# TaskStream

## Documentación técnica

Alternativa a Jira para equipos de desarrollo: proyectos, tickets, QA, releases y seguimiento en tiempo real.

API NestJS · GraphQL · CQRS · Event sourcing · Angular · NgRx Signals · @pi/ui

**Versión 1.0** · 3 de octubre de 2026

Generado a partir de docs/ del repositorio task-stream

# Índice


## 1. Visión general

- Qué resuelve
- Stack
- Estructura del monorepo

## 2. Arquitectura

- 2.1 Capas hexagonales
- 2.2 Flujo completo de una escritura
- 2.3 El cliente web
- 2.4 Diagrama de despliegue

## 3. API

- 3.1 Modelo de dominio (contexto tracker)
- 3.2 GraphQL
- 3.3 REST con OpenAPI (para otras apps)

## 4. Cliente web

- 4.1 Librerías
- 4.2 Estado: NgRx Signals con eventos
- 4.3 GraphQL
- 4.4 @pi/ui
- 4.5 Diseño
- 4.6 Rutas

## 5. Autenticación (Pireon)

- 5.1 Cómo valida la API un token
- 5.2 Iniciar sesión desde el navegador
- 5.3 Webhooks de Pireon (opcional)
- 5.4 Usuarios en TaskStream

## 6. Desarrollo

- 6.1 Requisitos
- 6.2 Puesta en marcha
- 6.3 Comandos
- 6.4 Cómo añadir una funcionalidad
- 6.5 Problemas conocidos

## 7. Despliegue

- 7.1 Docker
- 7.2 Variables de entorno de la API
- 7.3 Escalado
- 7.4 Antes de producción
- 7.5 Reconstruir el read model

## 8. Decisiones de diseño

## 9. Servidor MCP

- 9.1 Cómo funciona
- 9.2 Herramientas

9.3 Conectar un cliente
9.4 Desarrollo
9.5 Despliegue

10. Lo que queda

10.1 Para dar el producto por terminado
10.2 Fallos y deuda conocidos
10.3 Siguientes funcionalidades
10.4 Descartado por ahora
10.5 Hecho desde la primera versión de esta lista

Anexo A. Librerías

# 1. Visión general

## Qué resuelve

| Necesidad | Cómo se resuelve |
|---|---|
| 🐛 Bugs | Tickets de tipo `BUG` con prioridad, área y etiquetas |
| 🌍 Públicos | Un proyecto público se puede leer sin iniciar sesión (web, GraphQL y REST): tablero, tickets, comentarios, releases, actividad y estadísticas; «Explorar» los lista |
| 🗑️ Borrar | El propietario borra un proyecto confirmando su clave; tickets, comentarios, releases y enlaces desde otros proyectos se borran en cascada (sagas) y la clave queda libre |
| 🧩 Configuración | Cada proyecto tiene sus estados (con categoría, color y motivo obligatorio), transiciones, tipos de ticket y áreas; lo usado se archiva, no se borra |
| 🎨 Cambios de UI | Tipo `UI_CHANGE` con enlace al frame de Figma (validado contra `figma.com`) |
| ⚙️ Features | Tipo `FEATURE` siguiendo el workflow completo |
| 🧪 QA | Por defecto, `READY_FOR_QA → IN_QA → DONE / QA_FAILED`; fallar QA exige un motivo |
| 🚀 Releases | Versiones a las que se asignan tickets; no se publican con tickets sin terminar (salvo forzando) |
| 📝 Notas de versión | Borrador en Markdown con los tickets terminados por tipo, editable y copiable, también tras publicar |
| 👥 Asignaciones | Solo a miembros del proyecto; panel de carga por persona |
| 💬 Comentarios | Decisiones y explicaciones dentro del ticket, con formato (Markdown), capturas pegadas, respuestas, reacciones (👍 🎉 ...) y `@menciones`; el autor edita, el admin modera |
| 🔎 Vistas | El tablero se filtra por texto, persona (o sin asignar), tipo, prioridad, área y release; los filtros se guardan como vistas propias o compartidas con el proyecto |
| 📋 Lista | Todos los tickets en filas: ordenar por cualquier columna, filtrar, cambiar estado, prioridad o persona sin abrirlos, editar en bloque y duplicar |
| 🚦 WIP y carriles | Límite de trabajo en curso por estado (la columna se marca al pasarse) y tablero agrupable por persona, épica, prioridad o tipo |
| 🎲 Estimación | Puntos por ticket (escala 0-1-2-3-5-8-13-21) en la ficha, las tarjetas, las columnas y el backlog; velocidad por sprint y aviso si un sprint planificado supera la media |
| 🏃 Sprints | Iteraciones con fechas y objetivo, una en curso por proyecto, con burndown (tickets o puntos) y traspaso de lo pendiente al cerrar |
| 📅 Calendario | Los sprints mes a mes como franjas, con las releases y los vencimientos de cada día; se planifica un sprint pulsando su primer día y todo se descarga en .ics |
| 📋 Backlog | Lo pendiente de empezar, en el orden en que se va a coger; se reordena arrastrando y el tablero sigue ese orden |
| 🗑️ Papelera | Borrar un ticket lo manda a la papelera (quien lo abrió o un admin); se recupera durante 30 días y luego se borra del todo |

| Necesidad | Cómo se resuelve |
|---|---|
| ☑️ En bloque | Selecciona varios tickets del backlog y cámbiales a la vez estado, persona, sprint, prioridad o etiqueta |
| ⌨️ Atajos y búsqueda | Ctrl/⌘ K o / busca tickets y proyectos de todos tus proyectos; `g b`, `g k`, `g s`, `g r` navegan, `c` crea, `e` edita, `?` los lista |
| 🗺️ Hoja de ruta | Las épicas en el tiempo (fecha de inicio y límite), con los sprints y las releases como referencias |
| 🗳️ Propuestas y votos | Un proyecto público puede abrir un formulario para que cualquiera, sin cuenta, reporte un error o proponga una idea; quien tenga sesión vota lo que quiere |
| 🌐 Web del colectivo | La página de cada proyecto en pireon.dev enseña su hoja de ruta, versiones, sprint y tickets para recién llegados ( `good-first-issue` ) |
| 🧩 Desglose | Épicas → tickets → subtareas (tres niveles), con su progreso; y una checklist dentro de cada ticket para los pasos pequeños |
| 👀 Seguir | Cualquier miembro sigue los tickets que le interesan y los ve juntos en «Que sigo»; quien lo abre, lo tiene asignado o comenta, lo sigue solo |
| 🔔 Avisos | Campana con lo que te concierne (asignaciones, cambios de estado, QA fallido, comentarios, respuestas), en vivo y opcionalmente por correo |
| ⏰ Recordatorios | Aviso (y correo, si se quiere) el día antes, el mismo día y al vencer la fecha límite, a quien lo tiene y a quien lo sigue; las tarjetas marcan lo que vence |
| 📎 Adjuntos | Capturas, logs y vídeos en el ticket o bajo un comentario; arrastrar, elegir o pegar con Ctrl+V; miniaturas de las imágenes |
| 🔗 Relaciones | `BLOCKS`, `DEPENDS_ON`, `TESTS`, `DUPLICATES`, `RELATES_TO`, también entre proyectos (FE ↔ BE ↔ QA); lo que espera a otro ticket abierto sale marcado como «Bloqueado» |
| 🌿 Git | Webhook para GitHub, Gitea o GitLab: commits, ramas y pull requests que nombran un ticket (WEB-12) aparecen en su ficha |
| 📄 Plantillas | Cada tipo de ticket puede traer una descripción inicial y una checklist que se aplican al crearlo |
| 🤖 Automatizaciones | Reglas «cuando… si… entonces…»: al crear, mover, asignar, comentar o fusionar una PR, mover, asignar, etiquetar, priorizar, comentar o meter en el sprint |
| 📡 Webhooks | Un POST firmado (HMAC-SHA256) a otro sistema cuando pasa algo: tickets, comentarios, releases, sprints; con prueba y último envío |
| 📦 Importar/exportar | Todos los tickets a CSV; importar un CSV propio o una exportación de Jira, con vista previa de lo que se va a crear |
| 🏠 Inicio | «Mi trabajo»: mis tickets en todos los proyectos, por categoría (QA fallidos primero), lo que espera QA, próximas releases e «Ir a…» por clave |
| 📊 Seguimiento | Panel con contadores, flujo por estado, el equipo con sus tickets abiertos (enlaza a su tablero), feed de actividad e historial por ticket |
| 📈 Estadísticas | Creados vs. resueltos, flujo acumulado, burnup de releases, tiempos de ciclo y entrega, tasa de fallo en QA y resueltos por persona |

| Necesidad | Cómo se resuelve |
|---|---|
| ⚡ Tiempo real | Tablero, comentarios, actividad, releases y membresías se actualizan solos (GraphQL subscriptions) |

Los **usuarios no se gestionan aquí**: vienen del servidor de identidad **Pireon** (Swagger). TaskStream gestiona proyectos, membresías y roles por proyecto (`OWNER`, `ADMIN`, `MEMBER`, `VIEWER`).

## Stack

| Pieza | Tecnología |
|---|---|
| Monorepo | Nx 23 |
| API | NestJS 11, `@nestjs/cqrs`, GraphQL (Apollo Server 5, code-first) con subscriptions `graphql-ws` |
| Escritura | PostgreSQL 17: *event store* + tablas de lookup (TypeORM) |
| Escritura → lectura | Redis 7: Redis Stream (eventos) y pub/sub (tiempo real); también caché de sesiones |
| Lectura | MongoDB 8: vistas desnormalizadas (Mongoose) |
| Identidad | Pireon (OAuth2 / Laravel Passport) |
| Cliente | Angular 22 (standalone, zoneless, Signal Forms), NgRx Signals (plugin de eventos), Apollo Angular, `@pi/ui` |
| Tests | Jest (API), Vitest (web), e2e de la API contra Docker |
| Despliegue | Docker: imagen Node de la API y nginx para la web |

## Estructura del monorepo

```
apps/
  api/          composición de la API (NestJS): config, GraphQL, conexiones, e2e
  web/          composición del cliente (Angular): rutas, providers, estilos, nginx
  mcp/          servidor MCP para asistentes de IA (NestJS), consume la API REST
libs/
  api/{shared,identity,tracker}/{domain,application,infrastructure,presenters}
  web/{shared,identity,tracker}/{domain,application,infrastructure,presenters}   (+
web/shared/ui)
docs/           esta documentación
codegen.ts      tipos GraphQL del cliente generados desde apps/api/schema.gql
Dockerfile      tres destinos: api, web y mcp
docker-compose.yml
```

API y web comparten la misma estructura: **contexto** (`shared`, `identity`, `tracker`) × **capa** (`domain`, `application`, `infrastructure`, `presenters`). Los imports siguen el mismo patrón: `@task-stream/api/tracker/domain`, `@task-stream/web/tracker/application`...

## Contextos

- **shared**: piezas comunes. En la API, event store, Redis, `VersionedAggregateRoot`, errores de dominio. En la web, el cliente GraphQL y la integración de `@pi/ui`.
- **identity**: quién es el usuario (Pireon), sesiones, inicio de sesión y autorización global.
- **tracker**: el producto (proyectos, tickets, comentarios, releases, actividad).

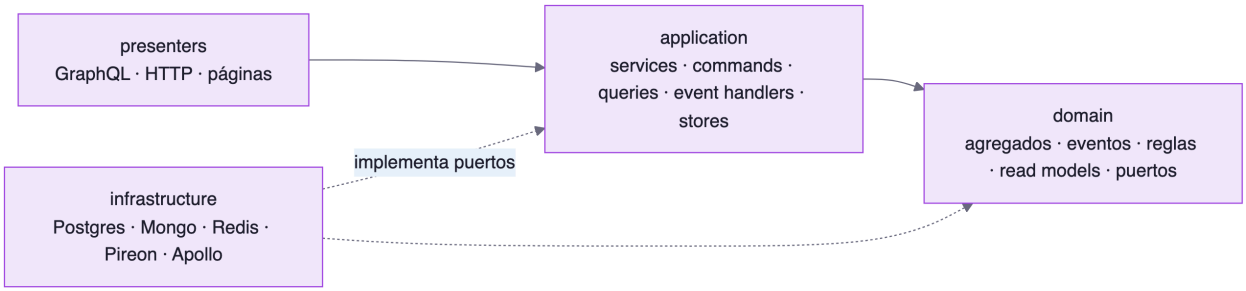
# 2. Arquitectura

TaskStream combina tres ideas:

1. **Arquitectura hexagonal** (puertos y adaptadores), aplicada igual en la API y en la web.
2. **CQRS**: las escrituras (commands) y las lecturas (queries) van por caminos y bases de datos distintas.
3. **Event sourcing** en la API: lo que se guarda no es el estado de los agregados sino sus eventos.

## 2.1 Capas hexagonales

Cada contexto se divide en cuatro capas, cada una en su propia librería Nx:




| Capa | Contiene | Puede depender de |
|---|---|---|
| `domain` | Modelo y reglas de negocio; puertos que necesita el dominio | `domain` |
| `application` | Casos de uso: services, commands/queries + handlers, proyecciones (API) o facades + stores (web), puertos de salida | `domain`, `application` |
| `infrastructure` | Adaptadores **de salida** que implementan los puertos (bases de datos, Redis, Pireon, GraphQL) | `domain`, `application`, `infrastructure` |
| `presenters` | Adaptadores **de entrada** (resolvers GraphQL, controllers HTTP, guards, páginas Angular) | `domain`, `application`, `presenters` |

Las reglas no son una convención: las comprueba `@nx/enforce-module-boundaries` en `eslint.config.mjs` (`npx nx run-many -t lint`).

- `domain` solo puede importar `@nestjs/common` (por `@Injectable()` de las factories) y `@nestjs/cqrs` (por `AggregateRoot`). Nada de TypeORM, Mongoose, Redis, GraphQL, Angular, NgRx ni Apollo.
- `application` no puede importar adaptadores (TypeORM, Mongoose, Redis, Apollo...).
- `presenters` nunca importa `infrastructure` ni usa los buses CQRS: habla con los `*.service.ts` (API) o con las facades (web).
- `infrastructure` nunca importa `presenters`.
- Los contextos también están acotados: `tracker` puede usar `identity` y `shared`; `shared` no depende de nadie; la web no depende de la API (y viceversa).

### Composición

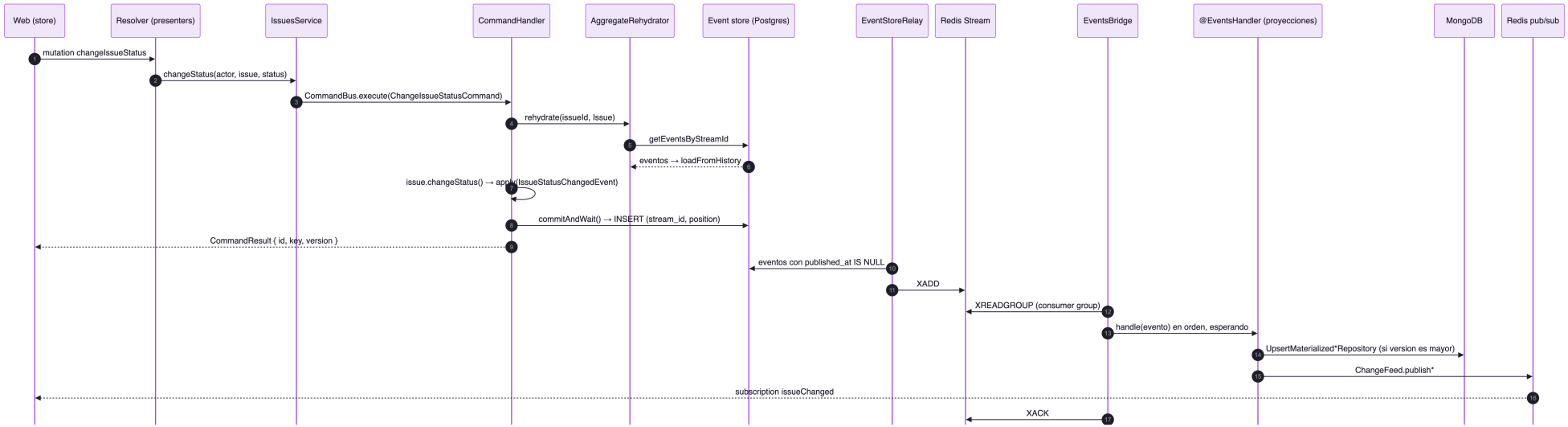
Las apps son la *composition root*: montan cada contexto de fuera hacia dentro.

```
// apps/api/src/app/app.module.ts
SharedModule.withInfrastructure(SharedInfrastructureModule.forRootAsync({ … }));
TrackerPresentersModule.withApplication(
  TrackerApplicationModule.withInfrastructure(TrackerInfrastructureModule),
);
```

```
// apps/web/src/app/app.config.ts
provideTrackerInfrastructure();                        // puertos → adaptadores GraphQL
{ provide: Notifications, useExisting: ToastService } // puerto de la aplicación → toasts de
@pi/ui
```

## 2.2 Flujo completo de una escritura

# Flujo completo de una escritura



## Escritura: event sourcing en PostgreSQL

- Los agregados (`Project`, `Issue`, `Comment`, `Release`) extienden `VersionedAggregateRoot`, que extiende el `AggregateRoot` de `@nestjs/cqrs` y añade la versión.
- **Crear**: una factory `@Injectable()` construye el agregado y aplica el evento de creación (`skipHandler`).
- **Modificar**: el `AggregateRehydrator` carga los eventos del stream y los reproduce (`loadFromHistory`); los métodos del agregado validan y hacen `apply(new XEvent(…))`; el estado solo cambia en los handlers `onXEvent`.
- **Guardar**: el `EventStorePublisher` sustituye al publisher del `EventBus`, así que `commit()` escribe en la tabla `events`. `commitAndWait()` espera a que la escritura termine (el `commit()` de Nest no devuelve la promesa).
- **Concurrencia optimista**: `UNIQUE(stream_id, position)`. Si dos peticiones modifican el mismo agregado a la vez, la segunda falla con `ConcurrencyError` (código GraphQL `CONFLICT`).
- **Lookups**: lo que no se puede comprobar dentro de un agregado vive en tablas aparte: claves de proyecto únicas (`project_keys`), numeración de tickets (`issue_sequences`) y nombres de release únicos (`release_names`).

## De escritura a lectura: Redis

- El `EventStoreRelay` copia los eventos nuevos (`published_at IS NULL`) al Redis Stream `task-stream:events`, en orden y con `FOR UPDATE SKIP LOCKED` (varias instancias no duplican). La entrega es *at-least-once*.
- El `EventsBridge` consume el stream con un *consumer group*, deserializa cada evento (`EventClsRegistry` + `@AutowiredEvent`) y lo entrega a los `@EventsHandler` registrados para su clase. A diferencia de `eventBus.subject$.next()`, **espera** a cada handler y procesa los eventos en orden; solo hace `XACK` cuando todos terminan. Si uno falla, el evento queda pendiente y se reintenta (y `XAUTOCLAIM` recupera los de instancias caídas).
- Después de los handlers, el bridge ejecuta también las **sagas** (ver abajo).

## Lectura: MongoDB

- Las proyecciones son `@EventsHandler` de la capa `application/event-handlers`, uno por evento (más el de actividad, que escucha todos). Escriben a través de los puertos `UpsertMaterialized*Repository`.
- Cada documento guarda la `version` del último evento aplicado; el upsert solo se aplica si la versión es mayor, así que reentregar un evento no tiene efecto.
- Las queries leen por los puertos `Find*Repository`. Los read models (`ProjectReadModel`, `IssueReadModel`...) están en `domain/read-models`.

## Tiempo real

Tras actualizar Mongo, el handler publica en Redis pub/sub (`ChangeFeed`). Las subscriptions GraphQL escuchan esos canales (uno por proyecto, ticket o usuario), así que funcionan con varias instancias de la API. Como se publica **después** de proyectar, quien recibe el aviso siempre lee el estado nuevo.

## Sagas: reacciones entre agregados

Cuando un evento de un agregado debe cambiar otros agregados, lo hace una saga (`application/sagas`), con el `@Saga()` estándar de `@nestjs/cqrs`: un observable que filtra eventos (`ofType`) y emite comandos. Como el `EventBus` en memoria nunca ve los eventos guardados (su publisher es el event store), las sagas las ejecuta el **`EventsBridge`**: a cada evento, tras las proyecciones, pasa ese evento solo (`of(event)`) por cada saga y ejecuta los comandos que emite, uno a uno y esperando. Así heredan las garantías del bridge: orden, reintento y nada se pierde si la API se cae a mitad.

| Saga | Evento | Comandos |
|---|---|---|
| `ProjectDeletionSaga` | `ProjectDeletedEvent` | `DeleteIssueCommand` por cada ticket, `DeleteReleaseCommand` por cada release y `DeleteSprintCommand` por cada sprint del proyecto |
| | `IssueDeletedEvent` | `PurgeCommentCommand` por cada comentario, `DropLinksToIssueCommand` en los tickets que lo enlazaban y `DetachFromParentCommand` en los que colgaban de él |
| `ReleaseArchivedSaga` | `ReleaseArchivedEvent` | `DetachIssueFromReleaseCommand` para sus tickets sin terminar (los terminados se quedan) |
| `SprintCarryOverSaga` | `SprintClosedEvent` / `SprintDeletedEvent` | `CarryOverIssueCommand`: lo pendiente pasa al sprint elegido o al backlog; al borrar, todos vuelven al backlog |
| `CommentersFollowSaga` | `CommentAddedEvent` | `WatchIssueCommand`: quien comenta pasa a seguir el ticket |
| `PeopleFollowSaga` | `IssueCreatedEvent`, `IssueAssignedEvent` | `WatchIssueCommand`: quien abre el ticket y quien lo tiene asignado pasan a seguirlo |
| `UserDeactivatedSaga` | `UserDeactivatedEvent` (contexto identity) | `UnassignIssueCommand` para sus tickets abiertos en todos los proyectos, como actor `system` («TaskStream») |

Reglas que cumplen:

- **A quién afecta sale del event store**, no del read model (`TrackerStreamsRepository`, `ReleaseIssuesRepository`), salvo la saga de usuarios, que busca tickets asignados ahora mismo en Mongo.
- **Comandos idempotentes**: un evento puede procesarse dos veces, así que cada comando comprueba si aún hay algo que hacer (`WriteAccess.live()` devuelve `null` si el agregado no existe o ya está borrado; desasignar solo si sigue asignado a esa persona...). Son comandos internos: ningún presenter los expone.
- **Nada bloquea el stream**: si el dominio rechaza un comando (no encontrado, regla de negocio...), el bridge lo registra y sigue; los conflictos de concurrencia se reintentan hasta 5 veces y, si persisten, el evento queda pendiente.
- **Autoría**: los cambios se atribuyen a quien provocó el evento (quien borró el proyecto o archivó la release).
- **Entre contextos**: el webhook de Pireon guarda `UserDeactivatedEvent` en el event store (`EventRecorder`, stream `identity-user-<id>`), y la saga del tracker reacciona a él sin que los

contextos se llamen entre sí.

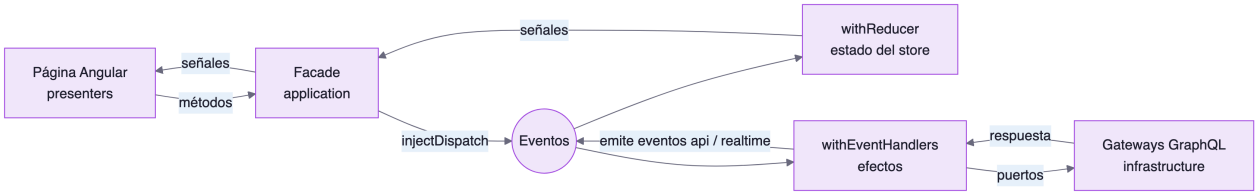
### Consistencia eventual

Las mutations devuelven `CommandResult { id, key, version }` en cuanto los eventos están guardados; Mongo se actualiza milisegundos después. Consecuencias:

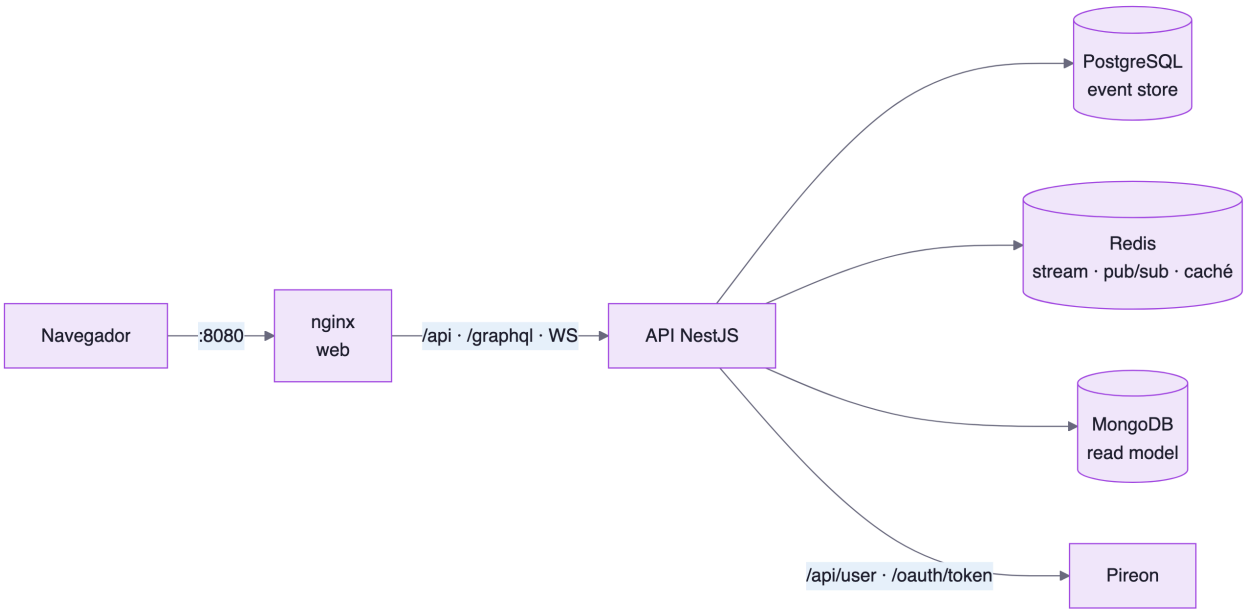
- El cliente reintenta las queries que devuelven *not found* justo después de crear algo.
- Si necesitas el dato actualizado, espera un evento de subscription con `version ≥ result.version`.
- Las reglas que necesitan datos exactos **no** leen Mongo: por ejemplo, "no publicar una release con tickets sin terminar" reconstruye esos tickets desde el event store (`ReleaseIssuesRepository`).

## 2.3 El cliente web

El cliente replica las capas (ver Cliente web):



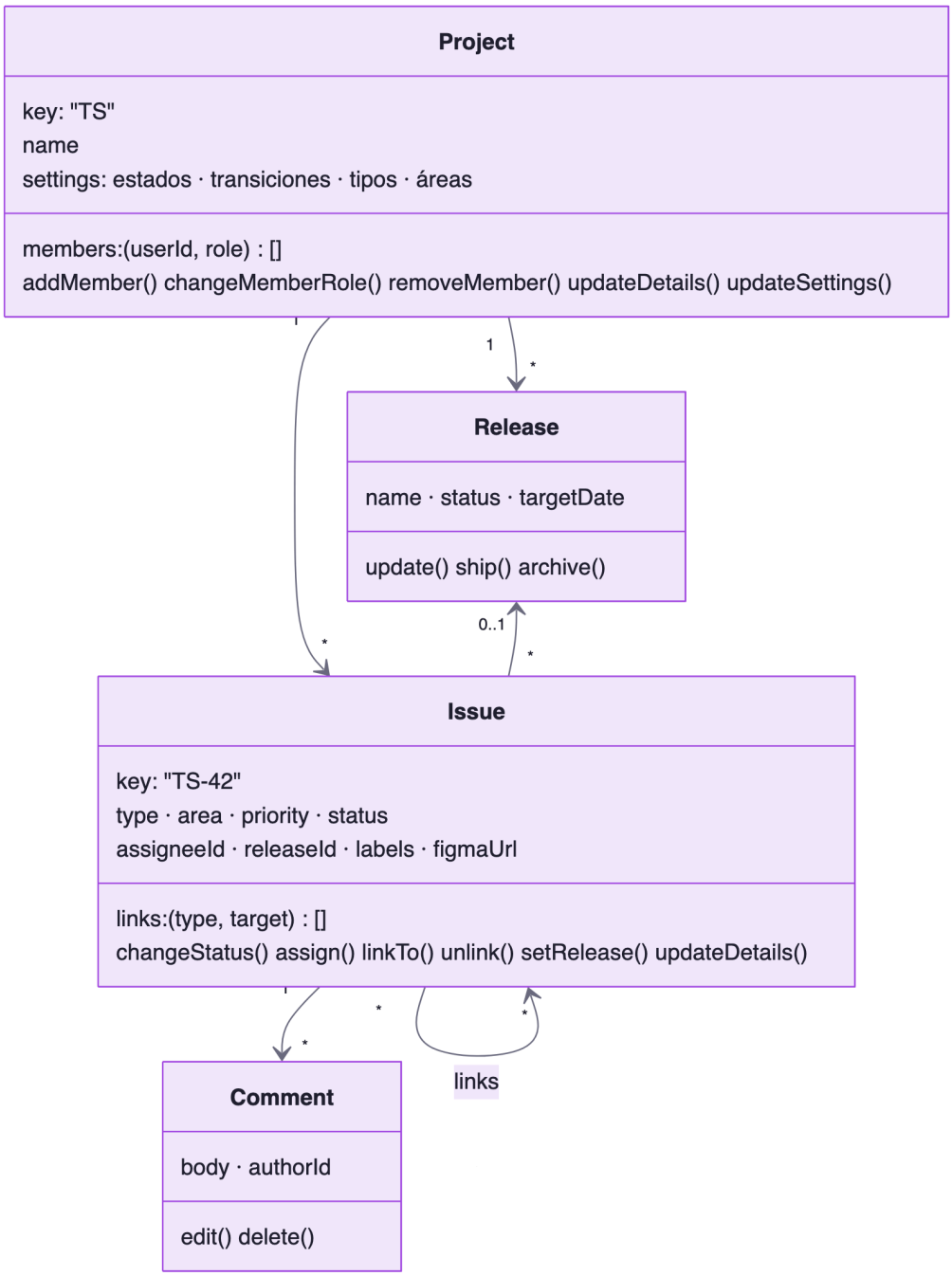

## 2.4 Diagrama de despliegue



# 3. API

API GraphQL en `/graphql` (HTTP para queries y mutations, WebSocket `graphql-ws` para subscriptions) más unos pocos endpoints REST bajo `/api`. El esquema completo se genera en `apps/api/schema.gql`.

## 3.1 Modelo de dominio (contexto `tracker`)




### Roles por proyecto

| Rol | Ver | Crear/modificar tickets, comentar | Gestionar miembros y releases |
|---|---|---|---|
| `OWNER` (quien crea el proyecto) | ✅ | ✅ | ✅ (no se le puede quitar ni cambiar el rol) |

| Rol | Ver | Crear/modificar tickets, comentar | Gestionar miembros y releases |
|---|---|---|---|
| `ADMIN` | ✅ | ✅ | ✅ |
| `MEMBER` | ✅ | ✅ | ❌ |
| `VIEWER` | ✅ | ❌ | ❌ |

Cualquiera puede salir de un proyecto (salvo el propietario). Los no miembros reciben *not found*, para no revelar que el proyecto existe.

## Proyectos públicos

Con `isPublic` (al crear, o después con `updateProject` / `PATCH /api/projects/{project}`, propietario o admin), **cualquiera puede leer** el proyecto, también sin token: el proyecto y sus miembros, tablero, tickets, comentarios, historial, releases (progreso y burnup), actividad, seguimiento y estadísticas. Escribir sigue siendo cosa de los miembros según su rol (sin token: 401; con token pero sin ser miembro: 403). Un usuario con sesión que no es miembro lee igual que un anónimo. Las subscriptions exigen sesión; con ella, también se pueden seguir proyectos públicos.

**Propuestas del público**: un proyecto público puede aceptar errores e ideas de cualquiera, sin cuenta (`acceptsProposals`, en `updateProject` / `PATCH`). `POST /api/explore/projects/{project}/proposals` (`kind`: `BUG` o `IDEA`, `title`, `description`, `author` opcional) crea un ticket con la etiqueta `propuesta`, a nombre de TaskStream, en el primer estado, de tipo Bug o Feature (o el primer tipo activo). No se pide correo, porque quedaría a la vista. Contra el spam, el campo `website` es una trampa (si llega relleno se responde como si nada y no se guarda) y cada dirección envía como mucho 5 cada 15 minutos (429); el contador vive en Redis (`RateLimitStore`), así que vale igual con varias instancias. Los propietarios y admins del proyecto reciben un aviso (`PROPOSED`). La web tiene el formulario en `/p/{CLAVE}/proponer`, enlazado en la cabecera del proyecto.

**Votos**: quien tenga sesión vota los tickets de un proyecto público, y los miembros los de uno privado (`voteIssue(issueId, on)`, `PUT/DELETE /api/issues/{issue}/vote`). Cada ticket da `votes` y `voted` (si he votado yo; en REST, solo con token); la lista de quién votó no se publica.

`publicProjects` (GraphQL) y `GET /api/explore/projects` (REST, sin token) los listan. Al volver a privado, quien no es miembro vuelve a recibir *not found*. Los enlaces a tickets de otros proyectos solo se muestran si el visitante puede leer esos proyectos.

## Borrar un proyecto

Solo el **propietario**, escribiendo la clave del proyecto para confirmar (`deleteProject(projectId, confirmKey)` o `DELETE /api/projects/{project}?confirm=CLAVE`). El proyecto desaparece al momento y, a continuación, una saga borra sus tickets (y sus comentarios), sus releases y los enlaces que tickets de otros proyectos tenían hacia los suyos (ver Arquitectura). La clave queda libre para un proyecto nuevo. Los eventos se quedan en el event store como auditoría, pero desde la API no se puede deshacer.

## Configuración por proyecto

Cada proyecto tiene sus **estados**, **transiciones**, **tipos de ticket** y **áreas** (`Project.settings`). Todos empiezan con los valores por defecto de abajo, y el propietario o un admin los cambia con

`updateProjectSettings` (o `PUT /api/projects/{project}/settings`); el cambio es un evento más (`ProjectSettingsChangedEvent`).

| Valor | Campos | Por defecto |
|---|---|---|
| Estado | `id`, `name`, `category`, `color`, `requiresReason`, `archived` (el orden es el del tablero) | `BACKLOG`, `TODO`, `IN_PROGRESS`, `IN_REVIEW`, `READY_FOR_QA`, `IN_QA`, `QA_FAILED`, `DONE`, `CANCELLED` |
| Tipo | `id`, `name`, `icon`, `color`, `archived` | `BUG`, `FEATURE`, `UI_CHANGE`, `TASK`, `QA` |
| Área | `id`, `name`, `archived` | `FRONTEND`, `BACKEND`, `MOBILE`, `QA`, `DESIGN`, `DEVOPS` |

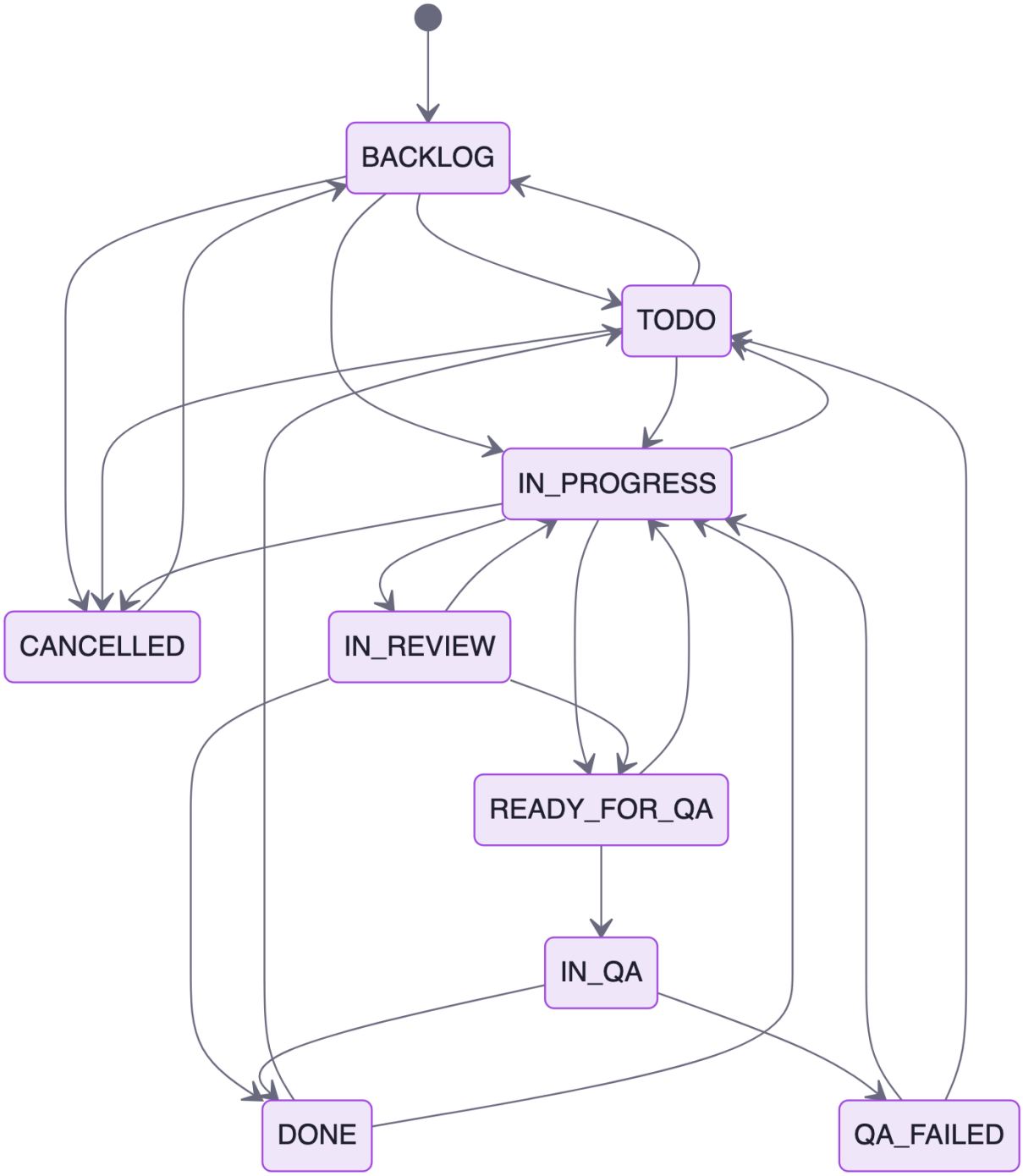
La **categoría** de un estado dice qué significa, se llame como se llame: `TODO` (pendiente), `IN_PROGRESS`, `QA`, `DONE` y `CANCELLED`. Releases («todo terminado» = `DONE` o `CANCELLED`), «Mi trabajo», el seguimiento y las estadísticas trabajan con categorías, así que siguen funcionando con estados propios. Los tickets guardan el id del estado y, en el read model, su categoría (`Issue.statusCategory`).

Reglas:

- **No se borra lo que ya existe: se archiva.** Tickets, historial y estadísticas siguen apuntando a ese id. Un valor archivado no se ofrece para tickets nuevos ni recibe transiciones; los tickets que ya lo tienen lo conservan.
- Los ids son estables (`A-Z`, `0-9`, `_`, hasta 30, empezando por letra); los nombres, de 1 a 40 caracteres, se pueden cambiar cuando se quiera.
- Hace falta al menos un estado activo de categoría `TODO` (los tickets nuevos entran en el primero), uno `DONE` y un tipo activo.
- Las transiciones solo pueden apuntar a estados que existen y nunca de un estado a sí mismo.
- Colores: `slate`, `stone`, `violet`, `blue`, `sky`, `teal`, `green`, `amber`, `orange`, `red`, `pink`. Iconos: `bug`, `feature`, `palette`, `task`, `qa`, `flag`, `star-fill`, `rocket`, `document`, `light-bulb`, `key`, `support`.

Los proyectos creados antes de que existiera la configuración usan los valores por defecto (sus eventos no la llevan). Al arrancar, la API rellena la categoría de los tickets del read model que no la tienen.

## Workflow por defecto




- `QA_FAILED` y `CANCELLED` exigen un **motivo** (`requiresReason`), que queda en el historial.
- No se puede pasar a `DONE` saltándose QA salvo desde `IN_REVIEW` (cambios que no requieren QA).
- `Issue.allowedTransitions` devuelve los estados activos a los que puede pasar el ticket según el workflow de su proyecto.

## Otras reglas

- **Proyecto**: clave de 2 a 10 caracteres alfanuméricos que empieza por letra y es única (los tickets se numeran `CLAVE-1`, `CLAVE-2`…).
- **Asignación**: solo a miembros del proyecto. Para añadir miembros, el usuario debe existir y estar activo en Pireon.
- **Relaciones**: nunca consigo mismo ni duplicadas; pueden cruzar proyectos si el actor puede ver el destino. Se muestran en los dos tickets (`OUTWARD` en el origen, `INWARD` en el destino).
- **Releases**: solo las `PLANNED` aceptan o pierden tickets; publicar con tickets sin terminar exige `force: true`. Las notas de versión (`notes`, Markdown) se escriben también después de publicar (no una vez archivada); `notesDraft` / `GET /api/releases/{id}/notes/draft` propone un borrador con los tickets hechos agrupados por tipo (los de QA quedan fuera).
- **Comentarios**: solo el autor edita; borra el autor o un admin del proyecto. Se puede responder a un comentario (`replyTo`), con un solo nivel de respuestas: responder a una respuesta la suma al mismo hilo, y la respuesta tiene que ser del mismo ticket. Quien escribió el comentario recibe un aviso (`REPLIED`). Un comentario borrado que tiene respuestas se queda, sin texto (`deleted: true`), para que el hilo se siga entendiendo, y desaparece con su última respuesta; no cuenta en `commentCount`.
- **Formato**: descripciones, comentarios y notas de versión se escriben en Markdown; la API los guarda tal cual y la web los muestra limpiando el HTML (DOMPurify). Una imagen pegada se sube como adjunto y queda en el texto como `![nombre](attachment:<id>)`: la subida responde `attachmentId`, y la web la descarga con la sesión de quien lee, así que sirve también en proyectos privados.
- **Reacciones**: 👍 👎 😄 🎉 😕 ❤️ 🚀 👀 en los comentarios (`reactToComment(commentId, emoji, on)`, `POST /api/comments/{id}/reactions`); los miembros, también los lectores. Cada comentario da `reactions` con quién eligió cada una. No generan actividad ni avisos.
- **Bloqueos**: un ticket espera a otro si ese otro lo bloquea (`BLOCKS`) o si él depende de ese otro (`DEPENDS_ON`), mientras el otro no esté hecho ni cancelado. `blockedBy` lo calcula al leer (los enlaces no guardan el estado del otro ticket), con una sola consulta por tablero o página.
- **Duplicar**: `duplicateIssue` / `POST /api/issues/{issue}/duplicate` copia lo que dice el ticket (tipo, título con «(copia)», descripción, área, prioridad, etiquetas, Figma, estimación, fechas, épica y pasos de la checklist sin marcar), no dónde está: la copia empieza en el primer estado, sin asignar, sin sprint, release, comentarios ni adjuntos.
- **Cambios de UI**: `figmaUrl` debe ser un enlace `https://…figma.com/…`.
- **Adjuntos**: los suben los miembros (no los lectores), hasta `ATTACHMENTS_MAX_MB` (25 MB) por archivo y 50 por ticket; pueden ir en el ticket o bajo uno de sus comentarios (`commentId`). Quita uno quien lo subió o un admin. Los bytes se guardan primero y el evento después, así que nunca queda un adjunto apuntando a un archivo que no existe; si el evento no se guarda, el archivo se borra. Al borrar un ticket se borran sus archivos.
- **Épicas y subtareas**: un ticket puede colgar de otro del mismo proyecto (`parent`), hasta tres niveles (épica → ticket → subtarea) y sin bucles; se comprueba sobre el event store, no sobre el read model. Una «épica» es cualquier ticket de primer nivel del que cuelgan otros (`epics(projectId)` da su progreso: terminados = hechos o cancelados, y puntos). Si se borra el padre, sus hijos vuelven al primer nivel.
- **Orden del backlog**: cada ticket tiene un `rank` (texto que se compara carácter a carácter). Mover uno (`rankIssue`, `PUT /api/issues/{issue}/rank`) solo reescribe ese ticket: recibe un rango entre

el de quien queda encima y el de quien queda debajo, leídos del event store; si entretanto el orden cambió, la API lo rechaza y la web recarga. Un ticket que nunca se movió ocupa su orden de creación (`IssueRankedEvent` no existe para él). Las columnas del tablero y `issues(filter: { sort: RANK })` siguen este orden.

- **Integración con Git**: un owner o admin añade en la configuración del proyecto sus repositorios (`api`, `web`, `admin`…), cada uno con su propio secreto. Todos llaman a la misma URL, `POST /api/webhooks/git/{proyecto}`, con los eventos *push* y *pull/merge request*; GitHub y Gitea firman el cuerpo con el secreto (HMAC-SHA256) y GitLab lo envía como `X-Gitlab-Token`, y el secreto que valida la firma dice de qué repositorio viene. Cada commit, rama o PR que nombre una clave del proyecto (`WEB-12`, también en minúsculas) queda en `devLinks` del ticket con el nombre de su repositorio (`IssueDevLinkRecordedEvent`, actor `system`); el mismo enlace solo vuelve a contar si cambia (una PR fusionada). Proyecto desconocido, sin repositorios o firma mala responden igual: 403. Cambiar el secreto de un repositorio o quitarlo no afecta a los demás, y lo ya enlazado se queda. Los repositorios viven en MongoDB (`git_integrations`, un documento por proyecto), no en el event store; un proyecto que activó el webhook único de antes conserva su secreto como repositorio «principal».
- **Escaparate público**: `GET /api/explore/projects/{project}/showcase` resume un proyecto público para webs de fuera (la del colectivo lo enseña en la página de cada proyecto): épicas con fechas y progreso, las últimas versiones (con sus notas si ya salieron), el sprint en curso y los tickets abiertos con la etiqueta `good-first-issue`. Sin token, con `Cache-Control: public, max-age=300`; un proyecto privado responde 404.
- **Papelera**: `trashIssue` / `DELETE /api/issues/{issue}` manda un ticket a la papelera (quien lo abrió o un owner/ admin; `IssueTrashedEvent`): desaparece del tablero, las búsquedas, las estadísticas y el progreso, y no se puede cambiar hasta recuperarlo (`restoreIssue` / `POST …/restore`, `IssueRestoredEvent`). `trash(projectId)` / `GET /api/projects/{project}/trash` los lista. A los `TRASH_DAYS` días (30) se borra del todo, con la cascada de siempre: comentarios, enlaces y subtareas de vuelta al primer nivel.
- **Edición en bloque**: `bulkUpdateIssues` / `POST /api/issues/bulk` aplica el mismo cambio (estado, persona o `unassign`, sprint o `noSprint`, release o `noRelease`, prioridad, etiquetas que añadir) a hasta 100 tickets. Cada uno pasa por los comandos normales (permisos, flujo, sprints y releases abiertos) y puede negarse sin parar al resto; la respuesta dice cuáles.
- **Plantillas por tipo**: en la configuración, cada tipo de ticket puede llevar `template` con una descripción (hasta 5000 caracteres) y una checklist (hasta 20 pasos). Al crear un ticket de ese tipo, la descripción se usa si llega vacía y los pasos se añaden siempre (las importaciones no aplican plantillas).
- **Automatizaciones**: owners y admins crean reglas (hasta 50 por proyecto, por GraphQL: `automations`, `createAutomation`, `updateAutomation`, `deleteAutomation`). Disparador: ticket creado, cambio de estado (opcionalmente a uno concreto), asignación, comentario o PR fusionada (integración Git). Condiciones: tipo, prioridad, área o etiqueta. Acción: mover a un estado, asignar (a alguien o a quien lo abrió), añadir una etiqueta, poner una prioridad, comentar o meter en el sprint en curso. Actúan como `system` («TaskStream») y respetando el flujo: si la transición no está permitida, la regla no hace nada y lo deja en `lastError`. Lo que hace una regla no dispara otras (no hay cadenas ni bucles) y cada regla corre una sola vez por evento aunque se reentregue (`automation_runs`). Viven en MongoDB (`automations`), fuera del event store.
- **Webhooks salientes**: owners y admins dan de alta URLs (hasta 20 por proyecto) suscritas a eventos (`issue.created`, `issue.status_changed`, `comment.added`, `release.shipped`, `sprint.closed`… — la lista en `webhookEvents` / `GET /api/webhook-events`), a familias (`issue.*`) o

a todo (`*`). Cada envío es un `POST` JSON con `event`, `deliveryId`, `occurredAt`, `actorId`, `project`, `issue` (si va de un ticket) y `data` (los campos del evento), y las cabeceras `X-TaskStream-Event`, `X-TaskStream-Delivery` y `X-TaskStream-Signature: sha256=<HMAC del cuerpo con el secreto>`. El secreto solo se muestra al crearlo. Se reintenta a los 2 y 10 s (no ante un 4xx), con 5 s de espera máxima; se guarda el último envío. Salen en segundo plano y no son duraderos: si la API se reinicia a mitad, ese envío se pierde. No llaman a direcciones privadas salvo con `WEBHOOKS_ALLOW_PRIVATE=true`.

- **Importar y exportar**: `GET /api/projects/{project}/export.csv` da todos los tickets (UTF-8 con BOM; las celdas que una hoja de cálculo ejecutaría como fórmula empiezan por `'`). `POST /api/projects/{project}/import` (owners y admins, multipart `file`, hasta 5 MB y 2000 filas) lee ese mismo CSV o una exportación CSV de Jira: tipos, estados y personas se buscan por id o por nombre (los de Jira por su significado: *Story* → Feature, *Highest* → crítica, categoría *In Progress* → primer estado en curso...), lo que no encaja se queda con un valor por defecto y un aviso, y los padres se reconstruyen al final. Con `dryRun=true` solo se cuenta lo que se crearía. Cada ticket importado es una creación normal (con sus eventos), en el estado que traía si está activo.
- **Límite WIP**: cada estado puede tener `wipLimit` (1-999) en la configuración del proyecto. Es un aviso, no un bloqueo: la API acepta pasarse y el tablero marca la columna.
- **Sprints**: los planifica y gestiona quien administra el proyecto (owner o admin); mete tickets cualquier miembro. Un sprint va de `PLANNED` a `ACTIVE` (uno solo en curso por proyecto) y a `CLOSED`; dura como mucho 90 días y un sprint en curso conserva su fecha de inicio. Un sprint cerrado ya no acepta ni pierde tickets. Al cerrarlo se elige adónde van los tickets sin terminar: a otro sprint abierto del proyecto o al backlog; los terminados se quedan como registro. Solo se borran sprints que no han empezado (sus tickets vuelven al backlog).
- **Vistas guardadas**: filtros del tablero con nombre. Las guardan los miembros del proyecto (los lectores, solo para sí); compartir una la hace visible a todo el proyecto. Borra una su autor, o un admin si es compartida; como mucho 50 por persona y proyecto. Son preferencias, no historia: viven en MongoDB (`saved_filters`), fuera del event store, y se borran con el proyecto.
- **Checklist**: hasta 100 pasos de hasta 300 caracteres por ticket; los cambia quien puede editar el ticket.

## Notificaciones

Se derivan de los eventos (`NotificationsEventHandler`) y se guardan **por persona** en MongoDB: no son estado de ningún agregado, solo cada uno lee y descarta las suyas.

| Cuándo | A quién | Tipo |
|---|---|---|
| Se crea un ticket asignado o se asigna | La persona asignada | `ASSIGNED` |
| Un ticket cambia de estado | Quien lo tiene asignado y quien lo abrió | `STATUS_CHANGED` |
| Un ticket vuelve de QA sin pasar a hecho | Los mismos, con el motivo | `QA_FAILED` |
| Alguien comenta | Quien lo tiene asignado y quien lo abrió | `COMMENTED` |
| Alguien responde a un comentario | Quien escribió el comentario | `REPLIED` |
| Te añaden a un proyecto | La persona añadida | `ADDED_TO_PROJECT` |
| Llega una propuesta del público | Propietarios y admins del proyecto | `PROPOSED` |

**Recordatorios** (`REMINDER`): cada hora la API mira los tickets sin terminar con fecha límite y avisa a quien lo tiene asignado y a quien lo sigue el día antes, el mismo día y una vez cuando ya ha vencido (el día cambia en `REMINDERS_TIMEZONE`). Cada aviso tiene id `reminder:<ticket>:<fecha>:<fase>:<persona>`, así que no se repite aunque varias instancias lo comprueben a la vez.

Nadie recibe aviso de lo que hizo él mismo, y cada persona recibe uno por evento: la primera razón de la tabla gana (una mención antes que «comentó en tu ticket», y eso antes que «un ticket que sigues»).

**Menciones**: se escriben `@[Nombre](id-de-usuario)` (la web las inserta al escribir `@`); el id dice a quién avisar y el nombre es lo que se ve, así que renombrar a alguien no rompe comentarios antiguos. Solo avisan a miembros del proyecto. **Seguir**: cualquier miembro, también los lectores, puede seguir un ticket (`watchIssue` / `unwatchIssue`, eventos `IssueWatchedEvent` / `IssueUnwatchedEvent`), y quien lo abre, quien lo tiene asignado y quien comenta pasan a seguirlo (`PeopleFollowSaga`, `CommentersFollowSaga`). Quien sigue un ticket (`WATCHED`) sabe cuándo cambia de estado, se asigna, se comenta, cambia su prioridad o su fecha límite, o entra o sale de un sprint; las ediciones menores (título, etiquetas...) no avisan. «Los que sigo»: `watching` en `myIssues` / `GET /api/me/issues` y en la búsqueda de un proyecto (`issues(filter: { watching: true })`, `?watching=true`; sin sesión, ninguno). El id es `<agregado>:<versión>:<persona>`, así que reentregar un evento no duplica avisos. Llegan en vivo por la subscription `notificationAdded` y, si la instalación tiene `SMTP_URL`, también por correo a quien lo active (`email`, desactivado por defecto); un correo que falla se registra y no bloquea el stream.

## Eventos de dominio

Todo cambio es un evento guardado en el event store (stream = id del agregado). Llevan `actorId`, `occurredAt`, `aggregateId`, `projectId` y `aggregateVersion` (la posición en el stream).

| Agregado | Eventos |
|---|---|
| Project | `ProjectCreatedEvent`, `ProjectUpdatedEvent`, `ProjectSettingsChangedEvent`, `ProjectMemberAddedEvent`, `ProjectMemberRoleChangedEvent`, `ProjectMemberRemovedEvent`, `ProjectDeletedEvent` |
| Issue | `IssueCreatedEvent`, `IssueUpdatedEvent`, `IssueStatusChangedEvent`, `IssueAssignedEvent`, `IssueLinkedEvent`, `IssueUnlinkedEvent`, `IssueReleaseChangedEvent`, `IssueAttachmentAddedEvent`, `IssueAttachmentRemovedEvent`, `IssueWatchedEvent`, `IssueUnwatchedEvent`, `IssueParentChangedEvent`, `IssueRankedEvent`, `IssueSprintChangedEvent`, `IssueDevLinkRecordedEvent`, `IssueChecklistItemAddedEvent`, `IssueChecklistItemUpdatedEvent`, `IssueChecklistItemRemovedEvent`, `IssueDeletedEvent` |
| Comment | `CommentAddedEvent`, `CommentEditedEvent`, `CommentDeletedEvent` |
| Sprint | `SprintCreatedEvent`, `SprintUpdatedEvent`, `SprintStartedEvent`, `SprintClosedEvent`, `SprintDeletedEvent` |
| Release | `ReleaseCreatedEvent`, `ReleaseUpdatedEvent`, `ReleaseShippedEvent`, `ReleaseNotesWrittenEvent`, `ReleaseArchivedEvent`, `ReleaseDeletedEvent` |

`IssueStatusChangedEvent` lleva además `meta` con el nombre y la categoría de los dos estados en el momento del cambio, para que el historial y las estadísticas no cambien si luego se renombra o recategoriza un estado.

### Estadísticas

Una proyección más (`issue_timeline` en MongoDB) guarda cada paso de cada ticket: a qué estado pasó, con qué categoría, quién lo tenía asignado, y cuándo entró o salió de una release. Las estadísticas se calculan sobre ella con funciones puras (`application/src/lib/analytics`):

| Métrica | Definición |
|---|---|
| Creados vs. resueltos | Por semana (lunes, UTC): tickets creados y tickets que entraron en una categoría `DONE` |
| Flujo acumulado | Tickets en cada estado al final de cada día |
| Tiempo de ciclo | Del primer paso a `IN_PROGRESS` o `QA` hasta `DONE`; media y percentil 85 |
| Tiempo de entrega | De la creación hasta `DONE`; media y percentil 85 |
| Throughput | Tickets resueltos por semana y por la persona que los tenía asignados |
| QA | Salidas de un estado `QA`: a `DONE` cuenta como superado; a cualquier otro sitio salvo `CANCELLED`, como fallido |
| Burndown de un sprint | Por día del sprint: tickets en él (sin cancelados), cuántos siguen abiertos y sus puntos; línea ideal hasta cero |
| Burnup de una release | Por día, desde que se creó hasta que se publica (o hoy, máx. 180 días): tickets en ella (sin cancelados) y hechos |

Los tickets anteriores a esta proyección solo tienen su creación y su estado actual; para tener su historia completa, reconstruye el read model (ver Despliegue).

## 3.2 GraphQL

Las operaciones requieren `Authorization: Bearer <token de Pireon>` (en subscriptions, en el `connection_init`), salvo las lecturas de proyectos públicos (`project`, `issue`, `issues`, `release`, `activity`, `publicProjects` y sus campos), que funcionan también sin token.

### Queries

| Query | Devuelve |
|---|---|
| `me` | El usuario autenticado, con sus roles en Pireon |
| `users(search, limit)` | Usuarios de Pireon (para asignar o añadir miembros) |
| `user(id)` | Un usuario |
| `myProjects` | Proyectos de los que soy miembro |
| `myIssues(assignedToMe, statuses, categories, search, limit, offset)` | «Mi trabajo»: tickets de todos mis proyectos, asignados a mí (por defecto) o de cualquiera (p. ej. `categories: [QA]`) |
| `upcomingReleases(limit)` | Releases planificadas de mis proyectos, por fecha objetivo |
| `publicProjects` | Proyectos públicos (sin token) |

| Query | Devuelve |
|---|---|
| `project(idOrKey)` | Un proyecto (mío, o público) con su `settings`. Campos calculados: `board(releaseId, assigneeId, unassigned, area, types, priorities, labels, search, perColumn)`, `stats`, `analytics(days)`, `releases` |
| `issues(filter, limit, offset)` | Búsqueda de tickets (estado, tipo, área, prioridad, persona, release, etiquetas, ticket padre, fecha límite entre `dueFrom` y `dueTo`, texto); `sort: RANK` para el orden del backlog |
| `webhooks(projectId)`, `webhookEvents` | Webhooks salientes del proyecto (owners y admins) y los eventos a los que se pueden suscribir |
| `gitIntegration(projectId)` | Si el webhook de Git está activo, su ruta y (para owners y admins) el secreto |
| `sprints(projectId)`, `sprint(id)` | Sprints del proyecto (en curso, planificados, cerrados). Campos calculados: `progress`, `burndown`, `issues` |
| `savedFilters(projectId)` | Mis vistas guardadas del tablero y las compartidas con el proyecto |
| `epics(projectId)` | Épicas del proyecto (tickets de primer nivel de los que cuelgan otros) con el progreso de sus hijos |
| `issue(idOrKey)` | Un ticket. Campos calculados: `allowedTransitions`, `linkedIssues`, `comments`, `history` |
| `release(id)` | Una release. Campos calculados: `progress`, `burnup`, `issues` |
| `activity(projectId, issueId?, limit, before?)` | Feed de actividad, del más reciente al más antiguo |

## Mutations

Todas devuelven `CommandResult { id, key, version }`.

| Ámbito | Mutations |
|---|---|
| Proyectos | `createProject`, `updateProject`, `updateProjectSettings`, `deleteProject(confirmKey)`, `addProjectMember`, `changeProjectMemberRole`, `removeProjectMember` |
| Tickets | `createIssue`, `updateIssue`, `changeIssueStatus`, `assignIssue`, `linkIssues`, `unlinkIssues`, `setIssueRelease`, `setIssueParent`, `rankIssue`, `addChecklistItem`, `updateChecklistItem`, `removeChecklistItem`, `watchIssue`, `unwatchIssue`, `removeAttachment` (subir: por REST), `duplicateIssue`, `voteIssue` |
| Comentarios | `addComment(replyTo)`, `editComment`, `deleteComment`, `reactToComment` |
| Webhooks | `createWebhook` (devuelve el secreto), `updateWebhook`, `deleteWebhook`, `testWebhook` |
| Git | `addGitRepository` (devuelve su secreto), `rotateGitRepositorySecret`, `removeGitRepository` |
| Sprints | `createSprint`, `updateSprint`, `startSprint`, `closeSprint(carryOverToSprintId)`, `deleteSprint`, `setIssueSprint` |
| Vistas | `saveFilter(input)` (devuelve el id), `deleteSavedFilter(id)` |
| Avisos | `markNotificationsRead(ids)`, `updateNotificationSettings(email)` |

| Ámbito | Mutations |
|---|---|
| Releases | `createRelease`, `updateRelease`, `publishRelease(force)`, `writeReleaseNotes`, `archiveRelease` |

## Subscriptions

| Subscription | Se emite cuando |
|---|---|
| `issueChanged(projectId)` | Se crea o cambia cualquier ticket del proyecto (incluye el ticket completo y `action` = nombre del evento) |
| `commentChanged(issueId)` | Se añade, edita o borra un comentario ( `comment` es `null` al borrar, o el que queda sin texto si tenía respuestas) |
| `activityAdded(projectId)` | Hay una nueva entrada en el feed |
| `releaseChanged(projectId)` | Cambia una release |
| `sprintChanged(projectId)` | Se planifica, cambia, empieza, cierra o borra un sprint ( `sprint` es `null` al borrar) |
| `projectChanged(projectId)` | Cambian los datos, la configuración o los miembros del proyecto |
| `myProjectsChanged` | Me añaden a un proyecto o me quitan de uno |

Solo los miembros pueden suscribirse a un proyecto; las conexiones sin token válido se cierran con el código 4403.

## Ejemplos

Estados, tipos y áreas son texto (ids de la configuración del proyecto), no enums: van entre comillas.

```
mutation ($settings: ProjectSettingsInput!) {
  # La configuración entera (la que devuelve project.settings, cambiada).
  updateProjectSettings(projectId: "…", settings: $settings) {
    version
  }
}

mutation {
  createProject(input: { key: "SHOP", name: "Checkout revamp" }) {
    id
    key
    version
  }
}

mutation {
  createIssue(input: { projectId: "…", type: "BUG", title: "El botón de pagar no responde en
Safari", area: "FRONTEND", priority: CRITICAL, assigneeId: "102", labels: ["safari"] }) {
    id
    key
```

```
    version
  }
}

mutation {
  changeIssueStatus(input: { issueId: "…", status: "QA_FAILED", reason: "Sigue fallando en
Safari 17" }) {
    version
  }
}

mutation {
  linkIssues(input: { issueId: "<FE>", type: DEPENDS_ON, targetIssueId: "<BE>" }) {
    version
  }
}

query {
  project(idOrKey: "SHOP") {
    stats {
      total
      open
      byStatus {
        key
        count
      }
      byAssignee {
        user {
          name
        }
        count
      }
    }
    board {
      status
      total
      issues {
        key
        title
        assignee {
          name
        }
      }
    }
    releases {
      name
      status
      progress {
```

```
        percent
      }
    }
    analytics(days: 90) {
      createdVsResolved {
        week
        created
        resolved
      }
      cycleTime {
        averageDays
        p85Days
      }
      qa {
        failureRate
      }
    }
  }
  release(id: "…") {
    burnup {
      dates
      scope
      done
    }
  }
  issue(idOrKey: "SHOP-1") {
    status
    allowedTransitions
    links {
      type
      direction
      issueKey
    }
    comments {
      items {
        author {
          name
        }
        body
      }
    }
    history {
      summary
      occurredAt
    }
  }
}
```

```
import { createClient } from 'graphql-ws';

const client = createClient({
  url: 'ws://localhost:3000/graphql',
  connectionParams: { authorization: `Bearer ${token}` },
});

client.subscribe(
  {
    query: 'subscription($p: ID!) { issueChanged(projectId: $p) { action version issue { key
status } } }',
    variables: { p: projectId },
  },
  { next: console.log, error: console.error, complete: () => {} },
);
```

### Errores

Los errores de dominio llegan como errores GraphQL con `extensions.code`:

| Código | Significado |
|---|---|
| `UNAUTHENTICATED` | Falta el token o Pireon lo rechaza |
| `NOT_FOUND` | No existe o no eres miembro |
| `FORBIDDEN` | No tienes permiso (p. ej. un `VIEWER` modificando) |
| `VALIDATION` | Datos inválidos (clave de proyecto, URL de Figma...) |
| `BUSINESS_RULE` | Regla de negocio (transición no permitida, release bloqueada...) |
| `CONFLICT` | Otro cambio llegó antes (concurrencia optimista): reintenta |

## 3.3 REST con OpenAPI (para otras apps)

Los mismos casos de uso que GraphQL, como API REST documentada con OpenAPI 3.0, para que otras apps de la plataforma la consuman o generen un cliente. Son otro adaptador de entrada (`presenters/http`) que llama a los mismos `*.service.ts`: misma validación, permisos y errores.

| | |
|---|---|
| Swagger UI | `/api/docs` (con «Authorize» contra Pireon) |
| Documento OpenAPI | `/api/docs-json` (y `/api/docs-yaml`) |
| Fichero exportado | `apps/api/openapi.json`: lo reescribe `nx serve api` al arrancar, como `schema.gql` |

### Autenticación: siempre a través de Pireon

Cualquier app o usuario que la consuma pasa por el OAuth de Pireon: todas las rutas, salvo `health`, `auth`, el webhook, `explore` y las lecturas de proyectos públicos, exigen `Authorization: Bearer <token de Pireon>` (el `AuthGuard` global, igual que en GraphQL). En el documento OpenAPI, las rutas

donde el token es opcional lo indican con un requisito de seguridad vacío ( `{}` ). Declara dos esquemas:

- `pireon-oauth` (OAuth2): *Authorization Code + PKCE* contra `oauth.pireon.dev/oauth/authorize` y, si `PIREON_PASSWORD_GRANT=true`, *password*. El intercambio de tokens va por `POST /api/auth/token` (Pireon no tiene CORS en `/oauth/token` y el `client_secret` se queda en el servidor). Para usar «Authorize» en Swagger UI, registra `<api>/api/docs/oauth2-redirect.html` como redirect URI del cliente en Pireon.
- `pireon` (Bearer): para una app que ya tiene un token de Pireon, por ejemplo con su propio cliente OAuth.

Lo que cada usuario puede ver o cambiar depende de su rol en cada proyecto, como en la web.

## Recursos

Proyectos y tickets se pueden indicar por **id o por clave** ( `WEB` , `WEB-12` ).

| Recurso | Rutas |
|---|---|
| Usuarios | `GET /api/users/me` , `GET /api/users?search=` , `GET /api/users/{id}` |
| Proyectos | `GET/POST /api/projects` , `GET/PATCH/DELETE /api/projects/{project}` ( `DELETE …?confirm=CLAVE` ), `PUT …/settings` , `…/board` , `…/epics` , `…/stats` , `…/analytics?days=` , `…/activity` , `…/members` (POST, PATCH, DELETE) |
| Tickets | `GET /api/me/issues` (mi trabajo), `GET/POST /api/projects/{project}/issues` , `GET/PATCH /api/issues/{issue}` , `PUT …/status` , `PUT …/assignee` , `PUT …/release` , `PUT/DELETE …/watch` , `PUT …/parent` , `PUT …/rank` , `POST …/checklist` , `PATCH/DELETE …/checklist/{item}` , `POST/DELETE …/links` , `GET/POST …/comments` , `GET …/history` , `POST /api/issues/{issue}/duplicate` , `PUT/DELETE …/vote` |
| Comentarios | `PATCH/DELETE /api/comments/{commentId}` , `POST …/reactions` |
| Sprints | `GET/POST /api/projects/{project}/sprints` , `GET/PATCH/DELETE /api/sprints/{sprint}` , `POST …/start` , `POST …/close` , `GET …/progress` , `…/burndown` , `…/issues` , `PUT /api/issues/{issue}/sprint` |
| Vistas | `GET/POST /api/projects/{project}/filters` , `DELETE /api/filters/{id}` |
| Avisos | `GET /api/me/notifications` , `POST /api/me/notifications/read` , `GET/PUT /api/me/notification-settings` |
| Adjuntos | `POST /api/issues/{issue}/attachments` (multipart: `file` y opcional `commentId` ), `GET/DELETE /api/issues/{issue}/attachments/{id}` |
| Releases | `GET /api/me/releases` (próximas), `GET/POST /api/projects/{project}/releases` , `GET/PATCH /api/releases/{id}` , `PUT …/notes` , `GET …/notes/draft` , `…/progress` , `…/burnup` , `…/issues` , `POST …/publish` , `POST …/archive` |
| Explorar | `GET /api/explore/projects` (proyectos públicos, sin token) |
| Importar | `GET /api/projects/{project}/export.csv` , `POST /api/projects/{project}/import?dryRun=` (multipart `file` ) |
| Webhooks | `GET /api/webhook-events` , `GET/POST /api/projects/{project}/webhooks` , `PATCH/DELETE /api/outgoing-webhooks/{id}` , `POST …/test` |

| Recurso | Rutas |
|---|---|
| Git | `GET /api/projects/{project}/git-integration`, `POST …/git-integration/repositories`, `POST …/repositories/{id}/secret`, `DELETE …/repositories/{id}`, `POST /api/webhooks/git/{project}` (lo llama el host de Git) |
| Explorar | `GET /api/explore/projects/{project}/showcase` (sin token, proyectos públicos): hoja de ruta con votos, versiones, sprint en curso, tickets `good-first-issue` y propuestas abiertas; `POST …/proposals` (sin token) |
| Sistema | `GET /api/health`, `GET /api/auth/config`, `POST /api/auth/token`, `POST /api/webhooks/pireon` |

Los archivos van por REST porque GraphQL no está hecho para bytes; la lista de adjuntos de un ticket llega con el ticket (`attachments`, en GraphQL y en REST). La descarga sirve en línea solo imágenes, vídeo, audio, PDF y texto plano, y todo con `X-Content-Type-Options: nosniff` y una CSP `sandbox`: un HTML o un SVG subido se descarga, nunca se ejecuta en el origen de la API.

Los filtros de tickets aceptan `status` (ids del proyecto) y `category` (`?category=QA`), repetidos o separados por comas. Las escrituras responden `{ id, key?, version }` en cuanto el evento se guarda; las lecturas salen del read model, que se pone al día en milisegundos (un `GET` justo después puede devolver la versión anterior, o 404 tras crear). Para tiempo real, las subscriptions de GraphQL.

## Errores

`{ statusCode, code?, message }`: `400` entrada no válida (`message` es una lista), `401` sin token o token rechazado por Pireon, `403` / `404` sin acceso, `409` cambio concurrente (reintentar) y `422` regla de negocio (p. ej. una transición que el workflow no permite).

## Generar un cliente en otra app

```
npx openapi-typescript apps/api/openapi.json -o src/task-stream-api.d.ts      # tipos
# o un cliente completo: openapi-generator, orval, ng-openapi-gen…
```

Ver también Autenticación.

# 4. Cliente web

Angular 22 (componentes standalone, zoneless, Signal Forms) en `apps/web`, con la misma arquitectura hexagonal que la API repartida en `libs/web/*`.

## 4.1 Librerías

| Librería | Contenido |
|---|---|
| `web/shared/infrastructure` | Cliente Apollo (`provideGraphql`: HTTP + `graphql-ws`), `API_CONFIG`, puerto `AccessTokenSource` |
| `web/shared/ui` | Integración de `@pi/ui` (`providePiUi`), modo oscuro, iconos, `ToastService`, `DrawerService`, `AppShellComponent` |
| `web/identity/domain` | `Session`, `CurrentUser`, `AuthConfig` y puertos `AuthGateway`, `ProfileGateway`, `SessionStorage`, `AuthorizationFlow` |
| `web/identity/application` | `AuthFacade` (pública); por dentro `AuthStore` y eventos `authPageEvents`, `authApiEvents`, `sessionEvents` |
| `web/identity/infrastructure` | `/api/auth/*` (HTTP), `me` (GraphQL), `localStorage`, PKCE con Web Crypto; enlaza `AccessTokenSource` con la `AuthFacade` |
| `web/identity/presenters` | Página de login, callback de Pireon, `authGuard` / `guestGuard`, menú de usuario |
| `web/tracker/domain` | Modelos, enums, workflow y puertos (`ProjectsGateway`, `IssuesGateway`, `TrackerRealtimeGateway`...) |
| `web/tracker/application` | Facades `ProjectsFacade`, `ProjectFacade`, `BoardFacade`, `IssueFacade` y puerto `Notifications`; por dentro, stores y eventos |
| `web/tracker/infrastructure` | Adaptadores GraphQL con operaciones tipadas (codegen) |
| `web/tracker/presenters` | Páginas: proyectos, tablero, lista, ticket, propuestas, releases, seguimiento, actividad y miembros; `ts-markdown` (texto con formato, saneado con DOMPurify), `ts-bulk-bar`, `ts-vote` |

## 4.2 Estado: NgRx Signals con eventos

Las páginas **no llaman a servicios ni a los stores**: hablan con una **facade**, que lee el estado como señales y traduce cada intención en un evento. Los stores reaccionan con reducers (estado) y con handlers (efectos), y los efectos emiten nuevos eventos con el resultado.

Cada store agrupa sus eventos por origen:

- `*PageEvents`: lo que hace el usuario (`opened`, `issueMoved`, `issueCreateRequested`...).
- `*ApiEvents`: lo que responde la API (`loaded`, `loadFailed`, `commandSucceeded`...).
- `*RealtimeEvents`: lo que llega por subscription (`issueChanged`, `activityAdded`...).

Ejemplo real (`BoardStore`, simplificado):

```
export const boardPageEvents = eventGroup({
  source: 'Board Page',
  events: {
    opened: type<BoardFilter>(),
    issueMoved: type<{ issueId: string; to: string; reason?: string | null }>(),
  },
});

export const BoardStore = signalStore(
  withState<BoardState>({ filter: null, columns: [], loading: false, … }),
  withReducer(
    on(boardPageEvents.opened, ({ payload }) ⇒ ({ filter: payload, loading: true })),
    on(boardApiEvents.loaded, ({ payload }) ⇒ ({ columns: payload, loading: false })),
    // Optimista: la tarjeta se mueve al instante…
    on(boardPageEvents.issueMoved, ({ payload }, s) ⇒ ({ columns: moveIssue(s.columns,
payload.issueId, payload.to) })),
    on(boardRealtimeEvents.issueChanged, ({ payload }, s) ⇒ ({ columns:
applyIssueChange(s.columns, payload.issue, s.filter) })),
  ),
  withEventHandlers((store, events = inject(Events), issues = inject(IssuesGateway)) ⇒ ({
    load$: events.on(boardPageEvents.opened).pipe(switchMap(({ payload }) ⇒
issues.board(payload).pipe(
      mapResponse({ next: boardApiEvents.loaded, error: (e) ⇒
boardApiEvents.loadFailed(describeError(e)) }),
    ))),
    // …y vuelve a su sitio si la API rechaza el cambio.
    move$: events.on(boardPageEvents.issueMoved).pipe(mergeMap(({ payload }) ⇒
      issues.changeStatus({ issueId: payload.issueId, status: payload.to, reason: payload.reason
}).pipe(
        mapResponse({ next: () ⇒ null, error: (e) ⇒
boardApiEvents.issueMoveFailed(describeError(e)) }),
      ))),
  })),
);
```

## Facades

Cada store tiene su facade en `application`, y es lo **único** que exportan esas librerías (junto al puerto `Notifications`): los stores y los eventos quedan como detalle interno.

```
@Injectable()
export class BoardFacade {
  private readonly store = inject(BoardStore);
  private readonly dispatch = injectDispatch(boardPageEvents);

  readonly columns = this.store.columns; // estado: señales de solo lectura
  readonly loading = this.store.loading;
```

```
  /** Abre el tablero si es otro proyecto; si es el mismo, solo cambia el filtro. */
  show(filter: BoardFilter): void {
    const same = this.store.filter()?.projectId === filter.projectId;
    if (same) this.dispatch.filterChanged(filter);
    else this.dispatch.opened(filter);
  }

  moveIssue(issueId: string, to: string, reason?: string | null): void {
    this.dispatch.issueMoved({ issueId, to, reason });
  }
}

export const provideBoardFacade = () => [BoardStore, BoardFacade];
```

```
// En la página
@Component({ providers: [provideBoardFacade()], … })
export class BoardPage {
  protected readonly board = inject(BoardFacade);
  drop(event) { this.board.moveIssue(issueId, to); }
}
```

Qué gana la página:

- **Intenciones, no eventos**: `board.moveIssue(…)` en lugar de conocer `boardPageEvents` y la forma de su payload.
- **Contexto implícito**: `ProjectFacade` e `IssueFacade` ponen el id del proyecto o del ticket abierto (`project.addMember(userId)`, `issue.addComment(body)`); si aún no ha cargado, el comando se ignora.
- **Un sitio para la lógica de presentación**: abrir o refiltrar el tablero, buscar usuarios a partir de 2 caracteres…
- **Tests sin plantilla**: la facade se prueba con gateways falsos (`facades.spec.ts`).

| Facade | Store | Ámbito | Responsabilidad |
|---|---|---|---|
| `AuthFacade` | `AuthStore` | raíz | Sesión, usuario, login (PKCE o email y contraseña de Pireon), renovación del token, restaurar al arrancar |
| `ProjectsFacade` | `ProjectsStore` | raíz | Mis proyectos; se actualiza cuando me añaden o quitan (`myProjectsChanged`) |
| `HomeFacade` | `HomeStore` | raíz | «Mi trabajo» en la portada: mis tickets abiertos por categoría (QA fallidos primero), esperando QA y próximas releases; se recarga al volver a la pestaña |
| `ExploreFacade` | `ExploreStore` | raíz | Proyectos públicos («Explorar»), también sin sesión |

| Facade | Store | Ámbito | Responsabilidad |
|---|---|---|---|
| `ProjectFacade` | `ProjectStore` | ruta `/p/:key` | Proyecto, miembros, stats, releases, actividad en vivo, búsqueda de usuarios, configuración (`workflow`, `saveSettings`) y estadísticas (`loadAnalytics`, `loadBurnup`) |
| `BoardFacade` | `BoardStore` | ruta `board` | Tablero kanban, filtros, movimiento optimista, nuevo ticket, cambios en vivo |
| `ListFacade` | `ListStore` | ruta `list` | Todos los tickets del proyecto en filas: filtros (enviados a la API), cambios de estado, prioridad y persona en la fila, edición en bloque; se recarga tras cada cambio propio y con los de otros |
| `IssueFacade` | `IssueStore` | ruta `issues/:issueKey` | Ticket, comentarios en vivo, historial y todas sus acciones |

Las de ámbito de ruta se registran con `provideProjectFacade()`, `provideBoardFacade()` o `provideIssueFacade()` en los `providers` del componente de la página (store + facade) y desaparecen al salir (con sus subscriptions). Las páginas hijas de `/p/:key` reciben la misma `ProjectFacade`.

Otros contextos también usan la facade: los stores del tracker leen el usuario de `AuthFacade`, y la infraestructura de identidad (`SessionTokenSource`) le pide el token para el cliente GraphQL.

### Tiempo real y consistencia

- Cada store se suscribe solo a lo que muestra (`issueChanges(projectId)`, `commentChanges(issueId)`...).
- Los cambios se fusionan por `version`: si llega un evento más antiguo que lo que hay en pantalla, se ignora.
- Tras crear algo, la query puede devolver *not found* unos milisegundos (el read model es eventualmente consistente); los stores lo reintentan hasta 5 veces.
- Los resultados de los comandos se comunican con el puerto `Notifications` (en la app, toasts de `pi-snack-bar`).

## 4.3 GraphQL

- **Apollo** (`apollo-angular` + `@apollo/client` 4) con la caché **desactivada**: los stores de NgRx son la única fuente de verdad.
- Las subscriptions van por `graphql-ws` con el token en `connectionParams`; si la API responde `UNAUTHENTICATED`, la sesión se cierra.
- El cliente manda un *ping* cada 10 s y, si no llega el *pong* en 5 s, cierra el socket y reconecta (una conexión medio muerta, p. ej. tras suspender el portátil o pasar por un proxy, no se nota de otro modo). Al reconectar, `RealtimeConnection.reconnected$` avisa y cada store vuelve a pedir lo que muestra, porque lo que pasó mientras tanto no llega por las subscriptions.
- Las operaciones están en `libs/web/*/infrastructure/src/lib/graphql/*.graphql`. Tras cambiar la API o una operación:

```
npx nx serve api           # reescribe apps/api/schema.gql
npx nx run web:codegen     # regenera generated.ts (TypedDocumentNode)
```

- Los adaptadores convierten los tipos generados en modelos de dominio en un único sitio (`adapters/mappers.ts`). Lo que se lee y luego se reenvía como input (la configuración del proyecto) se copia campo a campo: Apollo añade `__typename` a cada objeto y los input types de GraphQL lo rechazan.

## 4.4 `@pi/ui`

La librería de componentes de Pireon se instala desde su Nexus (scope `@pi`, configurado en `.npmrc`; el token va en `~/.npmrc`). Se integra igual que en `landing-web`, y conviene conocer sus particularidades:

| Particularidad | Cómo se resuelve |
|---|---|
| `styles/common/index.scss` no compila con Material ≥ 19 | `apps/web/src/styles.scss` importa sus partes una a una y declara el tema con `mat.theme` |
| No trae modo oscuro | Redefinimos sus tokens `--pi-color-*` bajo `html.dark-mode .pi-shell` (paleta Pireon, igual que la landing) |
| `PiThemeService` persiste mal el modo oscuro (`"false"` se lee como verdadero) | `AppearanceService` guarda su propia preferencia |
| No trae SVGs: `pi-icon` busca en el registro de Material | `shared/ui/src/lib/icons.ts` registra los iconos en línea, incluidos los que usa la propia librería (`eye`, `eye-off`, `add` ...) |
| Sus textos (errores de campos, "Cerrar") van por ngx-translate | `providePiStrings()` los registra al arrancar, sin peticiones HTTP |
| Varios inputs son obligatorios (`input.required`) | `pi-note` necesita `gap`, `status`, `title` y `text` |
| Los campos usan Signal Forms | `[formField]="form.campo"` para formularios; `[(value)]` para controles sueltos |
| Drawers y modales se abren con `MatDialog` | `DrawerService.open(título, template)` con `getDrawerConfigs` de la librería |
| `pi-search-field` es un buscador plegable (un icono) | Para un campo de búsqueda normal se usa `pi-text-field` |

| `button` globales: `min-width`, padding y radio con selectores muy específicos | Los botones propios de la app usan `.ts-icon-button`, o anulan esas reglas con `!important` (votos, reacciones); `pi-action-button size="w100"` necesita `box-sizing` (en `styles.scss`) | | Los campos pintan cualquier error que reciben, aunque no se hayan tocado | En el login, los errores se pasan solo tras salir del campo o intentar entrar (`(touch)`) |

Componentes en uso: `pi-action-button`, `pi-icon-button`, `pi-icon`, `pi-note`, `pi-text-field`, `pi-email-field`, `pi-password-field`, `pi-textarea`, `pi-select-field`, `pi-segmented-control`, `pi-skeleton`, `pi-spinner`, `pi-empty-state`, `pi-divider`, `pi-drawer` y `pi-snack-bar`.

## 4.5 Diseño

TaskStream es un producto de la plataforma Pireon y se ve como tal: comparte con pireon.dev (`landing-web`) el logo, la paleta, la tipografía, la cabecera y las superficies. `@pi/ui` aporta los controles; lo propio vive en `apps/web/src/styles.scss` como tokens CSS, con su versión oscura bajo `html.dark-mode .pi-shell`.

| De pireon.dev | En TaskStream |
|---|---|
| Logo (`logo.png`, `favicon.ico`) | `apps/web/public`, pintado por `LogoComponent` |
| Rampas `primary` (#a347e1) y `accent` (#1879bf) sacadas del logo | Las mismas `brand-ramp` / `dark-ramp` |
| Inter; fondo `#f7f7fa` (oscuro `#0f1116`) | Igual (`--ts-font-body`, `--ts-bg`) |
| Cabecera translúcida con desenfoque, borde y sombra al hacer scroll | `AppShellComponent`, con «Pireon \| TaskStream» |
| Luces difusas violeta y azul arriba de cada página | `.ts-main::before/::after` |
| *Eyebrow* azul en mayúsculas y titulares con un tramo en degradado | `.ts-eyebrow` y `.ts-gradient-text` |
| Tarjeta con lavado violeta→azul y borde en degradado (`landing-card`) | `.ts-panel` (paneles) y `.ts-card-link` (tarjetas que navegan, con elevación) |
| Cifras con `pi-stat-card` | Resumen de Seguimiento |

Encima, el lenguaje del producto: `--ts-font-mono` (Geist Mono) solo para datos (claves `WEB-12`, contadores, versiones) y un color por estado (`--ts-s-*`) y por tipo de ticket (`--ts-t-*`), el mismo en tablero, tarjetas, gráficas e historial. Salen de `tone` en `ui/copy.ts` y las plantillas los aplican con `[style.--tone]`; utilidades globales como `.ts-pill` o `.ts-key` leen esa variable. Las animaciones se desactivan con `prefers-reduced-motion`.

### Configuración del proyecto en la UI

Estados, tipos y áreas son de cada proyecto, así que la UI nunca los tiene fijos: `ProjectFacade.workflow()` envuelve `project.settings` (`Workflow` en el dominio web) y da nombre, color, icono, categoría, transiciones permitidas y si un estado pide motivo. `statusLook` / `typeLook` de `ui/copy.ts` lo convierten en etiqueta y tono (`--ts-c-<color>`, con su versión oscura). Un id que ya no está configurado sigue mostrándose (`IN_REVIEW` → «In review»). «Mi trabajo» mezcla proyectos: cada fila usa el workflow de su proyecto y los grupos son por categoría.

En el tablero, un ticket solo se puede soltar en las columnas a las que su workflow le deja pasar, y los tipos archivados no se ofrecen al crear. Los tipos con icono `palette` piden un frame de Figma.

### Gráficas

Las de Estadísticas son SVG propios (`ui/series-chart.ts`), no `pi-line-chart`: hacen falta áreas apiladas y una paleta con versión oscura, que los gráficos de `@pi/ui` (Chart.js con colores fijos) no traen. Siguen las reglas de visualización de la plataforma:

- Colores `--ts-viz-*` en `styles.scss`, validados contra la superficie clara y la oscura (separación para daltonismo, contraste): dos series categóricas (azul y naranja), una rampa ordinal de un solo

tono para las etapas del flujo y bueno/crítico reservados para QA superado/fallido, siempre con icono y texto.
- Líneas de 2 px, rejilla de 1 px continua, separación de 2 px entre bandas apiladas, una sola escala por gráfica.
- Cruceta que se ajusta al día o semana más cercano y un único tooltip con todas las series; con teclado, ← → Inicio Fin sobre la gráfica.
- Leyenda con dos o más series, etiqueta de valor al final de cada línea solo si no se solapan, y «Ver los datos» con la tabla, para que nada dependa del color ni del ratón.
- Mientras se recalcula un periodo, se queda la gráfica anterior atenuada.

### Proyectos públicos y visitantes sin sesión

- Solo la portada ( `/` ) exige sesión. `/explorar` y las páginas de proyecto ( `/p/:key/…` ) no: la API decide qué ve cada visitante, y las páginas ya ocultan lo que no puede cambiar ( `canContribute` , `canManage` ).
- Sin sesión, la cabecera muestra «Explorar» e «Iniciar sesión»; dentro de un proyecto público aparece un aviso de modo lectura, y en uno privado (o inexistente), el enlace para iniciar sesión volviendo a esa página ( `returnUrl` ).
- El tiempo real necesita sesión: `GraphqlRealtimeGateway` no abre subscriptions sin token (si lo hiciera, la API cerraría el WebSocket y el cliente reintentaría sin fin). Los visitantes anónimos ven los datos al cargar la página.
- Configuración tiene el interruptor «Público» (propietario y admins) y, solo para el propietario, «Eliminar el proyecto», que exige escribir la clave. Al borrarlo se vuelve a la portada; si otro lo borra mientras lo tienes abierto, la subscription avisa ( `ProjectChanged.deleted` ) y también se vuelve a la portada.

## 4.6 Rutas

| Ruta | Página |
|---|---|
| `/login` | Inicio de sesión (solo sin sesión) |
| `/auth/callback` | Retorno de Pireon (PKCE) |
| `/` | Inicio: «Ir a…» ( `WEB-12` ), mi trabajo ( `ts-my-work` ) y mis proyectos |
| `/explorar` | Proyectos públicos (sin sesión) |
| `/p/:key/board` | Tablero kanban ( `?release=` y `?assignee=` preseleccionan los filtros) |
| `/p/:key/issues/:issueKey` | Ticket |
| `/p/:key/releases` | Releases |
| `/p/:key/dashboard` | Seguimiento: cifras, flujo por estado, equipo (cada persona enlaza a su tablero) y tipos |
| `/p/:key/analytics` | Estadísticas: creados vs. resueltos, flujo acumulado, burnup de una release, tiempos de ciclo y entrega, QA y resueltos por persona (30 días a 1 año) |

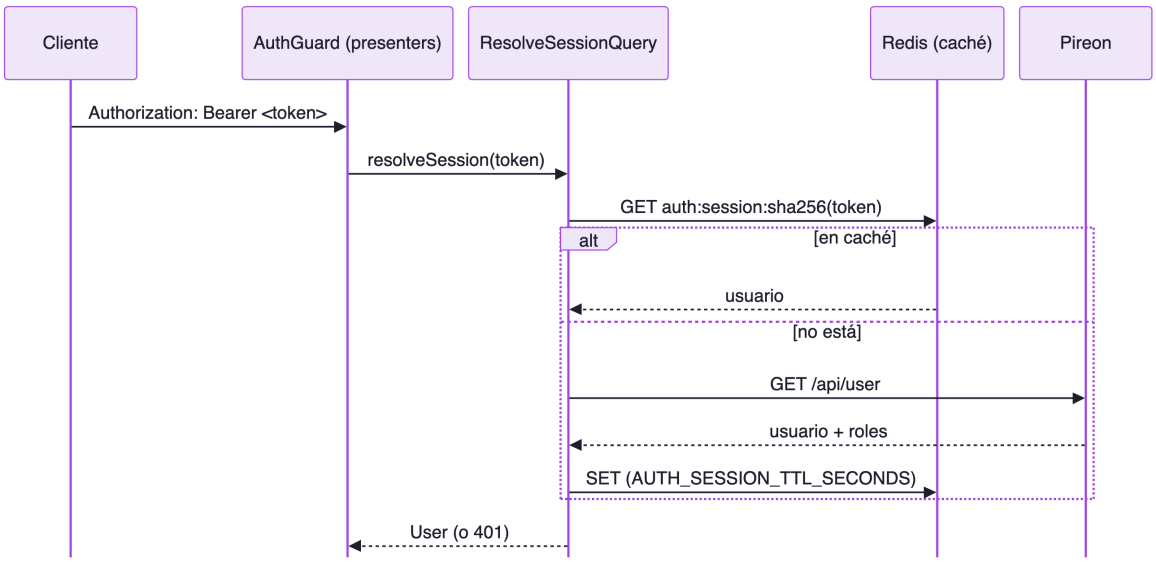
| Ruta | Página |
|---|---|
| `/p/:key/settings` | Configuración (propietario y admins; pestaña oculta para el resto): estados (orden, categoría, color, motivo), transiciones, tipos y áreas; archivar en lugar de borrar |
| `/p/:key/activity` | Actividad en vivo |
| `/p/:key/members` | Miembros |

Solo `/` está protegida por `authGuard` (redirige a `/login?returnUrl=…`); `/login` y `/auth/callback` son para quien no tiene sesión, y el resto se puede abrir sin ella si el proyecto es público. Los parámetros de ruta llegan como inputs de los componentes (`withComponentInputBinding`).

# 5. Autenticación (Pireon)

Los usuarios pertenecen a **Pireon** (`https://oauth.pireon.dev`, OAuth2 con Laravel Passport). TaskStream no guarda contraseñas: solo valida tokens y guarda una copia local de nombre y email para mostrarlos.

## 5.1 Cómo valida la API un token




- Se cachea el **hash SHA-256** del token, nunca el token en claro.
- Si Pireon marca al usuario como inactivo, el token se rechaza.
- El guard es global: todo requiere token salvo lo marcado con `@Public()` (health, configuración de login, proxy de tokens, webhook y `GET /api/explore/projects`).
- `@OptionalAuth()` (o `@ApiOptionalPireonAuth()` en REST) deja pasar sin token: es para las lecturas que permiten los proyectos públicos (proyecto, tablero, tickets, comentarios, releases, actividad, estadísticas). Sin token, el usuario es `undefined` y el visitante es anónimo; un token presente pero inválido sigue dando 401. Las escrituras, las subscriptions y lo «mío» (`myProjects`, `myIssues`, `/api/me/…`) siguen exigiendo sesión.
- En subscriptions, el token se valida una vez al abrir el WebSocket (`connection_init`).
- `@RequireAppRoles('…')` permite exigir roles del cliente OAuth en Pireon (los admins globales de Pireon los tienen todos).

## 5.2 Iniciar sesión desde el navegador

Pireon **no permite CORS** en `/oauth/token` y el `client_secret` no debe llegar al navegador, así que la web pasa por la API: `POST /api/auth/token` hace de proxy hacia Pireon.

Cada cliente OAuth de Pireon admite **un único tipo de grant**, así que cada forma de entrar usa su cliente:

| Método | Cliente de Pireon (`grant_type`) | Variables | Flujo |
|---|---|---|---|
| **Entrar con Pireon** (Authorization Code + PKCE) | `authorization_code` | `PIREON_CLIENT_ID`, `PIREON_CLIENT_SECRET` | Web → `/oauth/authorize` de Pireon → `/auth/callback` → `POST /api/auth/token` (`authorization_code` + `code_verifier`) |
| **Email y contraseña** | `password` | `PIREON_PASSWORD_CLIENT_ID`, `PIREON_PASSWORD_CLIENT_SECRET` | `POST /api/auth/token` (`password`); se desactiva con `PIREON_PASSWORD_GRANT=false` |

Si no se definen las variables `PIREON_PASSWORD_CLIENT_*`, el login con contraseña usa `PIREON_CLIENT_ID` (sirve si solo hay un cliente de tipo `password`, pero entonces «Entrar con Pireon» falla con `unauthorized_client`). Al renovar la sesión, la API prueba los dos clientes: un `refresh_token` solo vale con el cliente que lo emitió.

El email y la contraseña son los de la cuenta de Pireon; TaskStream no los guarda, solo los reenvía. (El login con token de acceso personal se ha retirado de la web por ahora; la API sigue aceptando esos tokens como `Bearer`.)

## Registrar TaskStream como cliente en Pireon

Con un token tuyo de Pireon (scope `account:write`), `POST https://oauth.pireon.dev/api/clients`:

1. **Cliente para «Entrar con Pireon»**: `{"name": "TaskStream", "grant_type": "authorization_code", "confidential": true, "redirect_uris": [...]}` con estas redirect URIs:
   - `http://localhost:4200/auth/callback` (desarrollo) y `http://localhost:8080/auth/callback` (Docker)
   - `https://taskstream.pireon.dev/auth/callback` (producción)
   - para el «Authorize» de Swagger UI: `http://localhost:3000/api/docs/oauth2-redirect.html` y `https://api.taskstream.pireon.dev/api/docs/oauth2-redirect.html` (ver API REST)

   → `PIREON_CLIENT_ID` y `PIREON_CLIENT_SECRET`. Al ser confidencial, **el secret es obligatorio**: sin él Pireon rechaza el canje del código (ver la tabla de errores más abajo).
2. **Cliente para email y contraseña**: `{"name": "TaskStream (password)", "grant_type": "password"}` → `PIREON_PASSWORD_CLIENT_ID` y `PIREON_PASSWORD_CLIENT_SECRET`.

Pireon muestra el secret (`plain_secret`) **una sola vez**, en la respuesta de alta. Si se pierde, se genera otro con `POST https://oauth.pireon.dev/api/clients/{client}/rotate-secret` (el anterior sigue valiendo 48 h).

Ponlos en `.env` (o en el entorno del contenedor; en Dokploy, en *Environment*) y reinicia la API; en Docker/Dokploy, **recrea** el contenedor (redesplegar), porque un reinicio no relee el entorno.

En producción:

| Servicio | URL | Redirect URIs registradas en el cliente `authorization_code` |
|---|---|---|
| Web | `https://taskstream.pireon.dev` | `https://taskstream.pireon.dev/auth/callback` |

| Servicio | URL | Redirect URIs registradas en el cliente `authorization_code` |
|---|---|---|
| API (REST) | `https://api.taskstream.pireon.dev` | `https://api.taskstream.pireon.dev/api/docs/oauth2-redirect.html` |

`GET /api/auth/config` le dice a la web qué formas de entrar hay: `clientId` del cliente `authorization_code` (o `null`: la web oculta «Entrar con Pireon») y `passwordGrant`. Devuelve `null` si no hay ningún cliente; en ese caso el login muestra el aviso y no deja entrar.

## Si el login falla

La API reenvía el error de Pireon con su `hint` entre paréntesis (lo que de verdad falla) y lo registra en su log como `Pireon rejected <grant> for client <id>: … (hint: …)`.

| Error | Causa | Solución |
|---|---|---|
| El login dice «Inicio de sesión no configurado» | `GET /api/auth/config` devuelve `null`: no hay ningún cliente en el entorno de la API | Define `PIREON_CLIENT_ID` y/o `PIREON_PASSWORD_CLIENT_ID` (con sus secrets) y recrea el contenedor |
| `unauthorized_client`: *not authorized to use this authorization grant type* | El cliente no es del tipo que pide el flujo (p. ej. uno `password` en `PIREON_CLIENT_ID`) | Un cliente `authorization_code` en `PIREON_CLIENT_ID` y uno `password` en `PIREON_PASSWORD_CLIENT_ID` |
| `invalid_request` … *(Check the `client_secret` parameter)* | El cliente es confidencial y la API no envía secret: la variable está vacía o no llega | Pon el secret en `PIREON_CLIENT_SECRET` (o `PIREON_PASSWORD_CLIENT_SECRET`) y redespliega |
| `invalid_client`: *Client authentication failed* | El secret llega pero no es el de ese cliente | Revisa que id y secret sean del mismo cliente; si no lo tienes, rótalo |
| `invalid_request` … *(Invalid redirect URI)* | La redirect URI del canje no es la de la autorización | Regístrala exactamente igual en el cliente (esquema, dominio y ruta, sin barra final) |
| `invalid_grant`: *Authorization code has expired / has been revoked* | El código caducó o ya se usó (p. ej. al recargar `/auth/callback`) | Vuelve a «Entrar con Pireon» |
| `The user credentials were incorrect.` | Email o contraseña de Pireon incorrectos | — |
| Errores `404 Cannot GET/POST /api/…` intermitentes a través de la web | Otro `api` en la red compartida de Dokploy recibe parte de las peticiones | Ya resuelto con alias únicos (`task-stream-api` …), ver Despliegue |

## Sesión en la web

- La sesión (`accessToken`, `refreshToken`, caducidad) se guarda en `localStorage` y se restaura al arrancar.
- Un minuto antes de caducar se renueva con `refresh_token`. Si llegan varias peticiones a la vez, se renueva una sola vez.
- Si la API responde `UNAUTHENTICATED`, se cierra la sesión y se vuelve al login.

- El parámetro `state` del flujo PKCE protege contra CSRF; el `code_verifier` vive en `sessionStorage` solo durante la redirección.

## 5.3 Webhooks de Pireon (opcional)

Para que la desactivación de un usuario o la revocación de un token surtan efecto al instante (sin esperar a que caduque la caché):

1. Registra `https://<tu-api>/api/webhooks/pireon` con los eventos `user.deactivated` y `token.revoked` (`POST /api/clients/{client}/webhooks`).
2. Pon el secret que devuelve en `PIREON_WEBHOOK_SECRET`.
3. Si usas un cliente aparte para el acceso con contraseña (`PIREON_PASSWORD_CLIENT_ID`), registra el webhook también en ese cliente y pon su secret en `PIREON_PASSWORD_WEBHOOK_SECRET`. Pireon firma cada aviso con el secret del cliente que lo registró, y la revocación de un token solo la avisa el cliente que lo emitió: sin este paso, los tokens de quien entra con contraseña no se olvidan al revocarlos.

La API verifica la firma `X-Webhook-Signature` (HMAC-SHA256 del cuerpo, válida con cualquiera de los dos secrets), marca al usuario como inactivo y borra sus sesiones cacheadas. Además guarda `UserDeactivatedEvent` en el event store, y una saga del tracker deja sin asignar sus tickets abiertos en todos los proyectos (ver Arquitectura). Repetir un aviso (llega uno por cliente) no tiene efecto. Sin ningún secret, el endpoint rechaza todas las llamadas.

## 5.4 Usuarios en TaskStream

- Al validar un token, el usuario se guarda (nombre, email, activo) en la colección `users` de MongoDB.
- `users(search)` sincroniza antes el directorio de Pireon con el token del usuario, así que se puede añadir a gente que aún no ha entrado en TaskStream (requiere el scope `users:read`).
- Los read models guardan el nombre de los usuarios desnormalizado: si alguien cambia su nombre en Pireon, las entradas antiguas mantienen el nombre anterior.

# 6. Desarrollo

## 6.1 Requisitos

- Node 22 y npm 10.
- Docker (PostgreSQL, MongoDB y Redis).
- Acceso al Nexus de Pireon para `@pi/ui`. El scope está en `.npmrc` (sin token); el token va en tu `~/.npmrc`:

  ```
  npm config set //nexus.pireon.dev/repository/npm-releases/:_authToken "<token>"
  ```

## 6.2 Puesta en marcha

```
npm install
docker compose up -d postgres mongo redis
cp .env.example .env            # ajusta PIREON_*
npx nx serve api                # http://localhost:3000 (GraphiQL en /graphql)
npx nx serve web                # http://localhost:4200
```

`nx serve web` usa `apps/web/proxy.conf.json` para enviar `/api` y `/graphql` (también el WebSocket) a la API, así que la web y la API comparten origen igual que en Docker.

## 6.3 Comandos

| Comando | Qué hace |
|---|---|
| `npx nx run-many -t lint` | Lint de todo, incluidas las reglas de capas y contextos |
| `npx nx run-many -t test` | Tests unitarios (dominio y aplicación, API y web) |
| `npx nx e2e api` | Flujo completo de la API contra Docker, con Pireon simulado |
| `npx nx run-many -t build` | Builds de la API y la web |
| `npx nx run web:codegen` | Regenera los tipos GraphQL del cliente desde `apps/api/schema.gql` |
| `npx nx run api:contracts` | Escribe `schema.gql` y `openapi.json` sin arrancar la API ni bases de datos |
| `npx nx serve mcp` | Servidor MCP en `http://localhost:3010/mcp`, contra la API local (MCP) |
| `npx nx run mcp:openapi-types` | Regenera los tipos REST del MCP desde `apps/api/openapi.json` |
| `npx nx run api:prune` | Build de la API con `package.json` y lockfile podados (lo usa Docker) |
| `npx nx graph` | Grafo de dependencias entre proyectos |

| Comando | Qué hace |
|---|---|
| `npx nx test api-tracker-domain` | Tests de una librería concreta ( `api-<contexto>-<capa>` , `web-<contexto>-<capa>` ) |

### Tests

| Dónde | Qué se prueba | Herramienta |
|---|---|---|
| `api/*/domain` | Reglas de los agregados y reconstrucción desde el historial de eventos | Jest |
| `api/*/application` | Commands contra un event store en memoria (incluida la concurrencia) | Jest + `@nestjs/testing` |
| `apps/api/e2e` | Todo el flujo: GraphQL → Postgres → Redis → Mongo → subscriptions | Jest contra Docker |
| `web/*/domain` , `web/*/application` | Sesión y reducers del tablero | Jest / Vitest |
| `apps/web` | Rutas y guards | Vitest |

El e2e sustituye a Pireon por usuarios fijos ( `token-ana` , `token-bob` ...) con `overrideProvider(IdentityProvider)` ; el resto de la pila es real.

## 6.4 Cómo añadir una funcionalidad

Ejemplo: **marcar un ticket como bloqueado con un motivo**. El recorrido es siempre el mismo, de dentro hacia fuera.

### API

1. **Dominio** ( `libs/api/tracker/domain` )
   - Evento `issue-blocked.event.ts` con `@AutowiredEvent` , que extiende `TrackerEvent` .
   - En `Issue` , el método `block(actorId, reason)` valida y hace `this.apply(new IssueBlockedEvent(…))` ; el handler `onIssueBlockedEvent(event)` cambia el estado.
   - Test en `issue.spec.ts` , incluida la reconstrucción desde el historial ( `replay` ).
2. **Aplicación** ( `libs/api/tracker/application` )
   - `commands/block-issue.command.ts` y `block-issue.command-handler.ts` : rehidratar ( `WriteAccess` ), llamar al dominio y `commitAndWait()` . Registrar el handler en `command-handlers.ts` .
   - Método en `IssuesService` .
   - Proyección `event-handlers/issue-blocked.event-handler.ts` ( `@EventsHandler` ) que actualiza el read model y avisa por `ReadModelNotifier` . Añadir el evento al `ActivityEventHandler` y una frase en `activity-summary.ts` .
   - Si el read model cambia, añadir el campo a `IssueReadModel` (dominio) y al esquema Mongo (infraestructura).

3. **Presenters** (`libs/api/tracker/presenters`): input y mutation `blockIssue` en `IssueResolver`, que llama al service.
4. Regenerar `apps/api/schema.gql` y `apps/api/openapi.json`: arrancando la API, o sin bases de datos con `npx nx run api:contracts`.

### Web

5. `npx nx run web:codegen` tras añadir la mutation a `issues.graphql`.
6. **Dominio**: método en el puerto `IssuesGateway`. **Infraestructura**: implementarlo en `GraphqlIssuesGateway`.
7. **Aplicación**: evento `blockRequested` en `issuePageEvents`, su efecto en `IssueStore` (`runCommand`) y el método `block(reason)` en `IssueFacade`.
8. **Presenters**: botón y formulario en la página del ticket que llaman a `facade.block(reason)`.

El lint impide atajos: si un resolver intenta usar un repositorio o una página intenta usar Apollo, falla. Y como `application` solo exporta facades, una página tampoco puede tocar un store ni despachar eventos por su cuenta.

## 6.5 Problemas conocidos

| Síntoma | Causa y solución |
|---|---|
| `npm install` falla con `Cannot read properties of null (reading 'edgesOut')` | Bug de npm con un peer opcional de Vite que arrastra vitest 5. El `overrides` de `package.json` fija vitest; no lo quites |
| `401` al instalar `@pi/ui` | Falta el token del Nexus en `~/.npmrc` (o `NEXUS_TOKEN` al construir con Docker) |
| `failed to stat /root/.npmrc` al desplegar | Versión antigua del `docker-compose.yml`: ahora el token llega por `NEXUS_TOKEN` (ver Despliegue) |
| `nx serve web` arranca en el puerto 3000 | Nx carga el `.env` raíz (`PORT=3000`) en todas las tareas; `apps/web/.env` lo fija en 4200 |
| La API no ve un evento nuevo al deserializar | La clase del evento no tiene `@AutowiredEvent` o no se importa |
| Un componente de `@pi/ui` lanza `Unable to find icon` | Falta registrar ese icono en `libs/web/shared/ui/src/lib/icons.ts` |
| Tablas antiguas (`projects`, `issues`, `outbox`) en tu Postgres local | Restos de una versión anterior sin event sourcing; `docker compose down -v` empieza de cero |
| El login con Pireon falla (`unauthorized_client`, `client_secret` ...) | Configuración de los clientes OAuth: ver Si el login falla |

# 7. Despliegue

## 7.1 Docker

```
docker compose up -d --build                # todo: web en http://localhost:8080
docker compose up -d postgres mongo redis   # solo infraestructura
docker compose logs -f api
docker compose down                         # parar (-v borra los datos)
```

| Servicio | Imagen | Puerto en local | Función |
|---|---|---|---|
| `web` | `task-stream-web` (nginx) | 8080 | SPA y proxy de `/api` y `/graphql` (incluido el WebSocket) hacia la API |
| `api` | `task-stream-api` (Node 22) | 3000 | API NestJS |
| `postgres` | postgres:17-alpine | 5432 | Event store y lookups |
| `mongo` | mongo:8 | 27017 | Read model |
| `redis` | redis:7-alpine (AOF) | 6379 | Stream de eventos, pub/sub y caché de sesiones |

### Producción y desarrollo

- **`docker-compose.yml`** es la definición de producción (la que despliega Dokploy) y **no publica puertos**: los servicios solo se ven entre sí por la red de Compose. Hacia fuera solo sale la web, por el proxy de la plataforma (en Dokploy: dominio → servicio `web`, puerto `80`). `web` y `api` declaran `expose` para documentarlo.
- **`docker-compose.override.yml`** publica los puertos de la tabla. Compose lo carga solo cuando se ejecuta sin `-f`, es decir, en local; Dokploy usa `-f docker-compose.yml` y no lo lee.

Para probar en local exactamente lo de producción: `docker compose -f docker-compose.yml up -d --build`.

### Nombres únicos en la red

Los servicios se llaman entre sí por **alias únicos** (`task-stream-api`, `task-stream-postgres`, `task-stream-mongo`, `task-stream-redis`), no por el nombre del servicio. Al dar dominio a un servicio, Dokploy lo conecta a la red compartida `dokploy-network`, donde otros proyectos también tienen un `api`, un `postgres` o un `redis`: con nombres genéricos, el DNS de Docker devuelve también los suyos y nginx reparte las peticiones entre ambos (síntoma: la mitad de las llamadas a `/api/*` responden `404 Cannot GET …` y el login dice «no configurado»). Si añades un servicio, dale su alias `task-stream-*` y úsalo en las URLs.

Todos tienen healthcheck; `web` espera a que `api` esté sana, y `api` a las tres bases de datos.

### Imágenes

El `Dockerfile` tiene tres destinos que comparten la instalación de dependencias:

- **Instalación**: `npm ci` del workspace. `@pi/ui` viene del Nexus privado; el token llega como **build secret** desde la variable `NEXUS_TOKEN` (`secrets.nexus_token` en `docker-compose.yml`), se escribe en un `npmrc` temporal y se borra en el mismo paso, así que nunca queda en una capa. Sin `NEXUS_TOKEN` la build falla en ese paso.
- `api`: `nx run api:prune` genera el bundle con un `package.json` y un lockfile podados a partir de `apps/api/package.json`. La imagen final solo instala esas dependencias (sin Angular ni paquetes privados). Si la API empieza a usar un paquete nuevo **en tiempo de ejecución**, hay que añadirlo a `apps/api/package.json`.
- `mcp`: el servidor MCP, podado igual que la API (`nx run mcp:prune`), sobre Node 24. Su servicio en `docker-compose.yml` va en el perfil `mcp`: solo arranca con `--profile mcp` (en Dokploy, `COMPOSE_PROFILES=mcp`).
- `web`: build de producción de Angular servido por nginx (`apps/web/nginx.conf`): fallback de rutas a `index.html`, caché larga para los bundles con hash y sin caché para `index.html`.

### Construir

`NEXUS_TOKEN` tiene que estar en el entorno de quien construye, o en el `.env` junto al `docker-compose.yml`:

```
# Local: reutiliza el token de tu ~/.npmrc
export NEXUS_TOKEN=$(grep '^//nexus.pireon.dev/repository/npm-releases/:_authToken=' ~/.npmrc |
cut -d= -f2-)
docker compose up -d --build
```

En **Dokploy** (despliegue con Docker Compose): pestaña *Environment* del servicio → `NEXUS_TOKEN=<token>`, junto a `PIREON_CLIENT_ID`, `PIREON_CLIENT_SECRET`, `PIREON_PASSWORD_CLIENT_ID`, `PIREON_PASSWORD_CLIENT_SECRET` y `PIREON_WEBHOOK_SECRET`. Dokploy los escribe en el `.env` del proyecto, de donde los lee Compose.

## 7.2 Variables de entorno de la API

| Variable | Por defecto | Descripción |
|---|---|---|
| `PORT` | `3000` | Puerto HTTP |
| `NODE_ENV` | `development` | En `production` no se devuelven stacktraces |
| `POSTGRES_URL` | — (obligatoria) | Event store y lookups |
| `POSTGRES_SYNCHRONIZE` | `false` | Crea las tablas desde las entidades. Cómodo en desarrollo; en producción, migraciones |
| `MONGO_URL` | — (obligatoria) | Read model |
| `REDIS_URL` | — (obligatoria) | Stream, pub/sub y caché |

| Variable | Por defecto | Descripción |
|---|---|---|
| `EVENT_STREAM_KEY` | `task-stream:events` | Clave del Redis Stream |
| `EVENT_STREAM_MAX_LENGTH` | `1000000` | Longitud aproximada que se conserva en el stream |
| `EVENTS_BRIDGE_ENABLED` | `true` | Si esta instancia proyecta eventos hacia Mongo (ver escalado) |
| `PIREON_BASE_URL` | `https://oauth.pireon.dev` | Servidor de identidad |
| `PIREON_CLIENT_ID` / `PIREON_CLIENT_SECRET` | — | Cliente OAuth `authorization_code`: «Entrar con Pireon» y Swagger UI |
| `PIREON_PASSWORD_CLIENT_ID` / `_SECRET` | el anterior | Cliente OAuth `password`: login con email y contraseña |
| `PIREON_SCOPES` | `users:read` | Scopes que se piden al iniciar sesión |
| `PIREON_PASSWORD_GRANT` | `true` | Permite el login con email y contraseña |
| `PIREON_WEBHOOK_SECRET` | — | Activa y firma los webhooks de Pireon |
| `PIREON_PASSWORD_WEBHOOK_SECRET` | — | El del webhook del cliente de contraseña, si es otro cliente |
| `PIREON_TIMEOUT_MS` | `5000` | Timeout de las llamadas a Pireon |
| `AUTH_SESSION_TTL_SECONDS` | `60` | Cuánto se confía en un token validado antes de volver a preguntar a Pireon |
| `GRAPHQL_PLAYGROUND` | `true` | GraphiQL e introspección |
| `GRAPHQL_SCHEMA_FILE` | `apps/api/schema.gql` | Dónde escribir el SDL (vacío = no escribir; así está en Docker) |
| `SWAGGER_ENABLED` | `true` | API REST documentada: Swagger UI en `/api/docs`, OpenAPI en `/api/docs-json` |
| `OPENAPI_FILE` | `apps/api/openapi.json` | Dónde escribir el documento OpenAPI al arrancar (vacío = no escribir; así en Docker) |
| `ATTACHMENTS_DIR` | `data/attachments` | Carpeta de los adjuntos de los tickets; en Docker, el volumen `attachments-data` |
| `SMTP_URL` | — | Servidor de correo para las copias de las notificaciones (`smtps://user:pass@host`); sin él no se envía nada |
| `MAIL_FROM` | `TaskStream <no-reply@pireon.dev>` | Remitente de esos correos |
| `WEB_URL` | — | Dirección pública de la web, para los enlaces de los correos |

| Variable | Por defecto | Descripción |
|---|---|---|
| `ATTACHMENTS_MAX_MB` | `25` | Tamaño máximo de un adjunto (nginx admite hasta 30 MB por petición) |
| `WEBHOOKS_ALLOW_PRIVATE` | `false` | Deja que los webhooks salientes llamen a direcciones de esta máquina o de su red privada (apagado, un webhook podría alcanzar servicios internos) |
| `REMINDERS_ENABLED` | `true` | Avisa cada hora de los tickets que vencen mañana, hoy o ya han vencido |
| `REMINDERS_TIMEZONE` | `Europe/Madrid` | Zona horaria en la que cambia el día para los recordatorios |
| `TRASH_DAYS` | `30` | Días que un ticket pasa en la papelera antes de borrarse del todo |

La web no tiene configuración de entorno: llama a la API en su mismo origen (`/api`, `/graphql`).

## 7.3 Escalado

- **API**: es *stateless* (sesiones en Redis, subscriptions por Redis pub/sub), así que admite varias réplicas detrás del balanceador. El WebSocket necesita que el balanceador permita `Upgrade`. La excepción son los **adjuntos**: se guardan en disco (`ATTACHMENTS_DIR`), así que varias réplicas tienen que compartir ese volumen.
- **Relay**: cada instancia publica eventos al stream; `FOR UPDATE SKIP LOCKED` evita duplicados.
- **Proyecciones**: el `EventsBridge` procesa los eventos en orden. Lo más sencillo es dejarlo activo en **una sola** instancia (`EVENTS_BRIDGE_ENABLED=true`) y desactivarlo en las demás. Varias instancias en el mismo consumer group se reparten los eventos y pierden el orden entre ellos.

## 7.4 Antes de producción

- Sustituir `POSTGRES_SYNCHRONIZE` por migraciones de TypeORM.
- Registrar el cliente OAuth y las redirect URIs de producción en Pireon.
- Servir por HTTPS (la web usa `wss://` automáticamente cuando la página va por HTTPS).
- Revisar `GRAPHQL_PLAYGROUND` (introspección) según el entorno.
- Copias de seguridad de PostgreSQL: es la fuente de verdad. Y del volumen de adjuntos (`attachments-data`): los eventos dicen qué archivos hay, pero los bytes solo están ahí.

## 7.5 Reconstruir el read model

MongoDB solo contiene proyecciones, así que se puede regenerar desde el event store:

1. Para la API (o desactiva el bridge en todas las instancias).
2. Borra la base de lectura (`taskstream_read`) salvo la colección `users`, que no sale de los eventos sino de Pireon.
3. Marca los eventos como no publicados: `UPDATE events SET published_at = NULL;`

4. Arranca la API: el relay vuelve a enviar todos los eventos, en orden, y las proyecciones reconstruyen las vistas. Las notificaciones en tiempo real se reenvían también, sin efecto si no hay clientes conectados.

Hazlo también una vez tras desplegar la configuración por proyecto y las estadísticas si quieres la historia completa de los tickets anteriores: sin reconstruir, la línea temporal (`issue_timeline`) solo empieza con los cambios nuevos y las estadísticas usan, para lo anterior, la creación y el estado actual de cada ticket. La categoría de los estados no lo necesita: la API la rellena al arrancar en los tickets que no la tienen.

# 8. Decisiones de diseño

Registro breve de las decisiones que más condicionan el código, con su motivo y su coste.

## Arquitectura hexagonal con una librería Nx por capa

Las reglas de dependencia las comprueba el lint, no la disciplina del equipo. **Coste**: más librerías y más ficheros (un command y un handler por fichero, como en el patrón de referencia del equipo).

## La misma estructura en API y web

`libs/api/<contexto>/<capa>` y `libs/web/<contexto>/<capa>`: quien conoce un lado se orienta en el otro.

## Event sourcing en PostgreSQL (y no en MongoDB)

El requisito era escritura en PostgreSQL y lectura en MongoDB. El event store (`events`) implementa la misma API que el `MongoEventStore` del patrón de referencia (`persist`, `getEventsByStreamId`). **Diferencia**: un conflicto de versión lanza `ConcurrencyError` en lugar de solo registrarse, para no perder cambios en silencio.

### `commitAndWait()`

El `commit()` de `@nestjs/cqrs` no devuelve la promesa del publisher: el command respondería OK aunque el event store fallara. `VersionedAggregateRoot.commitAndWait()` espera la escritura y avanza la versión del agregado.

## Redis entre escritura y lectura, y un bridge que espera

El requisito era que Redis llevase los datos de escritura a lectura. El `EventsBridge` hace el papel del *change stream* del patrón de referencia, pero no usa `eventBus.subject$.next()`: así los handlers se ejecutarían en paralelo (`mergeMap`), un `IssueStatusChanged` podría proyectarse antes que su `IssueCreated` y un fallo se perdería. El bridge espera a cada `@EventsHandler` en orden y confirma en Redis (`XACK`) al final.

## `EventClsRegistry` y `@AutowiredEvent` en el dominio

En el patrón de referencia están en infraestructura, pero los eventos de dominio los usan; ponerlos allí rompería la regla de capas.

## Lookups fuera de los agregados

La unicidad de claves de proyecto, la numeración de tickets y los nombres de release no caben en un agregado. Se resuelven con tablas pequeñas en PostgreSQL. **Coste**: no forman parte de la misma transacción que los eventos (una clave reservada puede quedar huérfana si el guardado falla después).

## Reglas críticas contra el event store, no contra Mongo

"No publicar una release con tickets sin terminar" se comprobaba primero contra el read model y fallaba si alguien metía un ticket y publicaba justo después. Ahora reconstruye esos tickets desde los eventos.

## Proxy de tokens en la API

Pireon no permite CORS en `/oauth/token` y el `client_secret` no debe llegar al navegador. `POST /api/auth/token` hace de intermediario, igual que la API de administración de la landing.

## Caché de Apollo desactivada en la web

El estado vive en los stores de NgRx (alimentados por queries y subscriptions). Tener dos fuentes de verdad (Apollo cache y stores) solo añadiría inconsistencias.

## Estado web con el plugin de eventos de NgRx Signals

Los efectos se declaran con `withEventHandlers` (nombre en v22 de lo que antes era `withEffects`), y lo que emiten se despacha automáticamente.

## Facades delante de los stores

Las páginas usan una facade por store (`BoardFacade`, `IssueFacade`...) y no conocen ni los stores ni los eventos, que ya no se exportan. La facade expone el estado como señales y las intenciones como métodos, y pone el contexto (id del proyecto o del ticket abierto). Así se puede cambiar cómo se gestiona el estado sin tocar las páginas, y la lógica de presentación se prueba sin plantillas. **Coste**: una clase más por store, casi toda de reenvío.

## Imagen de la API con dependencias propias

El `package.json` raíz incluye Angular y `@pi/ui` (del Nexus privado). La imagen de la API usa `apps/api/package.json`, así que ni lleva el frontend ni necesita el token del Nexus en ejecución.

## Adjuntos en disco, notificaciones fuera del event store

Los **adjuntos** se registran como eventos del ticket, pero los bytes van a un puerto `AttachmentStorage` (disco, `ATTACHMENTS_DIR`): un event store no es sitio para megabytes. Se escribe el archivo antes que el evento, y si el evento falla se borra el archivo, así que nunca hay un adjunto apuntando a nada. **Coste**: el volumen hay que copiarlo y compartirlo entre réplicas.

Las **notificaciones** sí salen de eventos, pero no son eventos: si un aviso está leído, o si alguien quiere copia por correo, solo le importa a esa persona y no necesita historia. Se guardan en MongoDB por destinatario y se marcan como leídas sin pasar por un agregado. Si se reconstruye el read model, se pueden regenerar desde los eventos (quedarían como no leídas). **Alternativa descartada**: un agregado «bandeja» por persona, con eventos `NotificationRead`; mucha ceremonia para un estado que nadie audita.

# 9. Servidor MCP

`apps/mcp` es un servidor MCP (Model Context Protocol) para que asistentes de IA (Claude Code, Claude Desktop, Cursor...) trabajen con TaskStream: consultar tableros y tickets, crear y mover tickets, comentar, gestionar releases o leer las estadísticas. Está hecho con NestJS y `@nestjs-mcp/server` 2.x, y consume la **API REST** como cualquier otra app: con tipos generados desde `apps/api/openapi.json`.

## 9.1 Cómo funciona

- **Un endpoint**: `POST /mcp`, transporte *Streamable HTTP*. Sin estado (la versión 2026-07-28 del protocolo retiró las sesiones): cualquier instancia atiende cualquier llamada.
- **Sin credenciales propias**: cada llamada lleva su `Authorization: Bearer <token de Pireon>`, y el MCP la reenvía tal cual a la API. Cada persona ve y puede cambiar exactamente lo que la API le permite (su rol en cada proyecto); **sin token**, solo se leen los proyectos públicos.
- **Errores legibles**: si la API rechaza algo (permiso, regla de negocio, transición no permitida...), la herramienta responde con un error que el modelo puede leer y corregir («Cannot move WEB-16 from Backlog to Hecho. Allowed: Por hacer, En curso, Cancelado»), en vez de cortar la llamada.
- **Validación**: los parámetros de cada herramienta se validan con zod antes de llamar a la API.
- **Solo en su dominio**: `/mcp` rechaza (403) las peticiones cuyo `Host` no esté en `MCP_ALLOWED_HOSTS`, y las de un navegador cuyo `Origin` tampoco lo esté. Es la protección contra *DNS rebinding* que pide la especificación de MCP; los clientes que llaman desde un servidor no envían `Origin` y solo necesitan el dominio correcto.
- **Respuestas compactas**: los listados devuelven lo que el modelo necesita (clave, título, estado, persona...); el detalle completo, con `get_issue` o `get_project`.

## 9.2 Herramientas

57 herramientas, agrupadas por ámbito:

| Ámbito | Herramienta | Qué hace |
|---|---|---|
| Usuario | `whoami` | El usuario del token |
| | `search_users` | Buscar personas de Pireon (para asignar) |
| | `my_notifications` | Mis notificaciones (por defecto, las no leídas) |
| | `mark_notifications_read` | Marcar como leídas unas o todas |
| Proyectos | `list_my_projects` | Mis proyectos |
| | `list_public_projects` | Proyectos públicos |

| Ámbito | Herramienta | Qué hace |
|---|---|---|
| | `create_project` | Crear un proyecto (quien lo crea es su dueño) |
| | `get_project` | Equipo y configuración: ids de estados, tipos y áreas que aceptan las demás herramientas |
| | `get_project_showcase` | El escaparate público: hoja de ruta, releases, sprint en curso y tickets para empezar |
| | `get_board` | Tablero por columnas (filtros por persona, release, área, sprint) |
| | `get_project_stats` | Cifras de seguimiento |
| | `get_project_analytics` | Estadísticas de los últimos días (creados/resueltos, flujo, tiempos, throughput, QA) |
| | `get_project_activity` | Actividad reciente |
| Tickets | `my_issues` | Mi trabajo en todos los proyectos (o de cualquiera, por categoría; o los que sigo) |
| | `search_issues` | Buscar por texto y filtros (épica, sprint, fechas...); el backlog: `noSprint` y orden `RANK` |
| | `get_issue` | Detalle: épica, lista de comprobación, relaciones, commits, seguidores, transiciones, comentarios |
| | `create_issue` | Crear un ticket (también colgado de una épica) |
| | `update_issue` | Cambiar título, descripción, tipo, área, prioridad, etiquetas, fechas... |
| | `change_issue_status` | Mover de estado (con motivo cuando el estado lo pide) |
| | `assign_issue` | Asignar o desasignar |
| | `set_issue_release` | Meter en una release o sacar |
| | `add_comment` / `edit_comment` / `delete_comment` | Comentar (o responder a un comentario), corregir o borrar |
| | `link_issues` / `unlink_issues` | Relacionar tickets (`DEPENDS_ON`, `BLOCKS`, `TESTS`...) o quitar la relación |
| | `watch_issue` | Seguir un ticket o dejar de seguirlo |
| | `vote_issue` | Votar un ticket o quitar el voto |
| | `duplicate_issue` | Crear otro ticket con lo mismo (sin estado, persona ni comentarios) |
| | `react_to_comment` | Reaccionar a un comentario (👍 🎉 ...) o quitar la reacción |

| Ámbito | Herramienta | Qué hace |
|---|---|---|
| | `add_checklist_item` / `update_checklist_item` / `remove_checklist_item` | La lista de comprobación: añadir, marcar o reescribir, quitar |
| Organización | `list_epics` | Épicas con el progreso de lo que cuelga de cada una |
| | `set_issue_parent` | Colgar un ticket de una épica u otro ticket, o soltarlo |
| | `rank_issue` | Mover un ticket en el backlog |
| | `bulk_update_issues` | El mismo cambio en hasta 100 tickets (por clave); dice cuáles no se dejaron |
| | `delete_issue` / `restore_issue` / `list_trash` | Papelera: borrar (30 días para recuperar), recuperar y ver |
| Sprints | `list_sprints` | Sprints de un proyecto |
| | `get_sprint` | Progreso, tickets y burndown |
| | `create_sprint` / `update_sprint` / `delete_sprint` | Planificar, cambiar o borrar (si no empezó) un sprint |
| | `start_sprint` / `close_sprint` | Empezarlo; cerrarlo pasando lo pendiente a otro sprint o al backlog |
| | `set_issue_sprint` | Meter un ticket en un sprint o devolverlo al backlog |
| Releases | `list_releases` | Releases de un proyecto |
| | `upcoming_releases` | Próximas releases de mis proyectos |
| | `get_release` | Progreso, notas, tickets y burnup |
| | `create_release` / `update_release` | Planificar una release o cambiar su nombre, descripción o fecha |
| | `draft_release_notes` / `set_release_notes` | Borrador de las notas desde los tickets terminados, y guardarlas |
| | `publish_release` | Publicar (con `force` si quedan tickets sin terminar) |
| | `archive_release` | Archivar una release que no saldrá (sus tickets la dejan) |

Las de lectura van marcadas como `readOnlyHint`, para que el cliente no pida confirmación al usarlas, y las que borran o archivan, como `destructiveHint`, para que la pida. Lo que es de administración (miembros, configuración, visibilidad, borrar proyectos, webhooks, integración con git, importar y exportar CSV) no se expone: se hace desde la web.

## 9.3 Conectar un cliente

Hace falta un token de Pireon. Con email y contraseña (si la API tiene configurado el cliente `password`):

```
curl -s https://api.taskstream.pireon.dev/api/auth/token \
  -H 'content-type: application/json' \
  -d '{"grant_type":"password","username":"tu@email.com","password":"…"}'
# → { "access_token": "…", "refresh_token": "…", "expires_in": … }
```

Claude Code:

```
claude mcp add --transport http taskstream https://mcp.taskstream.pireon.dev/mcp \
  --header "Authorization: Bearer $TASKSTREAM_TOKEN"
# en local: http://localhost:3010/mcp
```

Otros clientes (JSON de configuración):

```
{
  "mcpServers": {
    "taskstream": {
      "type": "http",
      "url": "https://mcp.taskstream.pireon.dev/mcp",
      "headers": { "Authorization": "Bearer <token de Pireon>" }
    }
  }
}
```

El token caduca como cualquier sesión de Pireon: cuando la herramienta responda «Error 401», hay que pedir otro (o renovarlo con `grant_type: refresh_token`). El MCP todavía no implementa el flujo de autorización OAuth del propio protocolo MCP, así que el cliente no lo obtiene solo.

## 9.4 Desarrollo

```
npx nx serve api                    # la API en :3000
npx nx serve mcp                    # el MCP en http://localhost:3010/mcp
(TASKSTREAM_API_URL=http://localhost:3000)
npx @modelcontextprotocol/inspector  # probar las herramientas en el navegador
npx nx test mcp                     # MCP real contra una API simulada: token reenviado,
errores, validación
npx nx run mcp:openapi-types        # regenerar los tipos tras cambiar la API
(apps/api/openapi.json)
```

| Variable | Por defecto | Qué es |
|---|---|---|
| `MCP_PORT` | `3010` | Puerto del MCP (no `PORT`: Nx carga el `.env` del workspace, donde `PORT` es el de la API) |

| Variable | Por defecto | Qué es |
|---|---|---|
| `TASKSTREAM_API_URL` | `http://localhost:3000` | Origen de la API (sus rutas ya llevan `/api`) |
| `MCP_ALLOWED_HOSTS` | (solo `localhost`) | Dominios en los que responde, separados por comas |
| `MCP_LOG_LEVEL` | `warn` | `error`, `warn`, `log`, `debug` o `verbose`. Por encima de `warn`, la librería registra cada herramienta en cada llamada |

Añadir una herramienta: un método con `@Tool({ name, description, paramsSchema: z.object(…), annotations })` en el resolver de su ámbito (`apps/mcp/src/app/tools`), que llame a la API con `this.api.as(ctx)` (cliente tipado con el token de quien llama) y devuelva `toolResult(…)`.

La imagen solo instala lo que lista `apps/mcp/package.json`: si el MCP empieza a usar un paquete nuevo en tiempo de ejecución, hay que añadirlo ahí (igual que en la API).

## 9.5 Despliegue

El servicio `mcp` de `docker-compose.yml` va en el perfil `mcp`, así que no arranca salvo que se pida:

```
docker compose --profile mcp up -d --build   # local: http://localhost:3010/mcp
```

En Dokploy: `COMPOSE_PROFILES=mcp` en las variables del proyecto y un dominio (p. ej. `mcp.taskstream.pireon.dev`) → servicio `mcp`, puerto 3010. Si el dominio es otro, ponlo en `MCP_ALLOWED_HOSTS` (por defecto `mcp.taskstream.pireon.dev,localhost`); si no, el MCP responderá 403. Habla con la API por la red interna (`http://task-stream-api:3000`). La imagen usa Node 24, que es lo que pide `@nestjs-mcp/server` 2.

# 10. Lo que queda

Lo pendiente, ordenado de más a menos urgente: primero lo que hace falta para dar el producto por terminado en producción, después los fallos y la deuda conocidos y al final las ideas. Cada punto dice qué es, por qué importa y, cuando hay pistas, por dónde empezar.

## 10.1 Para dar el producto por terminado

| Qué | Por qué | Por dónde empezar |
|---|---|---|
| Comprobar en producción que el tiempo real ya no se pierde | Ya hay *ping* cada 10 s, reconexión si no hay respuesta y recarga de cada pantalla al reconectar, pero no se ha probado allí | Dejar una pestaña abierta más que la vida del token y tras suspender el portátil, y mover tickets desde otra. Si aún falla, mirar el token de `connectionParams`, que solo se manda al conectar |
| Pruebas de navegador en la integración continua | Los fallos del tablero (no se podía arrastrar entre columnas) y de la ficha (no se refrescaba) solo se vieron usándolo | Ya existe la receta: API real con usuarios simulados (`IdentityProvider` sustituido), `nx serve web` y Playwright con la sesión en `localStorage` (`task-stream.session`). Convertirla en `apps/web-e2e` |
| Copias de seguridad de Postgres y Mongo | Postgres guarda el event store, la fuente de verdad; Mongo se puede reconstruir desde él, Postgres no | Copias programadas en Dokploy y **una restauración de prueba** |
| Limpiar los datos de prueba de producción | DEMO, TEST y los tickets de las pruebas del MCP | Borrar los proyectos desde Configuración (borrado definitivo con su clave) |
| Registrar el webhook del cliente de contraseña en Pireon | Sin él, revocar un token de quien entra con contraseña no se nota hasta que caduca la caché | Solo si `PIREON_PASSWORD_CLIENT_ID` tiene valor: webhook en ese cliente y su secret en `PIREON_PASSWORD_WEBHOOK_SECRET` (ver Autenticación) |
| Una versión 1.0 | Un punto fijo al que volver | Desplegar todo, etiquetar `v1.0.0` y redactar la lista de cambios (las notas de versión de TaskStream dan el borrador) |
| Revisión de seguridad | Hay rutas sin sesión (proyectos públicos, escaparate, propuestas) y un MCP abierto a Internet | Permisos por rol, qué ve un anónimo, límites de peticiones (solo las propuestas tienen), CORS, cabeceras |
| Registro de errores en producción | Hoy un fallo solo se conoce si alguien avisa | Sentry (o similar) en la API, la web y el MCP |

## 10.2 Fallos y deuda conocidos

- **Error al crear subtareas en producción**: lo reportó el usuario y no se reprodujo en local; con el arreglo de la ficha la lista ya se refresca, pero falta el mensaje exacto si vuelve a salir.
- **Imágenes pegadas**: ya no salen en la lista de adjuntos, pero si se borran del texto el adjunto sigue guardado.
- **Seguimiento**: los tickets que ya existían no se apuntan a quien los creó o los tiene asignados (solo los nuevos).
- **El MCP no hace el flujo OAuth del protocolo**: el cliente tiene que llevar un token de Pireon en la cabecera y renovarlo a mano cuando caduca.

## 10.3 Siguientes funcionalidades

En orden de lo que más aportaría:

1. **Asistente en la web** (la base ya está en el MCP):
   - resumir un ticket largo con sus comentarios;
   - al crear un ticket, sugerir tipo, prioridad y área, y avisar si se parece a otro que ya existe;
   - redactar las notas de versión en lenguaje natural a partir del borrador.
2. **Avisos a Discord, Slack o Telegram**: los webhooks salientes mandan JSON en bruto, que esos chats no muestran bien. Una integración por canal (o un formato por destino en los webhooks) mandaría mensajes ya legibles: «Ana movió WEB-12 a En revisión», con enlace.
3. **Conector de claude.ai**: el MCP necesitaría el flujo OAuth del protocolo, y Pireon publicar su metadata de autorización y admitir el registro dinámico de clientes.

## 10.4 Descartado por ahora

Mucho trabajo para lo que aportan hoy; se pueden recuperar si alguien los necesita:

- **Registro de horas** dedicadas a cada ticket.
- **Objetivos u OKR** por encima de las épicas.
- **App móvil**: la web ya se adapta a pantallas estrechas.

## 10.5 Hecho desde la primera versión de esta lista

- Tiempo real: *ping* y reconexión del WebSocket, y recarga de lo que se ve al reconectar.
- Arrastrar a una columna prohibida devuelve la tarjeta a su sitio.
- Reacciones y «Bloqueado» en componentes propios (los estilos de la ficha vuelven a caber en 8 kB).
- Límite de propuestas en Redis, válido con varias instancias.
- Aviso `PROPOSED` a propietarios y admins cuando llega una propuesta.
- `voted` también por REST.
- Borrar un proyecto borra su integración con Git.
- Sin avisos de lint.

- La web pública (landing-web) enseña las propuestas más votadas, el enlace al formulario y los votos de la hoja de ruta.
- Un proyecto de la landing creado sin tablero puede abrirlo después: «Crear en TaskStream» en su ficha o `taskstream: true` en un `PATCH`.
- Paquete inicial de la web de 2,13 MB a 1,88 MB con `@pi/ui` 0.0.4, que ya no registra Chart.js al importarse ni depende de `lodash`.

# Anexo A. Librerías

Resumen de cada librería de `libs/api` y `libs/web` (procede de su `README.md`). Comandos de cada una: `npx nx lint <nombre>` y `npx nx test <nombre>`.

## api-identity-application

Casos de uso de identidad.

- **Import**: `@task-stream/api/identity/application`
- **Capa**: `application` · **Tags**: `scope:api-identity`, `type:application`

- `IdentityService`: punto de entrada para los presenters.
- Queries: `ResolveSessionQuery` (token → usuario, con caché), `FindUserQuery`, `FindUsersByIdsQuery`, `SearchUsersQuery`, `GetAuthorizationConfigQuery`.
- Commands: `IssueTokenCommand` (login / refresh vía Pireon), `HandleIdentityWebhookCommand` (al desactivar a un usuario, guarda `UserDeactivatedEvent` en el event store para que el tracker reaccione).
- Puertos: `IdentityProvider`, `SessionCache`, `UserRegistry`.

## api-identity-domain

El usuario, tal como lo conoce TaskStream (sus datos vienen de Pireon).

- **Import**: `@task-stream/api/identity/domain`
- **Capa**: `domain` · **Tags**: `scope:api-identity`, `type:domain`

- `User` (agregado): nombre, email, activo, roles globales y roles en el cliente; `hasAppRole()`, `deactivate()`.
- `UserFactory`: construye usuarios a partir de Pireon o de la caché.
- Tipos del inicio de sesión: `TokenGrant`, `TokenSet`, `AuthorizationConfig`.

## api-identity-infrastructure

Adaptadores de salida de identidad.

- **Import**: `@task-stream/api/identity/infrastructure`
- **Capa**: `infrastructure` · **Tags**: `scope:api-identity`, `type:infrastructure`

- `PireonIdentityProvider`: `GET /api/user`, `/api/users` y `POST /oauth/token` de Pireon.
- `RedisSessionCache`: tokens validados, guardados por hash SHA-256 y con TTL.
- `MongoUserRegistry`: copia local de usuarios (colección `users`).

## api-identity-presenters

Adaptadores de entrada de identidad.

- **Import**: `@task-stream/api/identity/presenters`
- **Capa**: `presenters` · **Tags**: `scope:api-identity`, `type:presenters`

- `auth/`: `AuthGuard` global, `@Public()`, `@OptionalAuth()` / `@ApiOptionalPireonAuth()` (lecturas de proyectos públicos: sin token, el usuario es `undefined`), `@CurrentUser()`, `@AccessToken()`, `@RequireAppRoles()`, `WsAuthenticator` (subscriptions) y `@ApiPireonAuth()` (esquemas OAuth2 / Bearer de Pireon en OpenAPI).
- `http/`: `AuthController` (`GET /api/auth/config`, `POST /api/auth/token`), `UsersController` (`/api/users/me`, directorio de Pireon) y `PireonWebhookController` (firma HMAC). Documentados con `@nestjs/swagger`.
- `graphql/`: queries `me`, `user` y `users`.

## api-shared-application

Contrato del event store y reconstrucción de agregados.

- **Import**: `@task-stream/api/shared/application`
- **Capa**: `application` · **Tags**: `scope:api-shared`, `type:application`

- `EventStore` (puerto): `persist` y `getEventsByStreamId`.
- `AggregateRehydrator`: carga los eventos de un stream y devuelve el agregado reconstruido, ya enlazado con el `EventPublisher`.
- `EventRecorder`: guarda un evento en un stream que no es de ningún agregado propio (p. ej. un usuario de Pireon), en la siguiente posición, para que otros contextos reaccionen a él.
- `SharedModule.withInfrastructure(...)`: módulo global que expone el event store, el rehydrator y el recorder a todos los contextos.

## api-shared-domain

Piezas de dominio comunes a todos los contextos de la API.

- **Import**: `@task-stream/api/shared/domain`
- **Capa**: `domain` · **Tags**: `scope:api-shared`, `type:domain`

- `VersionedAggregateRoot`: extiende el `AggregateRoot` de `@nestjs/cqrs` con la versión (`loadFromHistory`, `commitAndWait`, `trackCommit`).
- `Version`, `SerializableEvent`, `DomainEvent` (actor, fecha y `aggregateVersion`), `newId()`.
- `EventClsRegistry` y `@AutowiredEvent`: registro de las clases de evento para deserializarlas.
- Errores de dominio: `NotFoundError`, `ForbiddenError`, `ValidationError`, `BusinessRuleError`, `ConcurrencyError` y `assertThat`.

## api-shared-infrastructure

Event store en PostgreSQL y el camino de los eventos hasta la lectura.

- **Import**: `@task-stream/api/shared/infrastructure`
- **Capa**: `infrastructure` · **Tags**: `scope:api-shared`, `type:infrastructure`
- `PostgresEventStore` (tabla `events`, `UNIQUE(stream_id, position)` → `ConcurrencyError`).
- `EventSerializer` / `EventDeserializer` y `EventStorePublisher` (sustituye al publisher del `EventBus`: `commit()` persiste).
- `EventStoreRelay`: copia los eventos nuevos al Redis Stream (`FOR UPDATE SKIP LOCKED`).
- `EventsBridge`: consume el stream y entrega cada evento a sus `@EventsHandler`, en orden y esperando (ACK al final); después ejecuta los `@Saga()` (el `EventBus` en memoria nunca ve los eventos guardados) y sus comandos, uno a uno.
- Conexiones Redis (`REDIS_CLIENT`, `GRAPHQL_PUB_SUB`) y `SharedInfrastructureModule.forRootAsync`.

## api-shared-presenters

Adaptadores de entrada comunes.

- **Import**: `@task-stream/api/shared/presenters`
- **Capa**: `presenters` · **Tags**: `scope:api-shared`, `type:presenters`
- `DomainErrorFilter`: convierte los errores de dominio en errores GraphQL (`extensions.code`) o respuestas HTTP.
- `ApiErrorDto`, `@ApiReadErrors()` y `@ApiCommandErrors()`: el formato de error de la API REST en OpenAPI.

## api-tracker-application

Casos de uso del tracker (CQRS).

- **Import**: `@task-stream/api/tracker/application`
- **Capa**: `application` · **Tags**: `scope:api-tracker`, `type:application`
- Services: `ProjectsService`, `IssuesService`, `CommentsService`, `ReleasesService`, `ChangesService` (subscriptions), `WorkflowsService` (workflow de cada proyecto para presentar tickets, con una caché de 2 s).
- `commands/`: un fichero por command y otro por handler; cargan con `WriteAccess` (rehydrator) y guardan con `commitAndWait()`.
- `queries/`: un fichero por query y otro por handler; leen de los puertos `Find*Repository` y comprueban el acceso (`ReadAccess`: miembros, o cualquiera si el proyecto es público; `viewerId` null = anónimo); `SearchMyIssuesQuery` y `GetUpcomingReleasesQuery` cruzan todos los proyectos del usuario («Mi trabajo»).
- `event-handlers/`: proyecciones (`@EventsHandler`) hacia el read model (incluida la línea temporal de cada ticket), notificaciones en tiempo real y feed de actividad.
- `sagas/`: `ProjectDeletionSaga` (cascada al borrar un proyecto y sus tickets), `ReleaseArchivedSaga` (los tickets sin terminar salen de una release archivada) y `UserDeactivatedSaga` (desasigna los tickets abiertos de un usuario que Pireon desactiva). Las ejecuta el `EventsBridge`; sus comandos son internos e idempotentes. Tests en `sagas.spec.ts`.
- `analytics/`: estadísticas y burnup calculados con funciones puras sobre la línea temporal (con tests).
- `ports/`: lookups de escritura, `Find*`, `UpsertMaterialized*`, `ReleaseIssuesRepository`, `ChangeFeed`, `UserDirectory`.
- `TrackerApplicationModule.withInfrastructure(...)`.

## api-tracker-domain

El modelo del producto: proyectos, tickets, comentarios y releases.

- **Import**: `@task-stream/api/tracker/domain`
- **Capa**: `domain` · **Tags**: `scope:api-tracker`, `type:domain`
- Agregados event-sourced: `Project`, `Issue`, `Comment`, `Release`. Validan en sus métodos y cambian de estado en `on<Evento>`.
- `events/`: un fichero por evento (`IssueCreatedEvent`, `IssueStatusChangedEvent`...), todos con `@AutowiredEvent`.
- Proyectos públicos (`Project.isPublic`) y borrado: `Project.delete` (propietario, confirmando la clave), `Issue.delete`, `Release.delete` (los usa la cascada); `SYSTEM_ACTOR_ID` para cambios que no hace una persona.
- `factories/`: `ProjectFactory`, `IssueFactory`, `CommentFactory`, `ReleaseFactory` (`@Injectable()`).
- `project/project-settings.ts`: configuración por proyecto (estados con categoría, transiciones, tipos, áreas), sus valores por defecto, `ProjectWorkflow` (lo que consultan tickets y queries) y `validateProjectSettings` (archivar, nunca borrar).
- `read-models/`: forma de las vistas de lectura (`ProjectReadModel`, `IssueReadModel`...), incluida la línea temporal de los tickets y las estadísticas (`analytics.read-model.ts`).

## api-tracker-infrastructure

Adaptadores de salida del tracker.

- **Import:** `@task-stream/api/tracker/infrastructure`
- **Capa:** `infrastructure` · **Tags:** `scope:api-tracker`, `type:infrastructure`
- `postgres/`: lookups de escritura (`project_keys`, `issue_sequences`, `release_names`), `ReleaseIssuesRepository` y `TrackerStreamsRepository` (consultan el event store: tickets y releases de un proyecto, comentarios y enlaces de un ticket).
- `mongo/`: esquemas del read model (incluida la línea temporal `issue_timeline`), repositorios `Find*` y `UpsertMaterialized*` (upsert idempotente por versión) y `StatusCategoryBackfill` (al arrancar, rellena la categoría del estado en los tickets proyectados antes de existir).
- `realtime/`: `RedisChangeFeed` (canales de Redis pub/sub por proyecto, ticket y usuario).
- `identity/`: `IdentityUserDirectory`, capa anticorrupción hacia el contexto de identidad.

## api-tracker-presenters

API GraphQL y REST (OpenAPI) del tracker.

- **Import:** `@task-stream/api/tracker/presenters`
- **Capa:** `presenters` · **Tags:** `scope:api-tracker`, `type:presenters`
- Resolvers `ProjectResolver`, `IssueResolver`, `CommentResolver`, `ReleaseResolver` y `SubscriptionResolver`; solo llaman a services.
- Tipos, inputs (con `class-validator`) y enums de GraphQL (code-first).
- `http/`: controladores REST `ProjectsController`, `IssuesController`, `CommentsController`, `ReleasesController` (mismos services), DTOs con `@nestjs/swagger` y `Refs` (proyectos y tickets por id o clave; añade a cada ticket sus transiciones permitidas según el workflow de su proyecto).
- `ExploreController` (`GET /api/explore/projects`, sin token). Las lecturas usan `@OptionalAuth()` y `viewerOf(me)` (null = anónimo) para servir proyectos públicos sin sesión.
- `common/settings.ts`: configuración del proyecto con las transiciones como lista (lo que viaja por GraphQL y REST).

## web-identity-application

Estado de autenticación (NgRx Signals con eventos) detrás de una facade.

- **Import:** `@task-stream/web/identity/application`
- **Capa:** `application` · **Tags:** `scope:web-identity`, `type:application`
- `AuthFacade` (lo único que se exporta, junto al tipo `AuthStatus`): sesión y usuario como señales; `signInWith*()`, `completeSignIn()`, `signOut()`, `validAccessToken()`.
- Interno: `AuthStore` (login con PKCE o email y contraseña de Pireon; restaura la sesión al arrancar y renueva el token) y eventos `authPageEvents`, `authApiEvents`, `sessionEvents`.

## web-identity-domain

Sesión y usuario en el navegador.

- **Import:** `@task-stream/web/identity/domain`
- **Capa:** `domain` · **Tags:** `scope:web-identity`, `type:domain`
- `Session`, `CurrentUser`, `AuthConfig`, `isExpired`, `sessionFrom`.
- Puertos: `AuthGateway`, `ProfileGateway`, `SessionStorage`, `AuthorizationFlow`.

## web-identity-infrastructure

Adaptadores de identidad del navegador.

- **Import:** `@task-stream/web/identity/infrastructure`
- **Capa:** `infrastructure` · **Tags:** `scope:web-identity`, `type:infrastructure`
- `HttpAuthGateway` (`/api/auth/*`), `GraphqlProfileGateway` (`me`), `LocalSessionStorage`.
- `BrowserAuthorizationFlow`: Authorization Code + PKCE con Web Crypto y `state`.
- `SessionTokenSource`: conecta el cliente GraphQL con la `AuthFacade`.
- `provideIdentityInfrastructure()`.

## web-identity-presenters

Pantallas y guards de identidad.

- **Import:** `@task-stream/web/identity/presenters`
- **Capa:** `presenters` · **Tags:** `scope:web-identity`, `type:presenters`
- Rutas `identityRoutes`: `/login` y `/auth/callback`.
- `authGuard`, `guestGuard` y `UserMenuComponent`.

## web-shared-infrastructure

Cliente GraphQL de la web.

- **Import:** `@task-stream/web/shared/infrastructure`
- **Capa:** `infrastructure` · **Tags:** `scope:web-shared`, `type:infrastructure`
- `provideGraphql()`: Apollo con HTTP para queries/mutations y `graphql-ws` para subscriptions; caché desactivada.
- `AccessTokenSource` (puerto): de dónde sale el token (lo implementa identity).
- `API_CONFIG` / `apiConfigFromLocation()`: la API se sirve en el mismo origen (`/api`, `/graphql`).

## web-shared-ui

Integración de `@pi/ui` y piezas de interfaz comunes.

- **Import**: `@task-stream/web/shared/ui`
- **Capa**: `presenters` · **Tags**: `scope:web-shared`, `type:presenters`
- `providePiUi()`: animaciones, iconos, textos de la librería, tema y modo oscuro.
- `icons.ts`: iconos SVG en línea, incluidos los que usa la propia librería.
- `AppearanceService` (modo oscuro), `ToastService` (`pi-snack-bar`), `DrawerService` (`pi-drawer`), `AppShellComponent` (la cabecera de pireon.dev: logo, «Pireon | TaskStream», luces de fondo) y `LogoComponent` (el logo de pireon.dev, en `apps/web/public`).

## web-tracker-application

Estado del tracker (NgRx Signals con eventos) detrás de facades.

- **Import**: `@task-stream/web/tracker/application`
- **Capa**: `application` · **Tags**: `scope:web-tracker`, `type:application`
- Facades (lo que se exporta): `ProjectsFacade` y `HomeFacade` («Mi trabajo») (raíz) y `ProjectFacade`, `BoardFacade`, `IssueFacade` (de ruta, con `provideProjectFacade()`, `provideBoardFacade()`, `provideIssueFacade()`). Tests en `facades/facades.spec.ts`.
- `ExploreFacade` (raíz): proyectos públicos, también sin sesión.
- `ProjectFacade` también da la visibilidad (`setPublic`) y el borrado (`deleteProject`, vuelve a la portada; y si otro lo borra, también), y la configuración del proyecto (`workflow`, `saveSettings`) y sus estadísticas (`loadAnalytics`, `loadBurnup`).
- Interno: stores `ProjectsStore`, `ProjectStore`, `HomeStore`, `BoardStore`, `IssueStore` y eventos por origen (`*PageEvents`, `*ApiEvents`, `*RealtimeEvents`).
- `board.reducers.ts`: transformaciones puras del tablero (movimiento optimista, cambios en vivo por versión).
- Puerto `Notifications`.

## web-tracker-domain

Modelo del tracker en el cliente.

- **Import**: `@task-stream/web/tracker/domain`
- **Capa**: `domain` · **Tags**: `scope:web-tracker`, `type:domain`
- Modelos (`Project`, `IssueSummary`, `Issue`, `Release`...), enums e inputs.
- Configuración por proyecto (`ProjectSettings`, `StatusCategory`) y `Workflow`: nombres, colores, iconos, categorías, transiciones permitidas y estados que piden motivo, con un nombre legible para ids que ya no están configurados.
- Estadísticas (`ProjectAnalytics`, `ReleaseBurnup`).
- Puertos: `ProjectsGateway`, `IssuesGateway`, `CommentsGateway`, `ReleasesGateway`, `ActivityGateway`, `UsersGateway`, `TrackerRealtimeGateway`.

## web-tracker-infrastructure

Adaptadores GraphQL del tracker.

- **Import**: `@task-stream/web/tracker/infrastructure`
- **Capa**: `infrastructure` · **Tags**: `scope:web-tracker`, `type:infrastructure`
- `graphql/*.graphql`: operaciones; `generated.ts` lo genera `npx nx run web:codegen`.
- `adapters/`: implementaciones de los gateways y `mappers.ts` (GraphQL → dominio, y la configuración de vuelta a input sin `__typename`).
- `provideTrackerInfrastructure()`.

## web-tracker-presenters

Pantallas del tracker con `@pi/ui`.

- **Import**: `@task-stream/web/tracker/presenters`
- **Capa**: `presenters` · **Tags**: `scope:web-tracker`, `type:presenters`
- Rutas `trackerRoutes`: inicio (proyectos + «Mi trabajo», `pages/my-work.ts`), tablero kanban (drag & drop), ticket, releases, seguimiento, estadísticas, actividad, miembros y configuración (con visibilidad y borrado), más «Explorar» (`pages/explore.page.ts`). Solo la portada exige sesión.
- `ui/`: `IssueCardComponent`, `AvatarsComponent`, `GoToComponent` («Ir a… WEB-12»), `ProjectCardComponent`, `SeriesChartComponent` (gráficas SVG de líneas y áreas apiladas) y `copy.ts` (textos y tonos; estados, tipos y áreas salen del `Workflow` del proyecto).