
ETAPA 0 — Acceso a la plataforma / Selección de institución (multi-tenant)

0.1 El usuario (familia / tutor / estudiante) entra a la landing de admisiones de una institución específica.

- Ejemplo: “Instituto X”, “Universidad Y”.
- Esa institución se identifica en el sistema como `tenant_id` / `institucion`.

0.2 El frontend carga la configuración visual y reglas de esa institución.

- Llama al endpoint `/api/tenant-config/:tenant_id` (ruta `tenantConfig.js`).
- El backend lee `TenantConfig` en MongoDB (`models/TenantConfig.js`).

MongoDB guarda por cada institución:

- nombre, logo, colores, textos de bienvenida,
- qué campos tiene que pedir en cada fase (A, B, C),
- si la fase requiere comité, entrevista, pago, etc.,
- programas que ofrece,
- ubicaciones, contacto.

💡 Esto es lo que les da el “white-label”: mismo sistema backend, pero cada institución tiene su look, mensajes y reglas. Todo viene de MongoDB, no está hardcodeado.

🔗 Relaciones importantes para el diagrama:
[Usuario] → [Frontend SPA]
[Frontend SPA] → (/api/tenant-config/:tenant_id) → [TenantConfig.js (backend)] → [MongoDB TenantConfig]

Salida de la etapa:
👉 Ya sabemos quién es el tenant (institución) y cuáles son sus reglas. Esa info se usa en todas las fases siguientes.

ETAPA 1 — Fase A: Prospección / Intención de postular

Objetivo de esta fase: capturar interesados a muy alta escala, rápido, con control de duplicados. Acá todavía NO es matrícula oficial. Es “quiero información / quiero empezar el trámite”.

1.1 El usuario completa un formulario inicial de interés:

- nombre, apellido
- email
- teléfono
- (opcional) carrera/programa elegido

1.2 El frontend manda esos datos a `/api/prospection` por POST (ruta `prospection.js`).

1.3 ¿Qué hace el backend en `prospection.js`?

- Aplica rate limiting (`express-rate-limit`) → protege de spam/bots.
- Busca en Redis si ese email ya existe (`GET prospecto:<email>`).
    - Si ya existe: responde “ya iniciaste tu postulación”.
    - Si no existe:
        - Guarda el prospecto en Redis (`SETEX prospecto:<email> ... TTL 30 días`).
        - Incrementa `contador:prospectos` en Redis (métrica de interés total).
        - Marca estado inicial `"interesado"`.
        - Devuelve al frontend un ok + timestamp.

Base usada acá: **Redis**
Por qué Redis:

- Altísima velocidad de escritura concurrente.
- Ideal para picos masivos (tipo 1M interesados en 2h).
- TTL automático para datos efímeros de lead / marketing.
- Validación de duplicados al toque.

🔗 Relaciones para dibujar:
[Usuario] → [Frontend "Formulario de Interés"]
→ `/api/prospection` → [Servicio Prospección (Redis)]
→ [Redis guarda prospecto + contador global]

Salida de la etapa:
👉 Tenemos un “prospecto” registrado en Redis, con `email` y “estado = interesado”.

👉 Ya no es anónimo. Ahora ese email se convierte en la clave del postulante para el resto del proceso.

ETAPA 2 — Fase B: Admisión / Expediente académico-administrativo

---

Objetivo de esta fase: armar el expediente del postulante, recolectar documentación, evaluarlo y dejar evidencia de todo. Acá entra el comité.

2.1 El usuario (o personal interno cargando datos del alumno) aporta información más completa:

- Documentación (DNI escaneado, analítico, constancias, etc.).
- Comentarios, notas de entrevista, observaciones.
- Estado del proceso (“en revisión”, “aprobado por comité”, “rechazado”, etc.).

2.2 El frontend envía esa info a `/api/admission` por POST (ruta `admission.js`).

2.3 ¿Qué hace el backend en `admission.js`?

- Usa el modelo `Expediente` definido en `config/mongodb.js`.
- Si ya existe un expediente MongoDB para ese `email`, no crea uno nuevo; si no existe, lo crea.
- Estructura almacenada (en MongoDB):
    - `email`
    - `documentos`: arreglo flexible de objetos (tipo, URL, etc.)
    - `comentarios`: libre, observaciones internas
    - `estado`: ej. `"en_revision"`, `"aprobado"`, `"rechazado"`
    - `metadata`:
        - `fechaCreacion`
        - `ultimaActualizacion`
    - `historial`: lista de eventos (“actualización de expediente”, quién lo tocó, cuándo y qué cambió)

También hay endpoints GET/PUT/DELETE:

- GET `/api/admission` → lista todos los expedientes (para el staff).

- GET `/api/admission/:email` → trae el expediente de una persona.
- PUT `/api/admission/:email` → actualiza, cambia estado, agrega al historial.
- DELETE `/api/admission/:email` → borra el expediente (sólo para testing/demos).

Base usada acá: **MongoDB**
Por qué MongoDB:

- Permite guardar documentos complejos y flexibles (cada institución puede pedir requisitos distintos).
- Guarda historial interno editable / auditoría narrativa.
- Sirve para que el comité académico tenga toda la info reunida.

🔗 Relaciones para dibujar:
[Usuario / Staff de admisiones]
→ [Frontend "Cargar Documentación / Evaluar"]
→ `/api/admission` → [Servicio Admisión / Expediente]
→ [MongoDB.Expediente con historial y estado del postulante]

Salida de la etapa:
👉 Existe un “expediente de admisión” en MongoDB vinculado a ese email.
👉 Ese expediente tiene un `estado` decidido por la institución/comité (“aprobado” es la llave para avanzar a la matrícula final).

———————————

ETAPA 3 — Fase C: Inscripción formal / Matrícula definitiva

———————————

Objetivo: convertir un postulante aprobado en alumno inscripto oficial; registro auditable e inmutable.

Esta parte está implementada en `enrollment.js` y la base Cassandra con `config/cassandra.js`.

3.1 Cuando el comité aprueba al postulante (por ejemplo, “aceptado”), el staff o el sistema llama al endpoint `/api/enrollment` (POST).

Body típico:

- `institucion` (el tenant / institución)
- `email`
- `nombre`, `apellido`
- `programa` (carrera / curso al que entra)

3.2 ¿Qué hace el backend en `enrollment.js` (POST)?

- Genera un `id_inscripcion` (UUID).
- Inserta un registro en Cassandra en la tabla `inscripciones`.

La tabla `inscripciones` (Cassandra) se crea en `config/cassandra.js` con este esquema clave:

```
CREATE TABLE IF NOT EXISTS inscripciones (
  institucion text,
  email text,
  id_inscripcion uuid,
  nombre text,
  apellido text,
  programa text,
  fecha_inscripcion timestamp,
  PRIMARY KEY ((institucion), email, id_inscripcion)
) WITH CLUSTERING ORDER BY (email ASC, id_inscripcion DESC)
```

Claves importantes:

- `institucion` es la partition key → multi-tenant REAL.
  Cada institución tiene su propia partición de datos.
- No se sobreescribe: se insertan nuevas filas con distintos `id_inscripcion`.
  Eso genera un historial ordenable por tiempo, auditable e inmutable.

3.3 Luego se pueden consultar las inscripciones:

- GET `/api/enrollment/institucion/:institucion`
  Devuelve TODAS las inscripciones hechas en esa institución.
  Esto sirve para un dashboard administrativo de esa institución.
- GET `/api/enrollment/institucion/:institucion/email/:email`
  Devuelve el historial de inscripción de ese email dentro de esa institución.

Base usada acá: **Cassandra**
Por qué Cassandra:

- Alta escritura concurrente y rápida sin bloquear.
- Mantiene histórico append-only (auditoría).
- Particiona por `institucion`, lo que respeta el modelo multi-tenant sin mezclar datos entre escuelas.

- Está pensada para ser el registro “oficial” de matriculación final.

🔗 Relaciones para dibujar:
[Comité aprueba al postulante] / [Staff confirma matrícula]
→ [Frontend "Confirmar inscripción final"]
→ `/api/enrollment` (POST) → [Servicio Inscripción Final]
→ [Cassandra.inscripciones (registro inmutable, particionado por institucion)]
→ Consultas GET para dashboards administrativos / reporte final.

Salida de la etapa:
👉 El estudiante ya está inscripto oficialmente en la institución X, carrera Y, con un alta con timestamp.
👉 Ese registro queda guardado en Cassandra como fuente de verdad final.

ETAPA 4 — Relaciones académicas y análisis institucional

Objetivo: entender vínculos y análisis tipo “quién se inscribió dónde”, popularidad de programas, redes entre alumnos ↔ institución ↔ programa. Esto no es aprobación/matrícula, es inteligencia.

Implementado en `relations.js` usando Neo4j.

4.1 Cuando un alumno se registra o se inscribe, el sistema puede crear relaciones en Neo4j:

- POST `/api/relations`
  Body:
  - `email_estudiante`
  - `institucion`
  - `programa`

4.2 ¿Qué hace el backend en `relations.js` (POST)?

- Abre una sesión Neo4j.
- Crea nodos:
  - `(Estudiante {email})`
  - `(Institucion {nombre})`
  - `(Programa {nombre})`
- Crea relaciones:

- `(:Estudiante)-[:POSTULA_A]->(:Institucion)`
- `(:Estudiante)-[:INSCRITO_EN]->(:Programa)`
- `(:Programa)-[:PERTENECE_A]->(:Institucion)`

4.3 Consultas disponibles:

- GET `/api/relations/:email`
  Devuelve en qué institución se postuló ese estudiante y en qué programa quedó.
- GET `/api/relations/institucion/:nombre`
  Devuelve todos los emails que postularon/están en esa institución.
- GET `/api/relations/stats/programas-populares`
  Devuelve qué programas tienen más inscriptos.
- GET `/api/relations/stats/instituciones`
  Devuelve ranking de instituciones por cantidad de estudiantes.

Base usada acá: **Neo4j**
Por qué Neo4j:

- Permite explotar las relaciones entre alumnos-instituciones-programas sin tener que hacer joins locos.
- Sirve para analíticas y métricas del sistema completo (tendencias, popularidad de carreras, etc.).
- Esto le da valor agregado al Ministerio / dirección académica.

🔗 Relaciones para dibujar:
[Servicio Inscripción / Admisión]
→ `/api/relations` → [Servicio Relaciones Académicas]
→ [Neo4j grafo alumno ↔ institución ↔ programa]
→ `/api/relations/...` para métricas, rankings, popularidad.

Salida de la etapa:
👉 Podemos responder preguntas tipo: “¿Cuál es el programa más elegido?” o “¿Cuántos alumnos entraron a la Institución X?” sin golpear Cassandra ni Mongo, usando grafo optimizado.

ETAPA 5 — Seguridad y control

En el backend hay middleware de autenticación (`authMiddleware.js`) y utilidades JWT (`utils/jwt.js`).
Esto controla que ciertos endpoints sensibles (ej. `/api/tenant-config/:tenant_id`)

sólo se puedan consultar si el usuario pertenece a ese tenant.
Traducción: un admin de la Universidad A no puede leer la config de la Universidad B.

🔗 Relaciones para dibujar:
[Frontend / Admin autenticado]
→ [authMiddleware valida JWT y tenant_id]
→ [/api/tenant-config/:tenant_id]
→ [MongoDB TenantConfig SOLO de su institución]

Esto refuerza el multi-tenant seguro.

RESUMEN PARA EL DIAGRAMA FINAL

Podés armar el workflow en draw.io con una línea principal y bifurcaciones por base, en este orden:

1. **Selección de institución (multi-tenant)**
    - Usuario entra → Frontend pide config de la institución
    - `/api/tenant-config/:tenant_id`
    - MongoDB TenantConfig
2. **Fase A – Prospección (Interés inicial)**
    - Usuario deja datos básicos → `/api/prospection`
    - Redis guarda prospecto + contador, valida duplicados
    - Estado = "interesado"
3. **Fase B – Admisión (Expediente)**
    - Usuario/staff sube docs, comité evalúa → `/api/admission`
    - MongoDB guarda expediente completo (documentos, comentarios, estado, historial de cambios)
    - Comité define si aprueba
4. **Fase C – Inscripción final (Matrícula)**
    - Comité confirma inscripción → `/api/enrollment`
    - Cassandra registra la inscripción oficial e INMUTABLE
    - Consultas GET listan inscripciones por institución o por persona
5. **Grafo de relaciones académicas**

    - Se registra relación estudiante ↔ institución ↔ programa → `/api/relations`
    - Neo4j guarda nodos y relaciones
    - Métricas: programas populares, instituciones con más inscriptos
6. **Autenticación / Control de acceso**
    - authMiddleware con JWT
    - Garantiza que cada institución sólo vea SU data (multi-tenant seguro)

Esta secuencia A→B→C cumple exactamente con el CVA que les piden (Interés → Admisión → Inscripción), y encima muestra la Persistencia Políglota que exige el trabajo:

- Redis = alta concurrencia / captura instantánea / lead
- MongoDB = expediente flexible y decisión del comité
- Cassandra = registro final oficial, auditable, particionado por institución
- Neo4j = análisis de relaciones, métricas y reporting inteligente