-server e io completo en diseño.

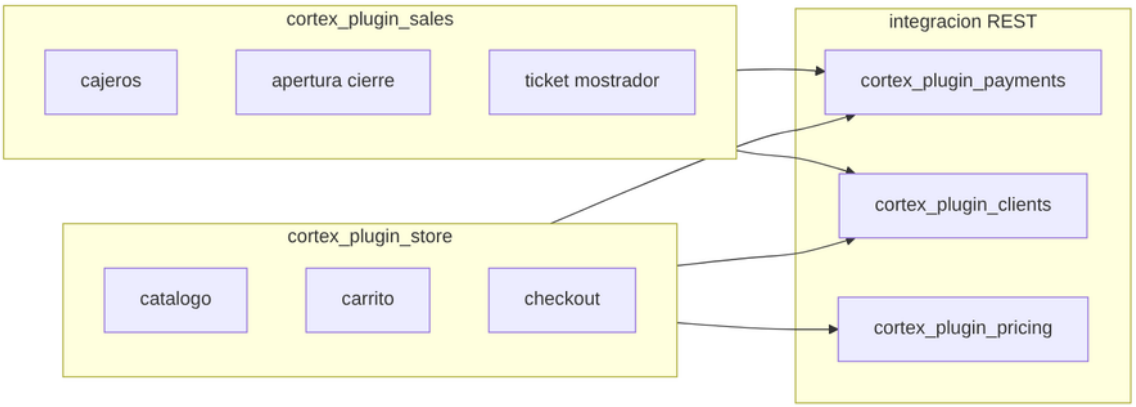

Figura 4.3: Diagrama 3

Leyenda: Ventas (POS) y Tienda (e-commerce) como dominios separados. Integración solo por API.

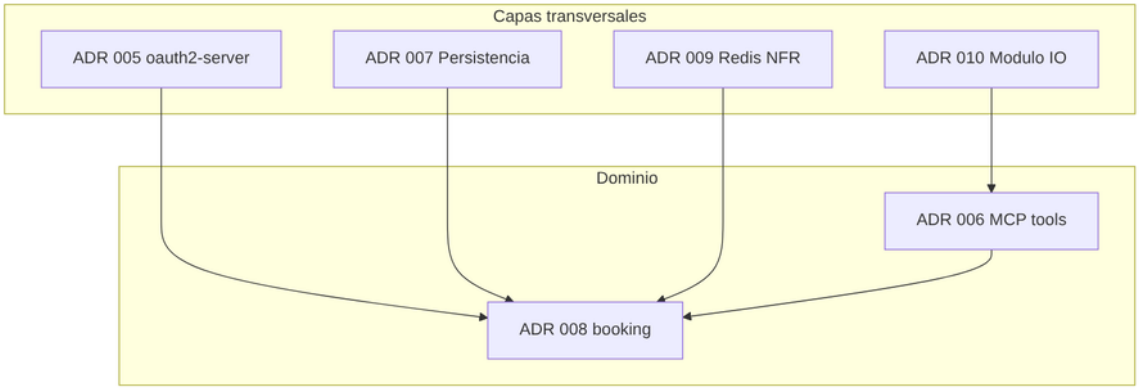

Figura 4.4: Diagrama 4

Leyenda: Dependencias entre ADRs transversales y dominio. Estado: diseño de ingeniería.

Capítulo 5

Ciclo de vida del plugin

Principio: HIVE no conoce las reglas de negocio ni el código React de un plugin. Solo conoce contratos SPI, hooks de registro y rutas HTTP genéricas de plataforma. Al descubrir un plugin, este **se coloca** en la aplicación: panels, módulos CUS, formularios y API REST — usando herramientas Dependientes (forms, UI/panel, widgets, `io.apirest`).

El arranque en `framework/cortex_framework/plugins/loader.py` tiene **dos fases**:

1. **Instancia:** `create_plugin()` → `PluginProtocol` (`api_router`, `resource_paths`) — lógica REST de dominio.
2. **Bootstrap:** hooks opcionales → el framework inyecta implementaciones concretas de los registradores definidos en `core/cortex_core/registrar.py` (inversión de dependencias).

5.1 Bootstrap vs runtime

Fase	Mecanismo	Responsabilidad del plugin
Bootstrap	`register_panels`, `register_dashboards`, `register_forms`, (diseño) `register_mcp_tools`	Dónde aparece en panels, pantallas JSON, formularios, tools MCP
Runtime	`api_router()`, `resource_paths()`	Endpoints y lógica de dominio bajo `/api/v1`

5.2 Hooks disponibles

Hook	Tipo inyectado (`core`)	Qué publica
`register_panels(registrar)`	`PanelRegistrar`	`PanelDefinition`: path UI, título, namespace API, `PanelAuthPolicy`
`register_dashboards(registrar)`	`DashboardRegistrar`	Manifest + dashboards JSON por `panel_id` / `module_id`
`register_forms(registrar)`	`FormRegistrar`	`FormDefinition` → `GET /api/v1/forms/{formId}`
`register_mcp_tools(registrar)` *(diseño)*	`McpToolRegistrar`	Tools MCP sobre REST existente (ADR 006)

El plugin **compone** pantallas con widgets conocidos del framework (`form`, `data-table`, `reservation-flow`, `section`, …); **no** aporta componentes React propios.

Límite del desacoplamiento: HIVE sí conoce los tipos de widget y las rutas HTTP de plataforma; lo desconocido es el dominio de negocio y el JSON concreto de cada módulo.

Leyenda: Ciclo completo discovery → bootstrap → ensamblado HTTP → shell React. **Actores:** discovery, plugin, registries, API, shell. **Estado:** Implementado (hooks panels/dashboards/forms); `register_mcp_tools` en diseño.

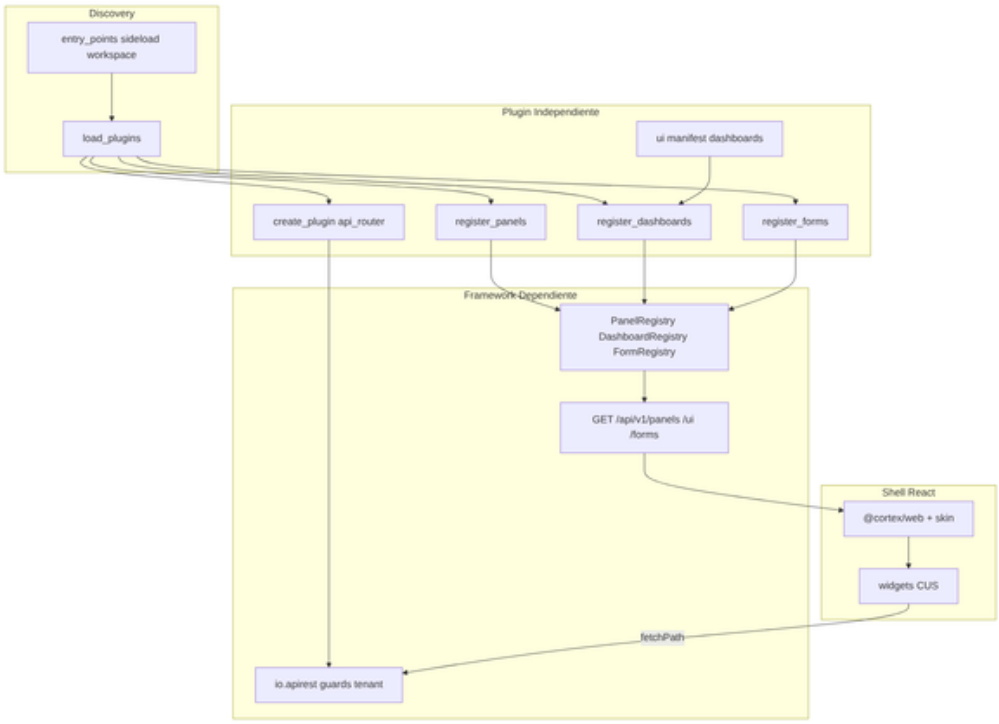

Figura 5.1: Diagrama 5

Leyenda: Secuencia de invocación de hooks en bootstrap. **Actores:** loader, módulo del plugin, registries etiquetados, rutas FastAPI. **Estado:** Implementado.

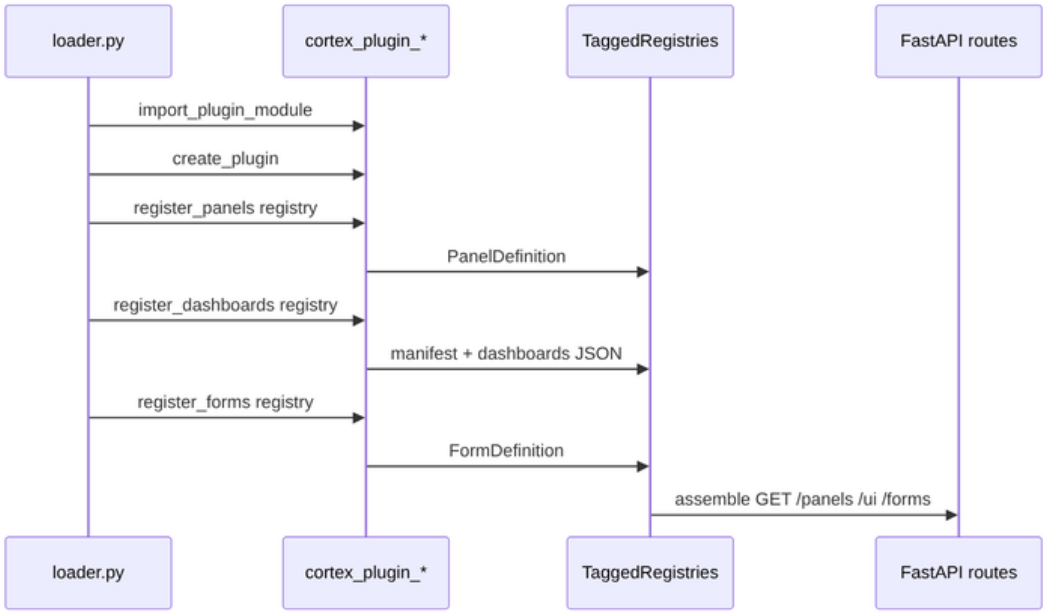

Figura 5.2: Diagrama 6

Capítulo 6

Decisiones de arquitectura (ADRs)

Las siguientes secciones reproducen los Architecture Decision Records aceptados en el repositorio. Cada ADR incluye contexto, decisión, consecuencias y diagramas con su leyenda explicativa.

Capítulo 7

ADR 001: Stack HIVE (core + framework + plugins)

7.1 Estado

Aceptado — 2026-06

7.2 Contexto

Cortex/HIVE es una plataforma multi-tenant ensamblada por plugins Python con frontend React declarativo. El monorepo organiza el producto en tres capas obligatorias: `core/`, `framework/` y `plugins/`.

7.3 Decisión

Componente	Tecnología
API	FastAPI (Python 3.12, Pydantic v2)
Datos	PostgreSQL 16 (una BD por tenant)
Build Python	uv workspace
Frontend	React 18 + Vite (`@cortex/web` + `@cortex/panel-shadcn`, demo `framework/web-shadcn`)
Cache / rate limit	Redis 7

Leyenda: Componentes del stack HIVE (Dependiente). **Actores:** API, BD, cache, frontend. **Estado:** Implementado (infra); Redis NFR parcial (ADR 009).

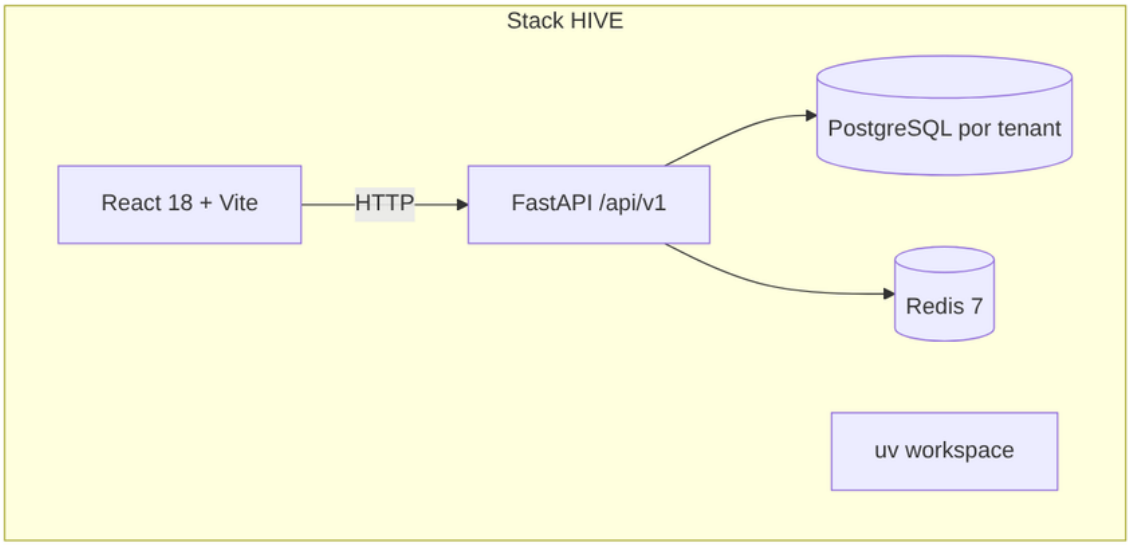

Figura 7.1: Diagrama 7

Leyenda: Dependencias entre capas del monorepo. **Actores:** framework ensambla plugins vía SPI en core. **Estado:** Implementado.

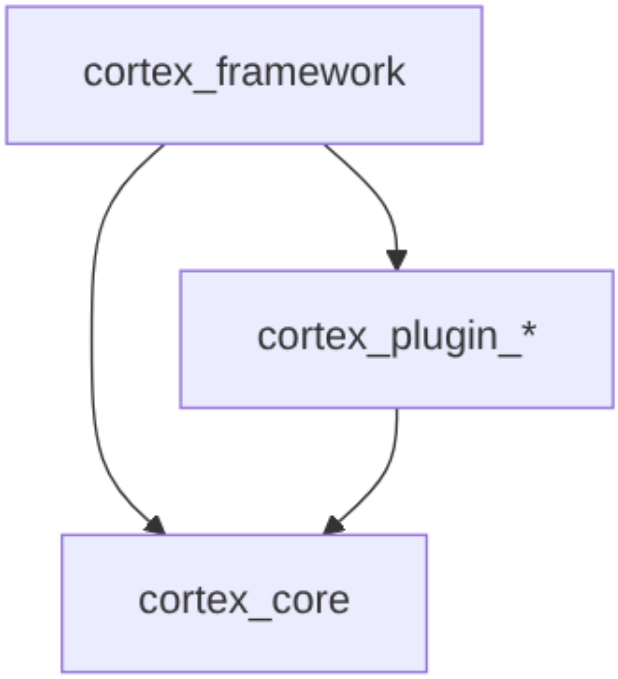

Figura 7.2: Diagrama 8

Leyenda: Regla de dependencia: `core` no importa capas superiores. **Estado:** Implementado.

Estructura obligatoria: `core/`, `framework/`, `plugins/`.

- REST: `/api/v1`, tenant en header `X-Tenant-Id`, OpenAPI automático.
- MCP: fachada por plugin; REST es fuente de verdad.
- Redis 7: caché y rate limiting (valores de producción por configurar).

7.4 Consecuencias

- Plugins activos vía `CORTEX_ENABLED_PLUGINS`.
- `core/` no depende de `framework` ni de `plugins`.
- Frontend en npm workspaces: `@cortex/panel-core`, `@cortex/panel-shadcn` y demo `framework/web-shadcn`.

Capítulo 8

ADR 004: Multi-panel

8.1 Estado

Aceptado — 2026-06

8.2 Contexto

HIVE evoluciona hacia una plataforma multi-tenant con varios **panels** independientes. El prototipo histórico UnoSport Club validó este modelo de panels ortogonales; cada panel tiene su propio path de UI, namespace de API y módulos CUS. El framework no debe hardcodear panels ni mezclar dominios de negocio en un mismo shell.

8.3 Decisión

1. **Panel** como unidad de composición UI/API, ortogonal al **tenant** (`X-Tenant-Id`).
2. Los plugins host declaran panels vía entry point `cortex.panels` y `PanelBuilder` (ver ADR 011). El hook legacy `register_panels()` queda deprecado.
3. El framework expone `GET /api/v1/panels` y ensambla manifests por panel/módulo.
4. El shell React (`framework/web-shadcn`) depende de `@cortex/web` y monta rutas dinámicas desde la API; la skin `@cortex/panel-shadcn` renderiza CUS por panel.

Leyenda: Panels registrados por plugins sobre un tenant. **Actores:** shells/reference/accounting registran panels; tenant en header. **Estado:** Implementado.

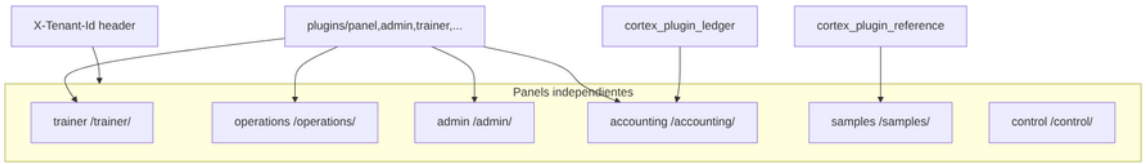

Figura 8.1: Diagrama 9

Leyenda: Mapa de panels Independiente vs tenant. **Estado:** Implementado.

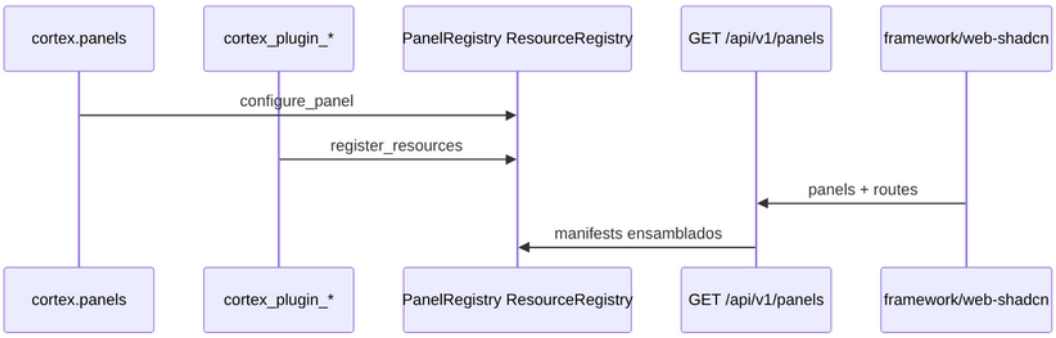

Figura 8.2: Diagrama 10

Leyenda: Registro dinámico de panels sin rutas hardcodeadas en React. **Estado:** Implementado.

8.3.1 Mapping de referencia (panels HIVE)

Panel ID	UI path	API namespace	Plugin
`trainer`	`/trainer/`	`/api/v1/trainer/`	panel host (`plugins/trainer`)
`panel`	`/panel/`	`/api/v1/panel/`	panel host (`plugins/panel`)
`accounting`	`/accounting/`	`/api/v1/accounting/`	panel host + módulo `ledger`
`samples`	`/samples/`	`/api/v1/samples/`	reference (To-Do)
`control`	`/control/`	`/api/v1/control/`	panel host (`plugins/control`)

El módulo **booking** vive como sub-módulo del panel `panel` (no panel propio). Ver ADR 008.

8.3.2 Plugins por defecto

```
CORTEX_ENABLED_PLUGINS=panel,trainer,accounting,control,ledger,reference,clients,booking
```

Ver ADR 011 para `cortex.panels` y `CORTEX_ENABLED_PANELS`.

El plugin monolítico **reserva** fue retirado del monorepo; reservas de negocio → plugin `booking` (ADR 008, pip futuro `cortex-plugin-booking` — ADR 002).

8.4 Consecuencias

- Positivas: panels aislados, extensibles por pip, shell único sin `panelConfig.ts` estático.
- Negativas: más superficie en registry/API; documentación y CI deben listar plugins activos explícitamente.
- Migración: To-Do pasa de `/todo` a panel `samples` en `/samples/todo`; API en `/api/v1/samples/todo`.

Capítulo 9

ADR 003: Cortex Panel (CUS + widgets)

9.1 Estado

Aceptado — 2026-06

9.2 Contexto

HIVE necesita construir apps y paneles admin de forma declarativa, sin una página React por plugin. JSON Forms cubre **formularios**; tablas, layout, cards y acciones requieren otro contrato.

Formily se descartó: añade complejidad y no sustituye vistas REST ni el shell del panel.

9.3 Decisión

Introducir **Cortex Panel** como parte de `cortex_framework`:

Capa	Responsabilidad
JSON Forms	Create/edit: `GET /api/v1/forms/{id}` → widget `form`
CUS (Cortex UI Schema)	Manifests + dashboards JSON servidos por `/api/v1/ui/...`
Widget registry (React)	`@cortex/panel-core` (headless) + skin `@cortex/panel-shadcn`

Leyenda: Módulo UI Dependiente: plugins Independiente declaran JSON; skins React renderizan. **Estado:** Implementado.

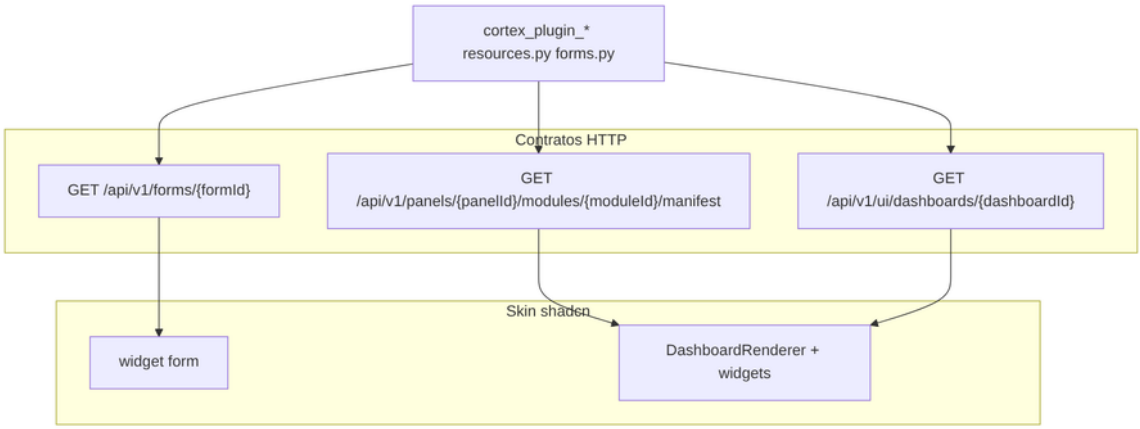

Figura 9.1: Diagrama 11

Leyenda: Flujo manifest/dashboard desde plugin a widgets. **Actores:** plugin, API UI, skin React. **Estado:** Implementado.

Figura 9.2: Diagrama 12

Leyenda: Capas npm del frontend Dependiente. **Estado:** Implementado.

9.3.1 Contratos HTTP (sin cambios de prefijo)

- `GET /api/v1/panels/{panelId}/modules/{moduleId}/manifest` — navegación y pantallas
- `GET /api/v1/ui/dashboards/{dashboardId}` — layout + widgets
- `GET /api/v1/forms/{formId}` — JSON Forms (sin duplicar schema en dashboard)

9.3.2 Widgets MVP

`type`	Uso
`form`	Formulario JSON Forms + submit REST
`data-table`	Tabla sobre `GET config.path`
`api-card`	Card con resumen de `GET config.path`
`section`	Contenedor con título e hijos

Plugins registran UI vía `register_resources()` (recomendado) o `register_dashboards()` legacy (manifest + JSON en `ui/`). El shell React interpreta CUS; **no** importan código de plugins.

9.4 Consecuencias

- Rutas dinámicas desde `GET /api/v1/panels` (shell `@cortex/web` + demo `framework/web-shadcn`).
- Meta-schemas en `cortex_framework/panel/schemas/` para documentación y validación ligera.
- Nuevos widgets MVP en `@cortex/panel-shadcn`; lógica compartida en `@cortex/panel-core`; extensiones vía `registry.register()` en el shell.
- Documentación de desarrollo: `docs/guias/cortex-panel.md`.

9.5 Referencias

- ADR 001 (stack HIVE), ADR 002 (pip)
- `plugins/reference/cortex_plugin_reference/ui/` (panel samples, To-Do)

Capítulo 10

ADR 011: PanelProvider y plugins host

10.1 Estado

Aceptado — 2026-06

10.2 Contexto

ADR 004 introdujo multi-panel con hook `register_panels()` en plugins de negocio. Eso mezclaba identidad del panel (path, auth, branding) con módulos de dominio y obligaba a un plugin monolítico (`shells`) para varios panels.

Cortex necesita el mismo orden de boot: **host primero**, negocio después.

10.3 Decisión

1. **Contratos** en `core` (`PanelConfiguration`, `PanelBrand`, `PanelTheme`) y `framework/cortex_framework/panel` (`PanelBuilder`, `PanelProvider`).
2. **Entry point** `cortex.panels` separado de `cortex.plugins`. Cada plugin host exporta `configure_panel(builder)`.
3. **Orden de carga** en `load_plugins()`:
   - `load_panel_hosts()` → `PanelRegistry` + módulo `home` stub
   - plugins de negocio (`cortex.plugins`) → `register_dashboards` / `register_forms`
   - `validate_panels()`
4. **Cuatro hosts** fijan namespace y shell (portal B2C futuro: host aparte, sin colisión con el módulo `clients`):

Panel	Plugin host	API namespace
Operación	`plugins/panel`	`operations`
Entrenador	`plugins/trainer`	`trainer`
Contabilidad	`plugins/accounting`	`accounting`
Control	`plugins/control`	`control`

Nota: El módulo de negocio **clientes** (`plugins/clients`, API `/api/v1/clients/`) no es un panel host; registra recursos en `operations` / `admin`. Un portal self-service para el cliente final, si se necesita, será un host con id distinto (p. ej. `customer-portal`).

5. **Negocio contable** en `plugins/ledger` (`cortex.plugins`), montado sobre `panel_id=accounting` con API `/api/v1/ledger/`.
6. `register_panels` **en negocio** queda deprecado (warning); solo hosts y legacy `reference`/`samples`.
7. `GET /api/v1/panels/{id}` incluye bloque `configuration` (brand, theme, defaultRoute) para el shell React.

Leyenda: Orden de boot: hosts → plugins → validación. **Actores:** entry points, registries, loader. **Estado:** Implementado.

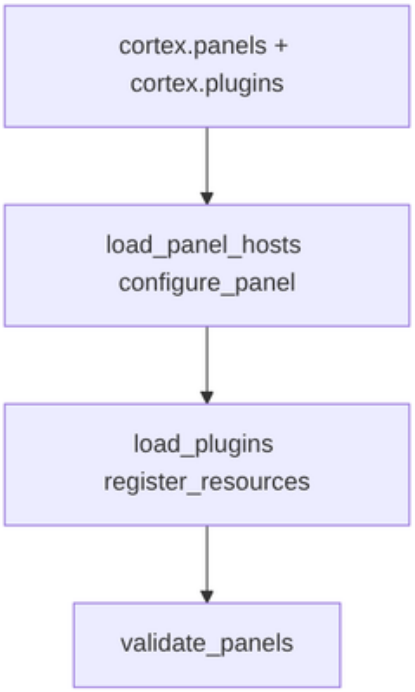

Figura 10.1: Diagrama 13

10.4 Conceptos del panel host

Concepto	Cortex
Id del panel	`builder.id()`
Ruta base UI	`builder.path()`
Autenticación	`builder.auth(scopes=...)`
Marca	`builder.brand()`
Módulos de negocio	plugins vía `register_resources` / `register_dashboards`
Ruta por defecto	`builder.default_route()`

10.5 Consecuencias

- `plugins/shells` eliminado; UI migrada a hosts.
- `CORTEX_ENABLED_PANELS` controla hosts; `CORTEX_ENABLED_PLUGINS` incluye hosts (para workspace) y negocio.
- Frontend (`PanelShell`) lee `configuration` de la API en lugar de valores hardcodeados.

10.6 Referencias

- ADR 004 — multi-panel
- Plan PanelProvider (2026-06)

Capítulo 11

ADR 010: Módulo IO (apirest, webhooks, MCP)

11.1 Estado

Aceptado — 2026-06 · Diseño de ingeniería (implementación parcial: solo `apirest`)

Relacionado con ADR 001, ADR 006 (MCP como submódulo), contexto-proyecto.md (taxonomía Dependiente).

11.2 Contexto

La pizarra de arquitectura clasifica como **Dependiente** todo lo que no es dominio de negocio. La capa **IO** agrupa constructores y middleware de **entrada/salida y transformación de datos**: cómo llegan peticiones (REST, MCP), cómo salen eventos (webhooks) y cómo se exponen capacidades REST a agentes sin duplicar lógica.

Hoy la lógica está dispersa en `cortex_framework/api/`, `tenant/` y diseño MCP en ADR 006 sin paquete unificado.

11.3 Decisión

11.3.1 Ubicación

Módulo interno `cortex_framework.io` — siempre cargado, **no** desactivable vía `CORTEX_ENABLED_PLUGINS`.

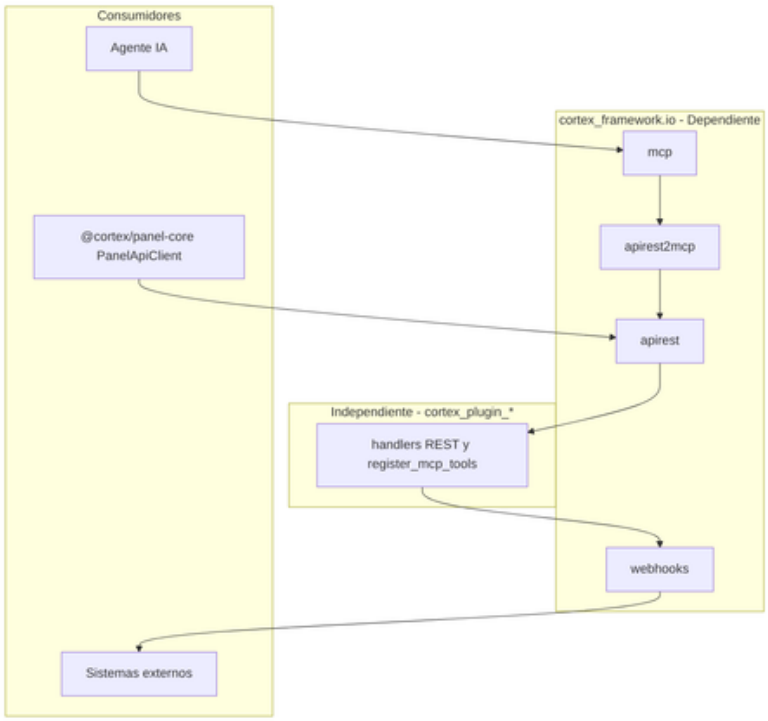

Figura 11.1: Diagrama 14

Leyenda: Capa IO del framework (Dependiente). **Actores:** SPA y agentes entran por REST/MCP; plugins registran routers y tools; webhooks notifican externos. **Estado:** `apirest` parcialmente implementado; webhooks, MCP y apirest2mcp en diseño.

11.3.2 Submódulos

Submódulo	Responsabilidad	Estado
`apirest`	FastAPI `/api/v1`, ensamblado de routers de plugins, guards (`require_plugin_enabled`, tenant, auth)	Parcial — `framework/cortex_framework/api/`
`webhooks`	Registro de suscripciones, firma HMAC, reintentos, entrega de eventos de dominio hacia URLs externas	Diseño
`mcp`	Servidor MCP HTTP, `McpToolRegistry`, hook `register_mcp_tools` (ver ADR 006)	Diseño
`apirest2mcp`	Generación/adaptación de tools MCP desde contratos OpenAPI o handlers REST existentes — **sin** duplicar reglas de negocio	Diseño

11.3.3 Reglas

1. Los plugins **Independiente** exponen negocio vía `api_router()` y opcionalmente `register_mcp_tools`; no implementan servidor HTTP propio.
2. Los handlers MCP invocan la **misma** capa de servicio que los endpoints REST del plugin (regla de oro ADR 006).
3. `webhooks` emite eventos **después** de operaciones exitosas en plugins; la configuración de endpoints vive en framework/tenant, no en cada plugin por separado.
4. `apirest2mcp` es preferible a escribir handlers MCP a mano cuando el contrato REST ya está estable.

11.3.4 Integración con auth y tenant

- `apirest` y `mcp` pasan por ADR 005 (`oauth2-server` / IdP externo opcional) y `TenantMiddleware`.
- Rutas públicas compartidas: health, ready, docs, OpenAPI, discovery OAuth (modo interno).

11.4 Fuera de alcance fase 1

- GraphQL u otros transports.
- Cola de mensajes distinta de Redis para webhooks (ADR 009 evalúa backing store).
- MCP stdio en producción (solo dev local).

11.5 Consecuencias

- Positivas: frontera clara entre plataforma IO y plugins de negocio; un solo lugar para middleware transversal.
- Negativas: refactor de `api/` hacia `io/apirest/` cuando se implemente el paquete nominal.

11.6 Referencias

- ADR 006
- `framework/cortex_framework/api/app.py`
- `framework/cortex_framework/api/routes.py`

Capítulo 12

ADR 005: Autenticación y oauth2-server

12.1 Estado

Aceptado — 2026-06 · Diseño de ingeniería (sin implementación aún)

Relacionado con ADR 004 (`PanelAuthPolicy`), ADR 001 y taxonomía **Dependiente** en contexto-proyecto.md.

12.2 Contexto

El marco técnico exige **OAuth2/OIDC**, **JWT**, **ACL** y permisos por panel. La pizarra de arquitectura ubica esto en el módulo interno `oauth2-server` (equivalente funcional a Keycloak embebido para las aplicaciones Cortex).

Decisión de producto: modo **interno por defecto** (`CORTEX_AUTH_MODE=internal`), con opción de delegar en un **IdP externo** (`CORTEX_AUTH_MODE=external`) para despliegues que ya operan Keycloak, Auth0 u otro.

Hoy `CORTEX_JWT_SECRET` existe en settings pero **no hay middleware ni servidor OAuth** en `cortex_framework`. El panel Control tiene placeholder de ACL sin enforcement.

12.3 Decisión

12.3.1 Módulo `oauth2-server` (Dependiente)

Paquete objetivo: `cortex_framework.oauth2_server` (nombre en código; producto: oauth2-server).

Responsabilidad	Descripción
Emisión de tokens	Authorization Code + PKCE, client credentials (servicio a servicio)
Validación JWT	Access tokens en cada request API y MCP
ACL / permisos	Roles, scopes, mapeo a `PanelAuthPolicy`
Usuarios y clientes	Registro de apps (SPA, agentes, integraciones) — persistencia en PG del tenant o BD de plataforma según fase

12.3.2 Modo interno (`CORTEX_AUTH_MODE=internal`)

- Cortex expone endpoints OAuth2/OIDC estándar (`.well-known/openid-configuration`, authorize, token, JWKS).
- Las demos `framework/web-shadcn` usan PKCE contra el servidor embebido.
- JWT firmados con claves rotables del módulo (no `CORTEX_JWT_SECRET` ad hoc en producción).

12.3.3 Modo externo (`CORTEX_AUTH_MODE=external`)

- Issuer configurable: `CORTEX_OIDC_ISSUER`.
- Audience: `CORTEX_OIDC_AUDIENCE`.
- Validación de firma vía **JWKS** del issuer (cache con TTL).
- Cortex **no** emite tokens; solo valida y autoriza.

12.3.4 Claims mínimos

Claim	Uso
`sub`	Identidad del usuario
`tenant_id` o claim custom mapeado	Tenant de negocio
`scope` / `roles`	Autorización de panels y rutas

12.3.5 Reglas de tenant

- Header `X-Tenant-Id` sigue siendo el contrato HTTP de la plataforma.
- Si **ambos** están presentes (header y claim), deben **coincidir**; si no, `403 TENANT_MISMATCH`.
- Si solo hay claim, el middleware deriva el tenant desde el token.
- Rutas públicas: `/api/v1/health`, `/api/v1/ready`, `/api/docs`, `/api/openapi.json`, discovery OAuth (modo interno).

12.3.6 Middleware y guards

Orden en `create_app`:

1. `TenantMiddleware` (existente)
2. `OidcAuthMiddleware` — valida Bearer (interno o externo según modo), pobla `request.state.user` y tenant efectivo
3. `RateLimitMiddleware` (ADR 009)
4. `PanelScopeGuard` — evalúa `PanelAuthPolicy.scopes` contra claims

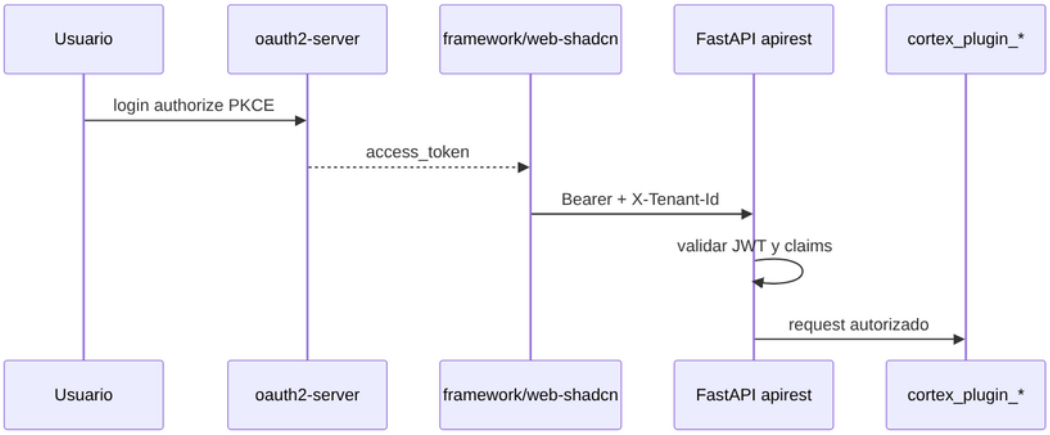

Figura 12.1: Diagrama 15

Leyenda: Flujo de login y API autorizada. **Actores:** usuario, módulo oauth2-server (interno o validación externa), SPA demo, capa apirest, plugin de negocio. **Estado:** Diseño.

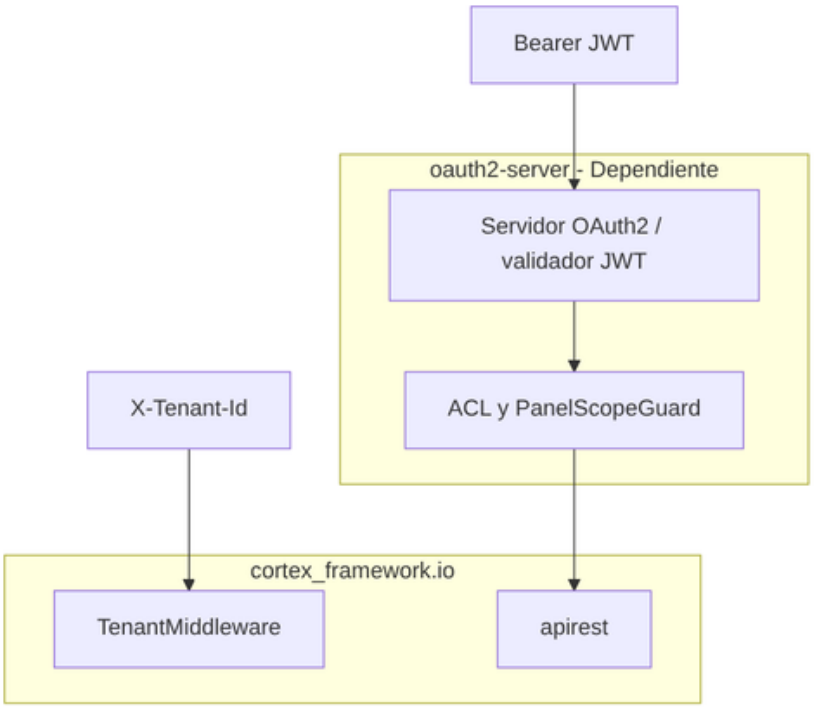

Figura 12.2: Diagrama 16

Leyenda: Módulos internos en la cadena de una petición autenticada. **Actores:** cliente HTTP con JWT y tenant header. **Estado:** Diseño (middleware tenant parcialmente implementado).

12.3.7 Frontend

- `@cortex/panel-core`: `PanelApiClient` acepta `getAccessToken(): Promise<string | null>`.
- `createPanelShellApp` recibe provider de token (cliente OAuth interno o externo).
- El SPA adjunta `Authorization` en cada `fetch`.

12.3.8 Errores estandarizados

Código	HTTP	Cuándo
`UNAUTHORIZED`	401	Token ausente o inválido
`FORBIDDEN`	403	Scope insuficiente o tenant mismatch
`TOKEN_EXPIRED`	401	`exp` vencido

Usar `ApiError` en `cortex_framework/api/errors.py`.

12.4 Fuera de alcance fase 1

- Refresh token en SPA (documentar PKCE puro o BFF con cookie httpOnly).
- RBAC granular completo del panel Control (fase 2).
- Federación social (Google, etc.) — solo OIDC estándar.

12.5 Consecuencias

- Positivas: self-hosted sin Keycloak obligatorio; flexibilidad enterprise con IdP externo; un solo módulo Dependiente para identidad.
- Negativas: el modo interno aumenta superficie de seguridad a mantener; tests requieren mock JWKS o servidor de dev.

12.6 Referencias

- `core/cortex_core/spi.py` — `PanelAuthPolicy`
- `framework/cortex_framework/settings.py`
- `framework/cortex_framework/tenant/middleware.py`
- ADR 010 — capa `apirest` y `mcp`

Capítulo 13

ADR 006: Plugin MCP interno

13.1 Estado

Aceptado — 2026-06 · Diseño de ingeniería (sin implementación aún)

Relacionado con ADR 001 (MCP como fachada IA), ADR 005 (auth en MCP) y ADR 008 (tools de booking).

13.2 Contexto

Los agentes de IA consumen módulos vía **Model Context Protocol (MCP)**. La API REST sigue siendo la **fuente de verdad** del negocio; MCP es la fachada estandarizada hacia agentes.

Patrón acordado: igual que REST/OpenAPI — **infraestructura en framework** (submódulo `io.mcp` — ADR 010), **capacidades en plugins**. Un módulo interno Dependiente expone el servidor MCP; cada plugin registra tools opcionales.

Hoy no existe código MCP en el monorepo.

13.3 Decisión

13.3.1 Módulo interno

- Paquete `cortex_framework.io.mcp` (siempre cargado, no desactivable vía `CORTEX_ENABLED_PLUGINS`). Ver ADR 010.
- Servidor MCP con transporte **HTTP** en ruta dedicada (p. ej. `/api/v1/mcp`).
- Stdio solo para desarrollo local / pruebas con agentes CLI.

13.3.2 SPI en core

Nuevo hook en `core/cortex_core/registrar.py`:

```
class McpToolRegistrar(Protocol):
    def register_tool(
        self,
        name: str,
        description: str,
        input_schema: dict[str, Any],
        handler: Callable[..., Awaitable[Any]],
    ) -> None: ...
```

Hook opcional del plugin: `register_mcp_tools(registry: McpToolRegistrar) -> None`.

Convención de nombres: `{plugin_id}.{action}` — p. ej. `booking.create`, `booking.cancel`.

13.3.3 Regla de oro

Los handlers MCP **no duplican lógica de negocio**. Invocan:

1. Servicios internos del plugin (funciones compartidas con los routers REST), o
2. Cliente HTTP interno hacia `localhost /api/v1/...` del mismo proceso (menos preferido).

OpenAPI del plugin documenta la API humana; MCP documenta la superficie agente.

13.3.4 Descubrimiento dinámico

- Al boot: `McpToolRegistry` ensambla tools de plugins en `CORTEX_ENABLED_PLUGINS`.
- Al togglear plugin en panel Control: registry se actualiza en caliente (misma semántica que plugin guard REST).
- `tools/list` del servidor MCP refleja solo tools de plugins activos para el tenant.

13.3.5 Autenticación MCP

- Mismas reglas que REST: Bearer OIDC + tenant (ADR 005).
- Conexión MCP autenticada; sin acceso anónimo en producción.

Leyenda: MCP como fachada IO hacia agentes; REST y PG son fuente de verdad. **Actores:** agente, servidor MCP, plugins activos. **Estado:** Diseño.

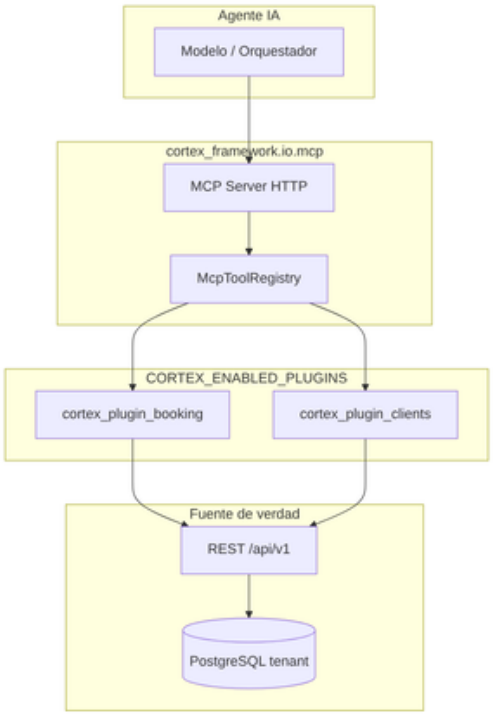

Figura 13.1: Diagrama 17

Leyenda: Flujo tools/list y tools/call sin duplicar lógica REST. **Estado:** Diseño.

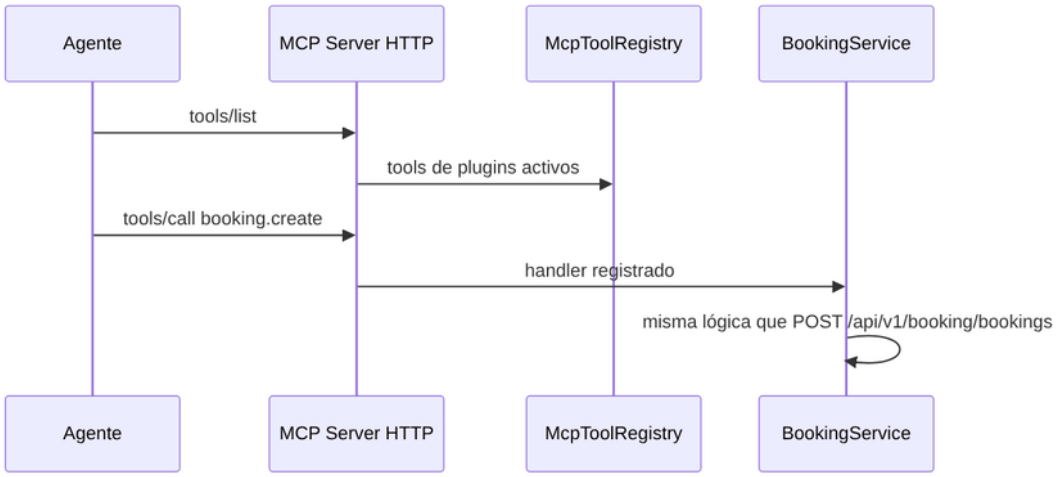

Figura 13.2: Diagrama 18

Leyenda: Ejemplo booking.create invocando el mismo servicio que POST REST. **Estado:** Diseño.

13.3.6 Integración con loader

En `load_plugins()` (tras hooks existentes):

1. Instanciar `McpToolRegistry`.
2. Para cada plugin habilitado: invocar `register_mcp_tools` si existe.
3. Montar router MCP en `create_app`.

13.4 Tools objetivo del plugin booking (detalle en ADR 008)

Tool	Acción REST equivalente
`booking.check_availability`	`GET /api/v1/booking/slots`
`booking.create`	`POST /api/v1/booking/bookings`
`booking.cancel`	`PATCH /api/v1/booking/bookings/{id}/cancel`
`booking.list_agenda`	`GET /api/v1/booking/agenda`

13.5 Consecuencias

- Positivas: desacople agente <-> módulo, tools aparecen al activar plugin, un solo lugar para protocolo MCP.
- Negativas: superficie de seguridad nueva; tests E2E con cliente MCP.
- Requiere ampliar `registrar.py` y loader sin romper plugins existentes (hook opcional).

13.6 Referencias

- `framework/cortex_framework/plugins/loader.py`
- `framework/cortex_framework/control/` — patrón de módulo embebido siempre activo
- ADR 008
- ADR 010

Capítulo 14

ADR 007: Persistencia multi-tenant (PostgreSQL por plugin)

14.1 Estado

Aceptado — 2026-06 · Diseño de ingeniería (sin implementación aún)

Relacionado con ADR 001 (stack HIVE) y ADR 008 (primer plugin con persistencia real).

14.2 Contexto

HIVE declara PostgreSQL 16 con **una base de datos física por tenant** (`cortex_{tenant}`), alineado con aislamiento de datos (Habeas Data / Ley 1581). Hoy:

- `CORTEX_DATABASE_URL_TEMPLATE` y `tenant_database_url()` existen en `framework/cortex_framework/tenant/da`
- Los plugins de dominio usan fixtures en memoria; no hay capa ORM ni migraciones activas.
- `alembic_stub.py` documenta la intención futura sin ejecutar migraciones.

Se requiere un modelo claro: **cada plugin es dueño de su esquema de datos**; el framework solo resuelve la URL por tenant y orquesta el aprovisionamiento.

14.3 Decisión

14.3.1 Aislamiento por tenant

- Una **base de datos física** por tenant: `cortex_{tenant}` (nombre sanitizado desde `X-Tenant-Id`).
- Sin schema compartido entre tenants en la misma instancia PG (salvo BD de sistema/admin para provisioning).

14.3.2 Responsabilidades

Capa	Responsabilidad
framework	`tenant_database_url()`, factory de sesión async, hook de provisioning (crear BD + invocar migraciones de plugins habilitados)
plugin	Modelos SQLAlchemy 2 async, `alembic/`, seeds de dev opcionales
core	Sin dependencia de ORM

14.3.3 Convención de tablas

- Tablas en schema `public` con **prefijo por plugin**: `booking_resources`, `booking_slots`, `booking_bookings`.
- Cada plugin mantiene su propio `alembic.ini` y cadena de revisiones independiente.
- Sin ORM compartido en framework; utilidad mínima: `get_async_session(tenant) -> AsyncSession`.

14.3.4 Provisioning de tenant

1. Alta de tenant (API admin o script ops).
2. `CREATE DATABASE cortex_{tenant}`.
3. Para cada plugin en `CORTEX_ENABLED_PLUGINS`: `alembic upgrade head` contra la URL del tenant.
4. Registro de versión de migración por plugin (tabla `cortex_migrations` en framework o metadata Alembic por plugin).

14.3.5 Runtime

1. `TenantMiddleware` fija el tenant en contexto.
2. Handler del plugin obtiene sesión vía factory con `tenant_database_url()`.
3. Transacciones por request; sin estado de sesión global.

Leyenda: Provisioning de BD por tenant y sesión en runtime (módulo Dependiente tenant). **Estado:** Diseño.

Figura 14.1: Diagrama 19

Leyenda: Flujo alta tenant → migraciones por plugin. **Estado:** Diseño.

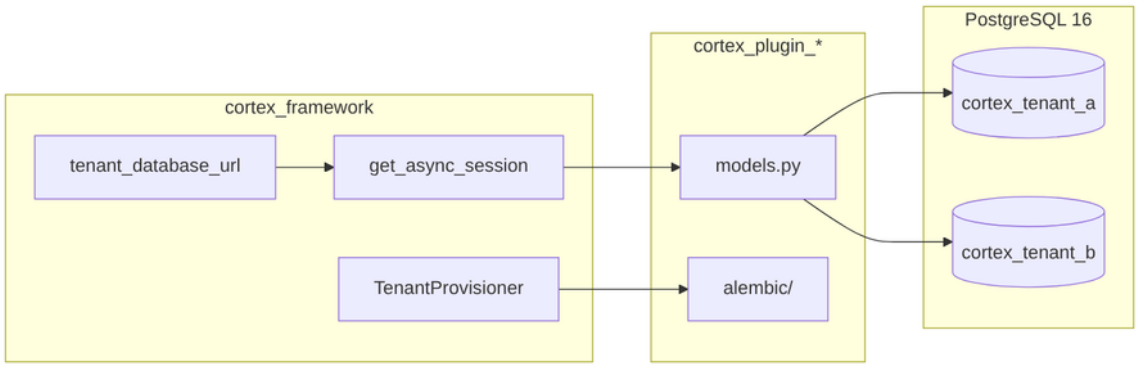

Figura 14.2: Diagrama 20

Leyenda: Framework resuelve URL; cada plugin Independiente posee modelos y Alembic. **Estado:** Diseño.

14.3.6 Migración desde fixtures

- **Desarrollo**: seed opcional desde JSON actual (`fixtures/`) tras migraciones.
- **Producción**: sin `_STORE` en memoria; datos solo en PG.

14.4 Variables de entorno

Variable	Rol
`CORTEX_DATABASE_URL_TEMPLATE`	`postgresql+asyncpg://user:pass@host:5432/`
`CORTEX_DEFAULT_TENANT`	Tenant para `/ready` y dev local

14.5 Consecuencias

- Positivas: aislamiento legal por tenant, plugins desacoplados en esquema, Alembic independiente por dominio.
- Negativas: N bases de datos que administrar; provisioning debe ser idempotente y observable.
- El plugin **booking** será el primero en adoptar este ADR (ver ADR 008).

14.6 Referencias

- `framework/cortex_framework/tenant/database.py`
- `framework/cortex_framework/tenant/alembic_stub.py`
- ADR 009 (`/ready` comprueba PG)

Capítulo 15

ADR 008: Módulo booking (dominio genérico)

15.1 Estado

Aceptado — 2026-06 · Diseño de ingeniería (sin implementación aún)

Relacionado con ADR 007, ADR 006 y ADR 004 (módulo en panel `panel`).

15.2 Contexto

El plugin `booking` existe como **scaffold**: fixtures en memoria, solo endpoints `GET`, conceptos residuales del prototipo histórico (`court`, `cancha`). **No** se replica el dominio del prototipo UnoSport Club.

El diseño objetivo es un modelo **genérico multi-recurso**: cualquier cosa reservable (espacio, persona, equipo) sin acoplar a deporte.

15.3 Decisión

15.3.1 Modelo de dominio

Entidad	Rol
Resource	Recurso reservable. `type`: `space`, `person`, `equipment`, … Atributos flexibles en JSON.
Slot	Ventana temporal sobre un recurso: `starts_at`, `ends_at`, `status` (`open`, `held`, `booked`).
Booking	Reserva confirmada: referencia a slot(s), `client_ref`, `status`, metadata.
AvailabilityQuery	VO de consulta (rango de fechas, filtros por `resource_type`).

Leyenda: Modelo ERD del plugin Independiente booking. **Estado:** Diseño.

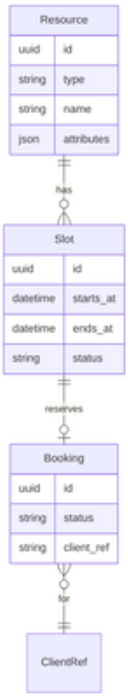

Figura 15.1: Diagrama 21

15.3.2 Tablas PostgreSQL (prefijo `booking_`)

Tabla	Campos clave
`booking_resources`	`id`, `type`, `name`, `attributes` (JSONB), `created_at`
`booking_slots`	`id`, `resource_id`, `starts_at`, `ends_at`, `status`
`booking_bookings`	`id`, `slot_id`, `client_ref`, `status`, `metadata` (JSONB), `idempotency_key`

Migraciones en `plugins/booking/alembic/`. Ver ADR 007.

15.3.3 API REST (`/api/v1/booking/`)

Método	Ruta	Uso
GET	`/resources`	Listar recursos (`?type=`)
GET	`/slots`	Disponibilidad en rango (`?from=&to=&resource_id=`)
POST	`/bookings`	Crear reserva; header `Idempotency-Key`
GET	`/bookings`	Listar (`?status=&from=`)
GET	`/bookings/{id}`	Detalle
PATCH	`/bookings/{id}/cancel`	Cancelar
GET	`/agenda`	Vista agenda agregada (solo lectura)

Errores vía `ApiError`:

Código	HTTP
`SLOT_UNAVAILABLE`	409
`BOOKING_NOT_FOUND`	404

Código	HTTP
`INVALID_RANGE`	422
`IDEMPOTENCY_REPLAY`	200 (misma respuesta cacheada)

Leyenda: Flujo UI → API → PG para consulta y alta de reserva. **Estado:** Diseño.

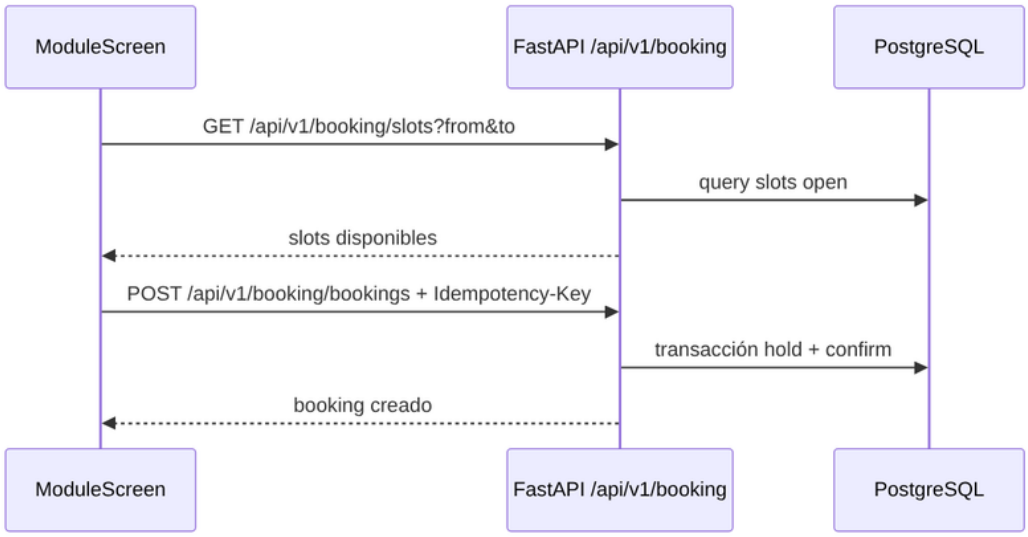

Figura 15.2: Diagrama 22

Leyenda: Secuencia de creación de reserva con idempotencia (ADR 009). **Estado:** Diseño.

15.3.4 UI CUS

- Panel `panel`, módulo `booking` en `plugins/booking/cortex_plugin_booking/ui/`.
- Renombrar en manifests/dashboards: `court` → `resource`; pantallas: Inicio, Agenda, Reservas, Recursos.
- Widgets: `api-table` para listados, `json-form` para alta de reserva; **sin páginas React nuevas**.

15.3.5 Tools MCP (ADR 006)

Tool	Descripción
`booking.check_availability`	Equivalente a `GET /slots`
`booking.create`	Equivalente a `POST /bookings`
`booking.cancel`	Equivalente a `PATCH /bookings/{id}/cancel`
`booking.list_agenda`	Equivalente a `GET /agenda`

15.3.6 Integración con otros plugins (futuro)

Plugin	Contrato
`clients`	`client_ref` resuelve contra `GET /api/v1/clients/{id}`
`payments` / `pricing`	Evento `booking.confirmed` (hook/documentar; sin implementar en fase 1)

Leyenda: Plugin Independiente booking dentro del panel operativo; MCP reutiliza REST. **Estado:** Diseño.

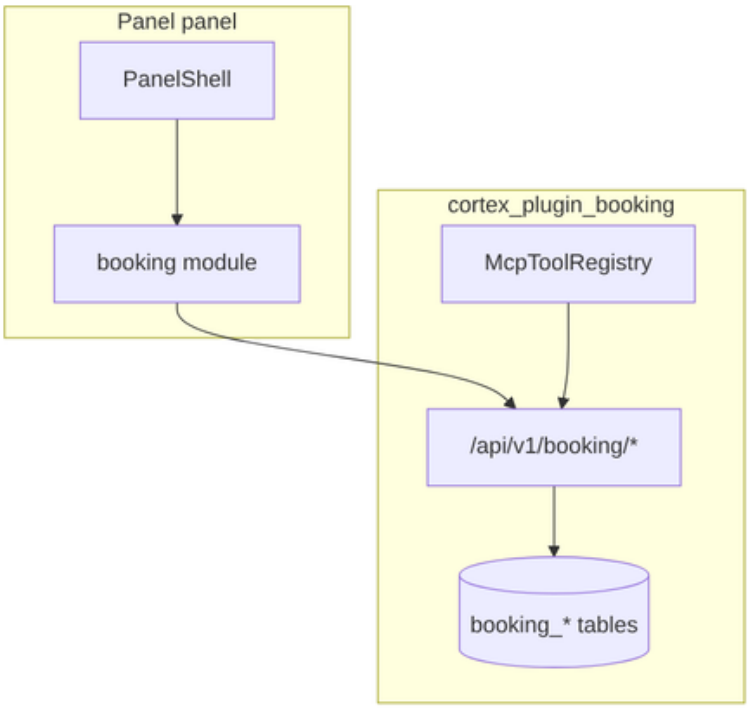

Figura 15.3: Diagrama 23

Leyenda: Integración panel CUS, API booking y persistencia. **Estado:** Diseño (scaffold sin PG).

15.3.7 Migración desde scaffold actual

1. Implementar modelos + Alembic (ADR 007).
2. Sustituir `_STORE` por repositorios async.
3. Seed dev desde `fixtures/` opcional.
4. Actualizar JSON CUS con paths y columnas del modelo genérico.

15.4 Consecuencias

- Positivas: dominio portable, desacoplado del prototipo, listo para MCP y multi-tenant real.
- Negativas: breaking change respecto a fixtures actuales; tests deben migrar a PG (testcontainers o BD efímera).

15.5 Referencias

- `plugins/booking/cortex_plugin_booking/` — scaffold actual
- ADR 007
- ADR 006
- cortex-panel.md — widgets CUS

Capítulo 16

ADR 009: Redis y requerimientos no funcionales

16.1 Estado

Aceptado — 2026-06 · Diseño de ingeniería (sin implementación aún)

Relacionado con ADR 001, ADR 007 y ADR 008 (idempotencia en POST).

16.2 Contexto

El marco técnico adopta prácticas NFR desde el desarrollo: rate limiting configurable por tenant, caché, health checks profundos e idempotencia en escrituras. Los umbrales numéricos (latencia, concurrencia, cuotas) se calibran en producción.

Hoy:

- Redis 7 está en `docker-compose.yml` y `CORTEX_REDIS_URL` en settings.
- `RateLimitMiddleware` es un **stub** que deja pasar todas las peticiones.
- `/ready` devuelve `{"status": "ready"}` sin comprobar dependencias.
- No hay cliente Redis en código Python ni patrón de idempotencia.

16.3 Decisión

16.3.1 Rate limiting

- Implementar en `RateLimitMiddleware` con Redis: `INCR cortex:rl:{tenant}:{window}` + TTL 60s.
- Límite por defecto: `CORTEX_RATE_LIMIT_PER_MINUTE` (120); override por tenant en fase 2 (config en PG o Redis hash).
- Respuesta `429` con cuerpo `ApiError` (`RATE_LIMIT_EXCEEDED`) y header `Retry-After`.
- Excluir: `/health`, `/ready`, `/api/docs`, `/api/openapi.json`.

16.3.2 Caché (fase 2 — documentar, no bloquear fase 1)

- GET idempotentes de lectura: manifests UI, listados estáticos de recursos.
- Clave: `cortex:cache:{tenant}:{path_hash}`; TTL corto (30–120s).
- Invalidación en writes del mismo recurso.

16.3.3 Health checks

Ruta	Comportamiento
`GET /api/v1/health`	Liviano; proceso vivo
`GET /api/v1/ready`	Comprueba Redis `PING` + PG `SELECT 1` en tenant default

Respuesta `/ready` si falla dependencia: `503` con detalle por servicio.

16.3.4 Idempotencia

- Header `Idempotency-Key` en `POST /api/v1/booking/bookings` (y escrituras críticas futuras).
- Redis: `cortex:idempotency:{tenant}:{key}` → respuesta serializada, TTL 24h.
- Replay: misma respuesta HTTP sin re-ejecutar lógica.

Leyenda: Rate limit y health checks con Redis y PG (módulo Dependiente). **Estado:** Diseño (stub actual).

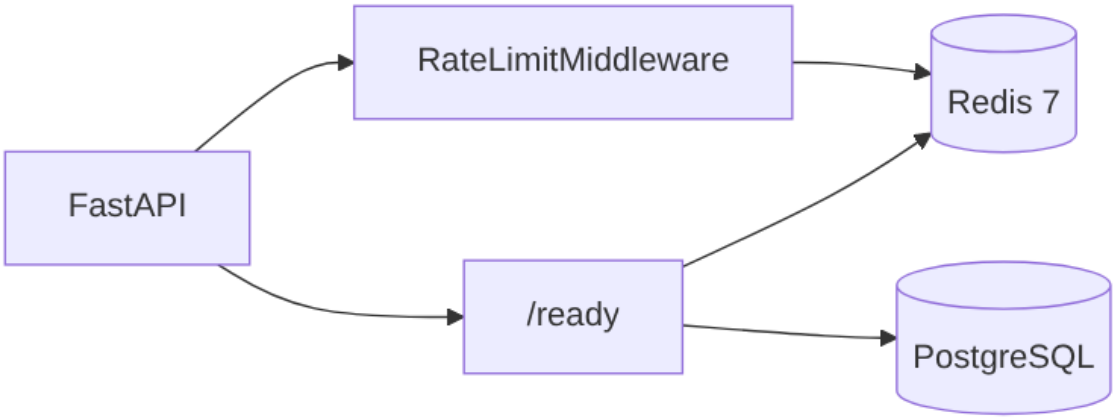

Figura 16.1: Diagrama 24

Leyenda: Middleware y `/ready` dependen de Redis y PostgreSQL. **Estado:** Diseño.

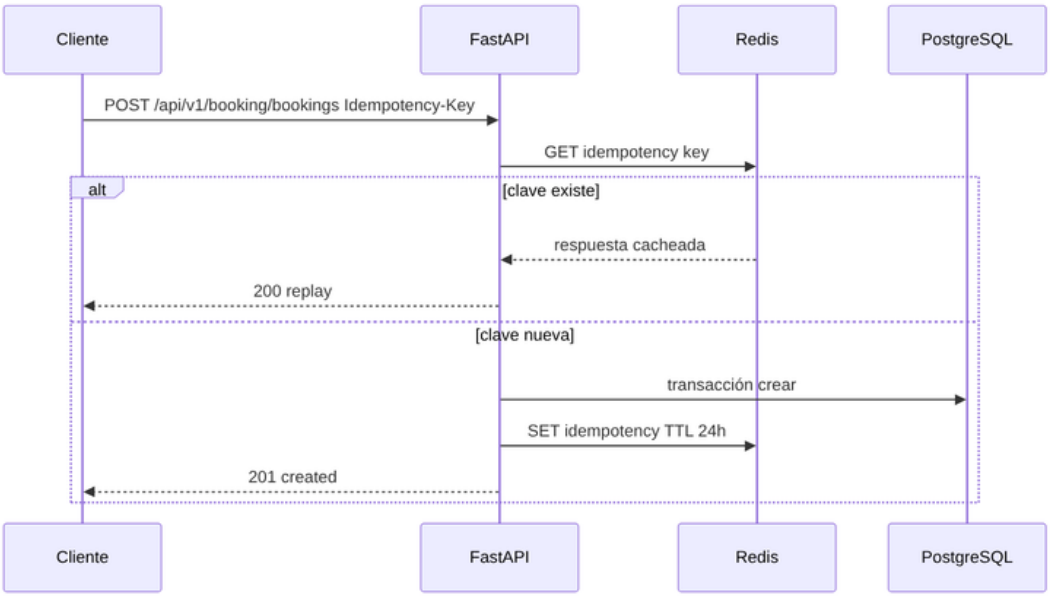

Figura 16.2: Diagrama 25

Leyenda: Idempotencia en POST booking vía Redis. **Estado:** Diseño.

16.3.5 Escalabilidad y stateless

- API sin sesión server-side; estado en PG y Redis.
- Réplicas horizontales detrás de Coolify/nginx sin sticky sessions.
- Pool de conexiones async SQLAlchemy por proceso (ADR 007).

16.3.6 Fuera de alcance documentado

- Umbrales numéricos de latencia/concurrencia (producción).
- Reintentos idempotentes en cliente (patrón recomendado en OpenAPI, no código framework).
- Rollback de despliegue (procedimiento ops Coolify).

16.4 Variables de entorno

Variable	Default	Rol
`CORTEX_REDIS_URL`	`redis://localhost:6379/0`	Rate limit, idempotency, caché
`CORTEX_RATE_LIMIT_PER_MINUTE`	`120`	Cuota por tenant por minuto

16.5 Consecuencias

- Positivas: NFRs medibles, alineado con correo técnico, base para calibrar en producción.
- Negativas: Redis pasa a dependencia dura para rate limit; `/ready` más lento pero útil para orquestadores.

16.6 Referencias

- `framework/cortex_framework/tenant/rate_limit.py` — stub actual
- `framework/cortex_framework/api/routes.py` — `/health`, `/ready`
- ADR 008 — `Idempotency-Key`

Capítulo 17

ADR 018: Routing de pantallas y jerarquía de títulos del panel

17.1 Estado

Aceptado — 2026-07

Relacionado con ADR 003 y ADR 012.

17.2 Contexto

El shell shadcn mostraba títulos duplicados (header del panel, `ModuleScreen`, `DashboardRenderer`, cards de widgets) y rutas ambiguas: una URL como `/panel/resources/create` podía resolverse contra `path: ":id"` con `id=create`, mostrando un infolist vacío en lugar de un formulario.

Los manifests manuales y el orden de `screens[]` no bastaban para garantizar el comportamiento correcto.

17.3 Decisión

17.3.1 1. Resolución de pantallas por especificidad

En `@cortex/panel-core`:

- `scoreScreenPath(path)` — prioriza segmentos literales sobre parámetros (`:id`).
- `matchModuleScreen(manifest, relativePath)` — elige la pantalla coincidente con mayor score.
- `relativeModulePath(pathname, moduleUrl)` — obtiene el segmento relativo bajo un módulo.

`ModuleScreen` y `PanelShell` deben usar estos helpers; no reimplementar matching por orden de array.

17.3.2 2. Jerarquía de títulos

Capa	Regla
`PanelShell`	Un `h1` por pantalla = `screens[].title` resuelto
`ModuleScreen`	`h2` solo si `theme.shellOwnsPageTitle === false`
`DashboardRenderer`	`h3` solo si `layout.hideTitle === false`
Widgets	`CardTitle` solo si `config.title` está definido

17.3.3 3. Flags de tema (`PanelTheme`)

Flag	Default en shadcn	Descripción
`shellOwnsPageTitle`	`true`	El shell muestra el título de pantalla
`showPanelSubtitle`	`true`	Muestra `panel.title` como subtítulo del header

17.3.4 4. Defaults en `ResourceBuilder`

Los dashboards generados para list/create/edit/view usan `emit_dashboard(..., hide_title=True)`.

Si `TableBuilder` no define título, se propaga el título del recurso al widget `data-table`.

El manifest emitido ordena pantallas: list → create → edit → view.

17.4 Consecuencias

- Manifests manuales con acción CREATE deben declarar `path: "create"` explícitamente.
- Plugins que migran a `ResourceBuilder` obtienen routing y títulos correctos sin cambios en React.
- Skins distintos de shadcn pueden dejar `shellOwnsPageTitle` en `false` para comportamiento legacy.
- Tests unitarios en `routingParams.test.ts` y `test_ui_form_builder.py` fijan el contrato.

17.5 Referencias

- Cortex Panel — runtime
- Recursos (plugins)
- `packages/cortex-panel-core/src/routingParams.ts`
- `packages/cortex-panel-shadcn/src/shell/PanelShell.tsx`

Capítulo 18

ADR 019: Panel admin (parametrización)

18.1 Estado

Aceptado — 2026-06

18.2 Contexto

La operación diaria (agenda, reservas, clientes, recursos) debe convivir con la parametrización de negocio (ventas, facturación, pagos, tarifas, ajustes por tenant) sin mezclar ambos perfiles en un mismo menú.

Cada panel host monta su **propio dashboard** de entrada. Los plugins de negocio pueden contribuir módulos, widgets y settings en **uno o varios paneles** según su criterio.

18.3 Decisión

1. Panel host `admin` (`/admin/`, namespace `admin`) vía entry point `cortex.panels`.
2. Módulos `sales`, `billing`, `payments`, `pricing` con `panel_id="admin"` (CRUD y navegación de back-office).
3. Los mismos plugins inyectan **widgets** en el dashboard de varios paneles con `register_widgets` (p. ej. KPI de pagos en `admin` y resumen en `panel`).
4. `merge_panel_infrastructure` **fusiona** widgets de plugins en el dashboard del host, sin reemplazarlo.
5. `register_resource` y `register_resource_for_panels` permiten el mismo recurso en varios paneles.
6. Ajustes por plugin con `register_settings(..., panel_id="admin")` y widget `settings` con sidebar de secciones.
7. El flujo de nueva reserva en `booking` orquesta widgets composables (`reservation-flow`).

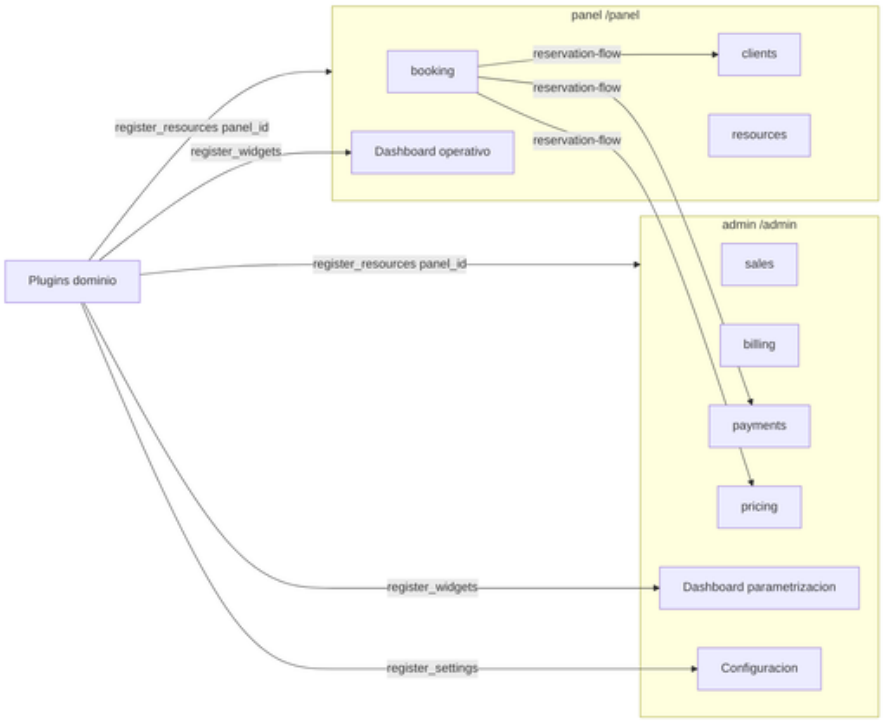

Figura 18.1: Diagrama 26

18.4 Consecuencias

- `CORTEX_ENABLED_PANELS` incluye `panel,admin` en el perfil de demo.
- El panel operativo no lista módulos de finanzas; puede mostrar widgets de esos dominios en su dashboard.
- `default_route` de `admin` apunta al dashboard (`/admin`); el operativo puede priorizar una pantalla de trabajo (`/panel/booking/agenda`).
- Ver panel hosts y settings.