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

Anexo A. Librerías

1. Visión general

Qué resuelve

Necesidad	Cómo se resuelve
🐛 Bugs	Tickets de tipo `BUG` con prioridad, área y etiquetas
🎨 Cambios de UI	Tipo `UI_CHANGE` con enlace al frame de Figma (validado contra `figma.com`)
⚙️ Features	Tipo `FEATURE` siguiendo el workflow completo
🧪 QA	Estados `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)
👥 Asignaciones	Solo a miembros del proyecto; panel de carga por persona
💬 Comentarios	Decisiones y explicaciones dentro del ticket; el autor edita, el admin modera
🔗 Relaciones	`BLOCKS`, `DEPENDS_ON`, `TESTS`, `DUPLICATES`, `RELATES_TO`, también entre proyectos (FE ↔ BE ↔ QA)
📊 Seguimiento	Panel con contadores, gráfico por estado, carga por persona, feed de actividad e historial por ticket
⚡ 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`

Pieza	Tecnología
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
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      dos destinos: api y web
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).

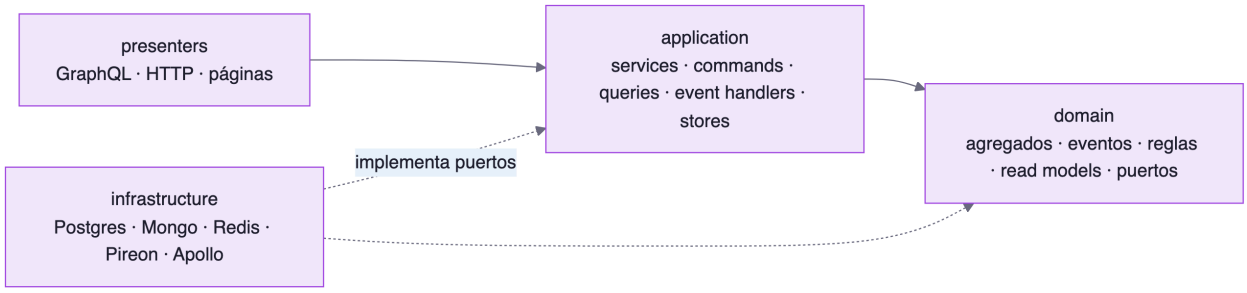
2. Arquitectura

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

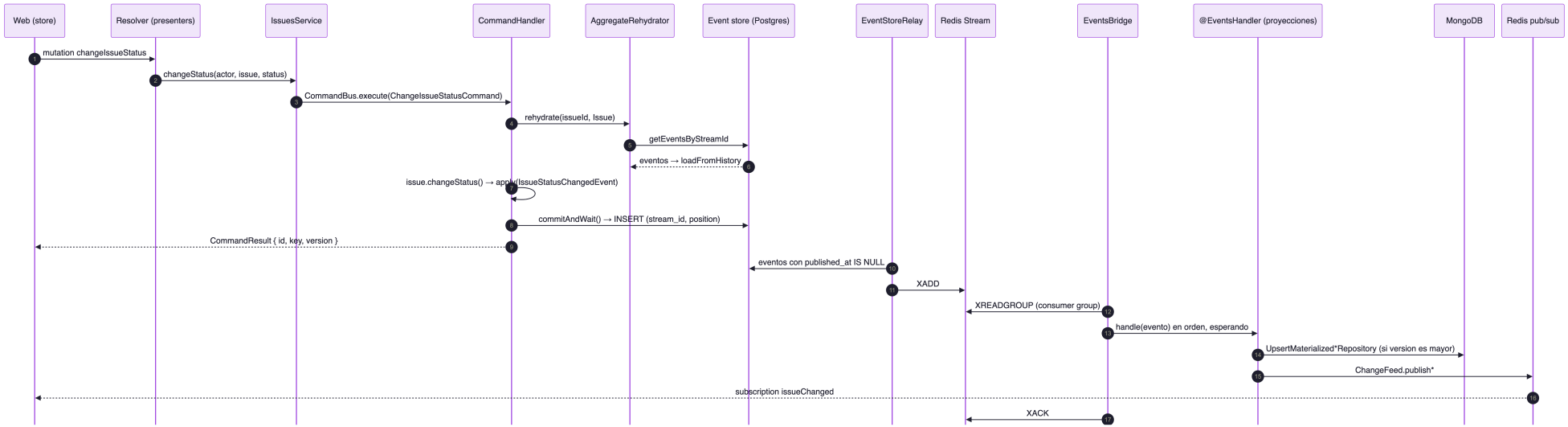
Las apps son la *composition root*: montan cada contexto de fuera hacia dentro.

```
// 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).

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.

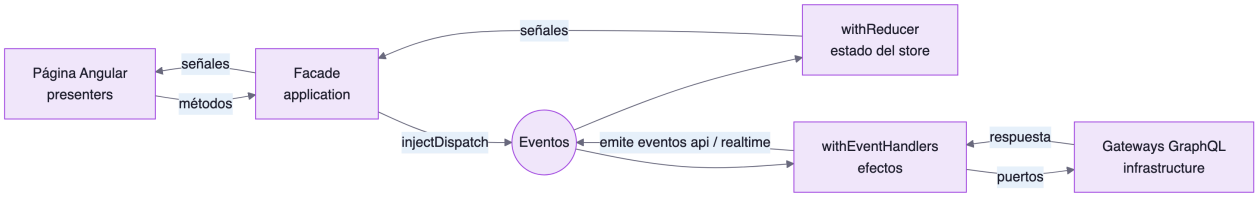
Consistencia eventual

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

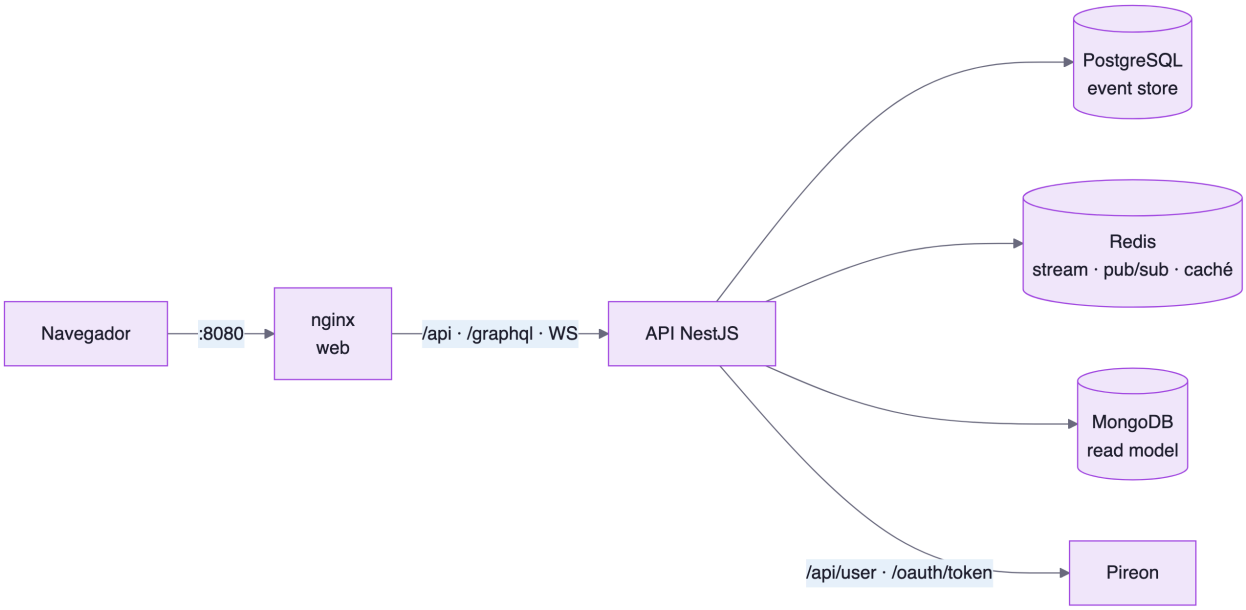
- El cliente reintenta las queries que devuelven *not found* justo después de crear algo.
- Si necesitas el dato actualizado, espera un evento de subscription con `version ⩾ result.version`.
- Las reglas que necesitan datos exactos **no** leen Mongo: por ejemplo, "no publicar una release con tickets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

2.3 El cliente web

El cliente replica las capas (ver Cliente web):



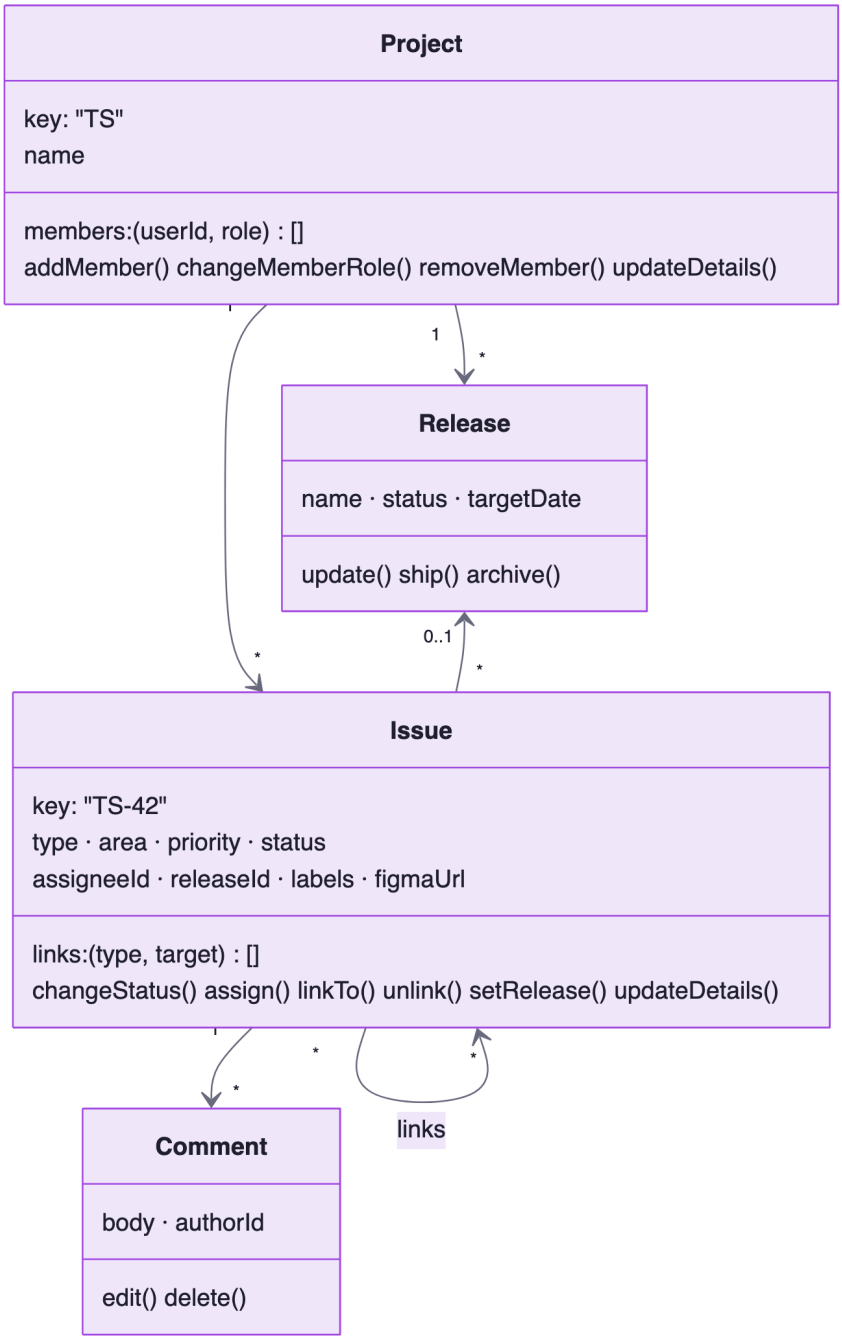
2.4 Diagrama de despliegue



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`)



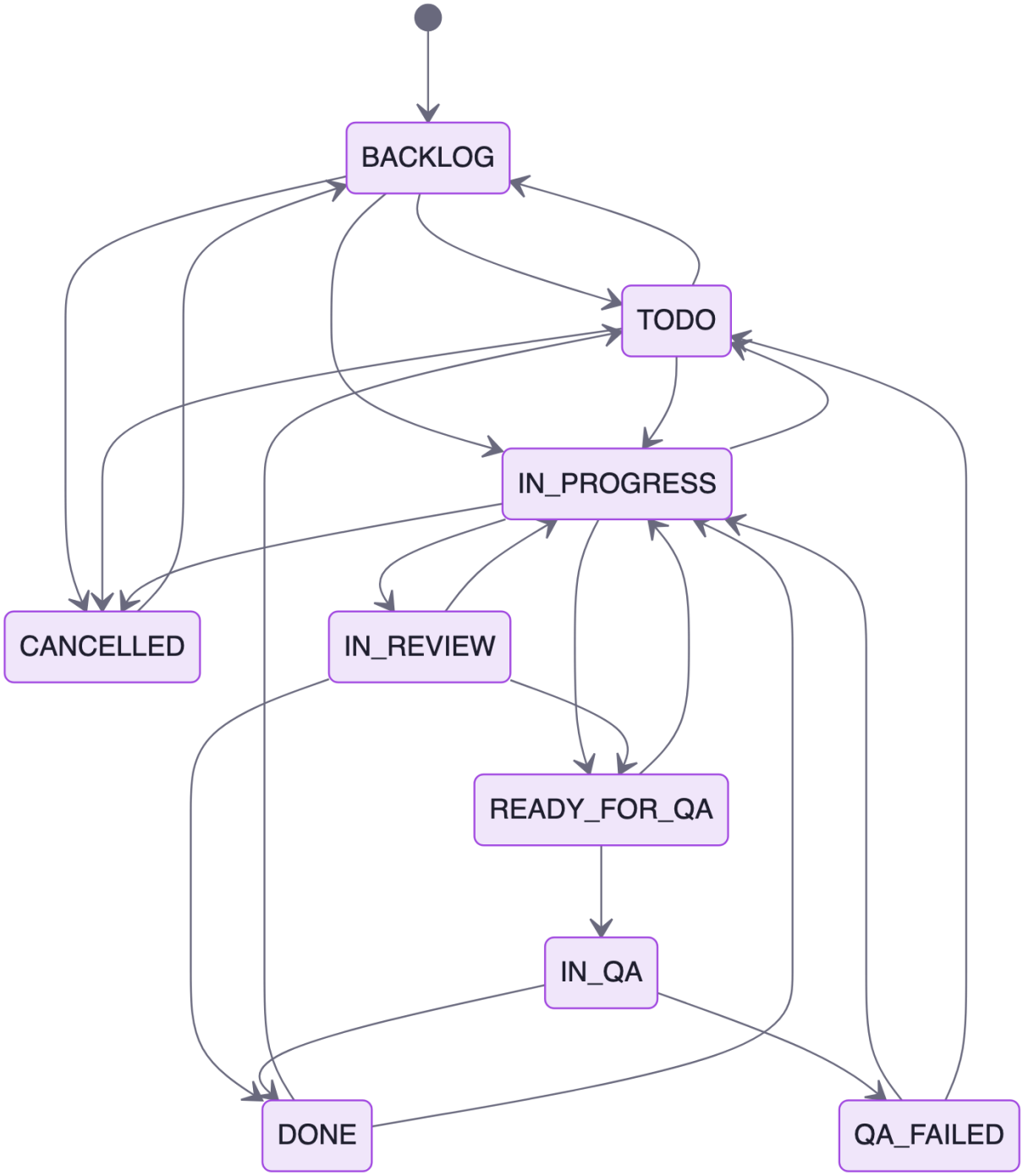
Roles por proyecto

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.

Workflow de un ticket



- `QA_FAILED` y `CANCELLED` exigen un **motivo**, que queda en el historial.
- No se puede pasar a `DONE` saltándose QA salvo desde `IN_REVIEW` (cambios que no requieren QA).
- `Issue.allowedTransitions` en GraphQL devuelve los estados permitidos desde el actual.

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`.
- **Comentarios**: solo el autor edita; borra el autor o un admin del proyecto.
- **Cambios de UI**: `figmaUrl` debe ser un enlace `https://…figma.com/…`.

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`, `ProjectMemberAddedEvent`, `ProjectMemberRoleChangedEvent`, `ProjectMemberRemovedEvent`
Issue	`IssueCreatedEvent`, `IssueUpdatedEvent`, `IssueStatusChangedEvent`, `IssueAssignedEvent`, `IssueLinkedEvent`, `IssueUnlinkedEvent`, `IssueReleaseChangedEvent`
Comment	`CommentAddedEvent`, `CommentEditedEvent`, `CommentDeletedEvent`
Release	`ReleaseCreatedEvent`, `ReleaseUpdatedEvent`, `ReleaseShippedEvent`, `ReleaseArchivedEvent`

3.2 GraphQL

Todas las operaciones requieren `Authorization: Bearer <token de Pireon>` (en subscriptions, en el `connection_init`).

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
`project(idOrKey)`	Un proyecto. Campos calculados: `board(releaseId, assigneeId, area, perColumn)`, `stats`, `releases`
`issues(filter, limit, offset)`	Búsqueda de tickets (estado, tipo, área, prioridad, persona, release, etiquetas, texto)
`issue(idOrKey)`	Un ticket. Campos calculados: `allowedTransitions`, `linkedIssues`, `comments`, `history`
`release(id)`	Una release. Campos calculados: `progress`, `issues`

Query	Devuelve
`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`, `addProjectMember`, `changeProjectMemberRole`, `removeProjectMember`
Tickets	`createIssue`, `updateIssue`, `changeIssueStatus`, `assignIssue`, `linkIssues`, `unlinkIssues`, `setIssueRelease`
Comentarios	`addComment`, `editComment`, `deleteComment`
Releases	`createRelease`, `updateRelease`, `publishRelease(force)`, `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)
`activityAdded(projectId)`	Hay una nueva entrada en el feed
`releaseChanged(projectId)`	Cambia una release
`projectChanged(projectId)`	Cambian los datos 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

```
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
      }
    }
  }
  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` y el webhook, exigen `Authorization: Bearer <token de Pireon>` (el `AuthGuard` global, igual que en GraphQL). El documento 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 /api/projects/{project}`, `…/board`, `…/stats`, `…/activity`, `…/members` (POST, PATCH, DELETE)

Recurso	Rutas
Tickets	`GET/POST /api/projects/{project}/issues`, `GET/PATCH /api/issues/{issue}`, `PUT …/status`, `PUT …/assignee`, `PUT …/release`, `POST/DELETE …/links`, `GET/POST …/comments`, `GET …/history`
Comentarios	`PATCH/DELETE /api/comments/{commentId}`
Releases	`GET/POST /api/projects/{project}/releases`, `GET/PATCH /api/releases/{id}`, `…/progress`, `…/issues`, `POST …/publish`, `POST …/archive`
Sistema	`GET /api/health`, `GET /api/auth/config`, `POST /api/auth/token`, `POST /api/webhooks/pireon`

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, ticket, releases, seguimiento, actividad y miembros

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: IssueStatus; 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: IssueStatus, 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`)
`ProjectFacade`	`ProjectStore`	ruta `/p/:key`	Proyecto, miembros, stats, releases, actividad en vivo y búsqueda de usuarios
`BoardFacade`	`BoardStore`	ruta `board`	Tablero kanban, filtros, movimiento optimista, nuevo ticket, cambios en vivo
`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.
- 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`).

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)

Particularidad	Cómo se resuelve
`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`; `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`

De pireon.dev	En TaskStream
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`.

4.6 Rutas

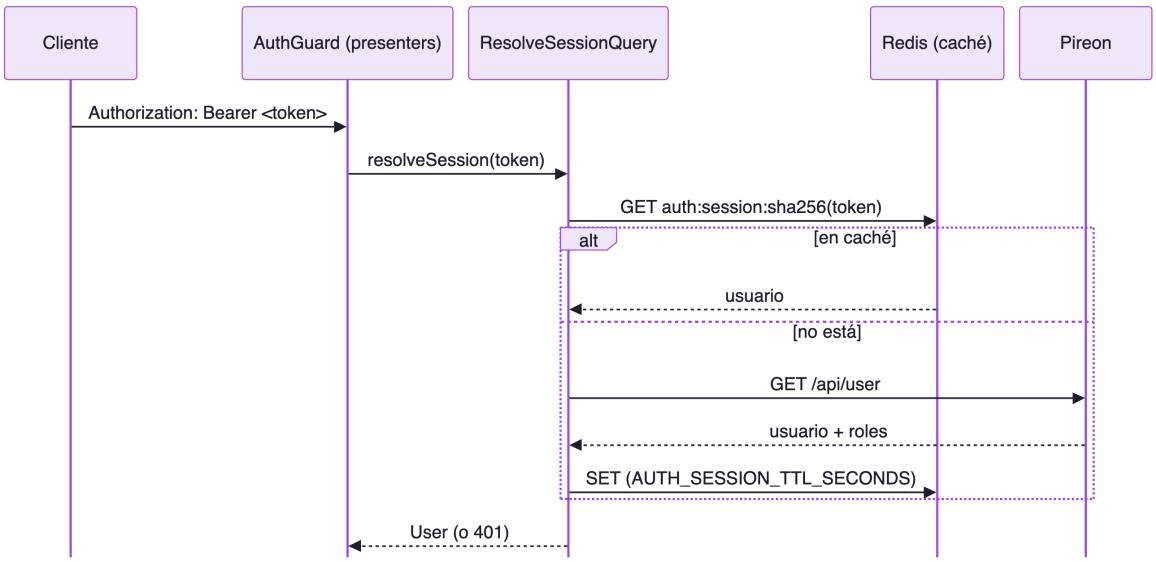
Ruta	Página
`/login`	Inicio de sesión (solo sin sesión)
`/auth/callback`	Retorno de Pireon (PKCE)
`/`	Mis proyectos
`/p/:key/board`	Tablero kanban (`?release=` filtra por release)
`/p/:key/issues/:issueKey`	Ticket
`/p/:key/releases`	Releases
`/p/:key/dashboard`	Seguimiento
`/p/:key/activity`	Actividad en vivo
`/p/:key/members`	Miembros

Todo salvo `/login` y `/auth/callback` está protegido por `authGuard`. 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 y webhook).
- 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.

Método	Necesita	Flujo
Entrar con Pireon (Authorization Code + PKCE)	`PIREON_CLIENT_ID`, `PIREON_CLIENT_SECRET` y la redirect URI registrada	Web → `/oauth/authorize` de Pireon → `/auth/callback` → `POST /api/auth/token` (`authorization_code` + `code_verifier`)
Email y contraseña	El mismo cliente, con `PIREON_PASSWORD_GRANT=true`	`POST /api/auth/token` (`password`)

Las dos opciones necesitan el cliente OAuth: sin `PIREON_CLIENT_ID` la API responde `503` y el login avisa de que no está configurado. 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

1. Crea el cliente: `POST https://oauth.pireon.dev/api/clients` (con un token tuyo de Pireon, scope `account:write`).
2. Registra como redirect URIs:
   - `http://localhost:4200/auth/callback` (desarrollo)
   - `http://localhost:8080/auth/callback` (Docker)
   - la de producción
   - para el botón «Authorize» de Swagger UI: `http://localhost:3000/api/docs/oauth2-redirect.html` y la de producción (ver API REST)
3. Pon `PIREON_CLIENT_ID` y `PIREON_CLIENT_SECRET` en `.env` (o en el entorno del contenedor).

`GET /api/auth/config` devuelve a la web lo necesario para iniciar el flujo (`authorizeUrl`, `clientId`, `scopes`, `passwordGrant`), o `null` si no hay cliente configurado; en ese caso el login muestra el aviso y no deja entrar.

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`.

La API verifica la firma `X-Webhook-Signature` (HMAC-SHA256 del cuerpo), marca al usuario como inactivo y borra sus sesiones cacheadas. Sin `PIREON_WEBHOOK_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:prune`	Build de la API con `package.json` y lockfile podados (lo usa Docker)
`npx nx graph`	Grafo de dependencias entre proyectos
`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. Arrancar la API para regenerar `apps/api/schema.gql`.

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

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 dos 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`.
- `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` 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
`EVENT_STREAM_KEY`	`task-stream:events`	Clave del Redis Stream
`EVENT_STREAM_MAX_LENGTH`	`1000000`	Longitud aproximada que se conserva en el stream

Variable	Por defecto	Descripción
`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 para el login desde la web
`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_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)

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`.
- **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.

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.

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.

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`.
- 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()`, `@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`.
- `SharedModule.withInfrastructure(...)`: módulo global que expone el event store y el rehydrator 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).
- 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).
- `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 la membresía (`ReadAccess`).
- `event-handlers/`: proyecciones (`@EventsHandler`) hacia el read model, notificaciones en tiempo real y feed de actividad.
- `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`.
- `factories/`: `ProjectFactory`, `IssueFactory`, `CommentFactory`, `ReleaseFactory` (`@Injectable()`).
- `issue/issue-workflow.ts`: transiciones de estado permitidas y estados que exigen motivo.
- `read-models/`: forma de las vistas de lectura (`ProjectReadModel`, `IssueReadModel`...).

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`) y `ReleaseIssuesRepository` (consulta el event store).
- `mongo/`: esquemas del read model, repositorios `Find*` y `UpsertMaterialized*` (upsert idempotente por versión).
- `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).

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` (raíz) y `ProjectFacade`, `BoardFacade`, `IssueFacade` (de ruta, con `provideProjectFacade()`, `provideBoardFacade()`, `provideIssueFacade()`). Tests en `facades/facades.spec.ts`.
- Interno: stores `ProjectsStore`, `ProjectStore`, `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.
- Workflow (`REASON_REQUIRED`, `BOARD_COLUMNS`).
- 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).
- `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`: proyectos, tablero kanban (drag & drop), ticket, releases, seguimiento, actividad y miembros.
- `ui/`: `IssueCardComponent`, `AvatarsComponent` y `copy.ts` (textos, iconos y tonos de color por estado, tipo, prioridad y proyecto).