Universidad Icesi

Spec-driven Development y Supabase

Seminario de I+D+I · Semana 14

Universidad Icesi

Universidad ICESI

Sesión 1

Spec-driven Development

El cuello de botella ya no es escribir código

Antes

Escribir el código a mano: horas

generar — el trabajo se movió acá

Hoy

segundos

Entender, revisar y corregir lo que se generó

Sin dirección clara
el modelo rellena los vacíos con lo que "suena bien"

Con una especificación
el modelo construye lo que se decidió de antemano

Herramientas como Copilot, Cursor, Claude Code o Antigravity generan cientos de líneas en segundos. **La velocidad ya no es el problema; la dirección sí.**

Vibe coding: velocidad sin dirección

El modelo alucina librerías obsoletas
Usa versiones o APIs que ya no existen o que tu proyecto no admite.

Viola la arquitectura por capas
Inyecta repositorios en controladores o mezcla la interfaz con la base de datos.

Acumula deuda técnica masiva
Cada respuesta funciona sola, pero nadie entiende cómo encaja el conjunto.

Universidad ICESI

Especificar

Planificar

Dividir

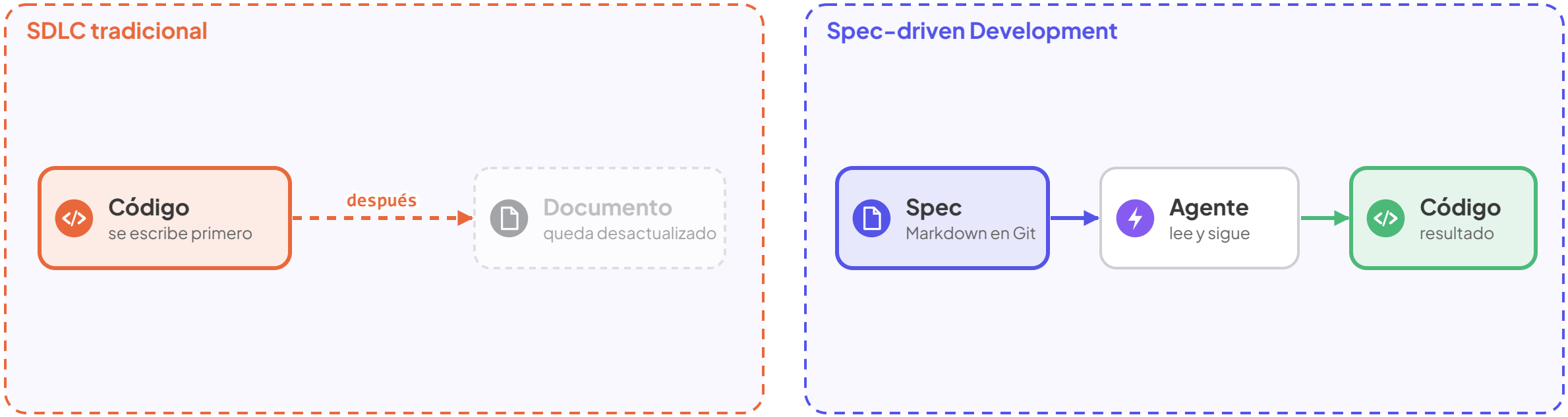
Implementar

Spec-driven Development

Una disciplina que traslada el foco de **"generar código rápido"** a **"definir con precisión qué debe construirse"** antes de escribir una sola línea.

- La especificación es un archivo Markdown versionado en Git
- El agente la lee y la sigue; tú la revisas
- Si el requisito cambia, cambia primero el documento

La documentación: ¿subproducto o motor?



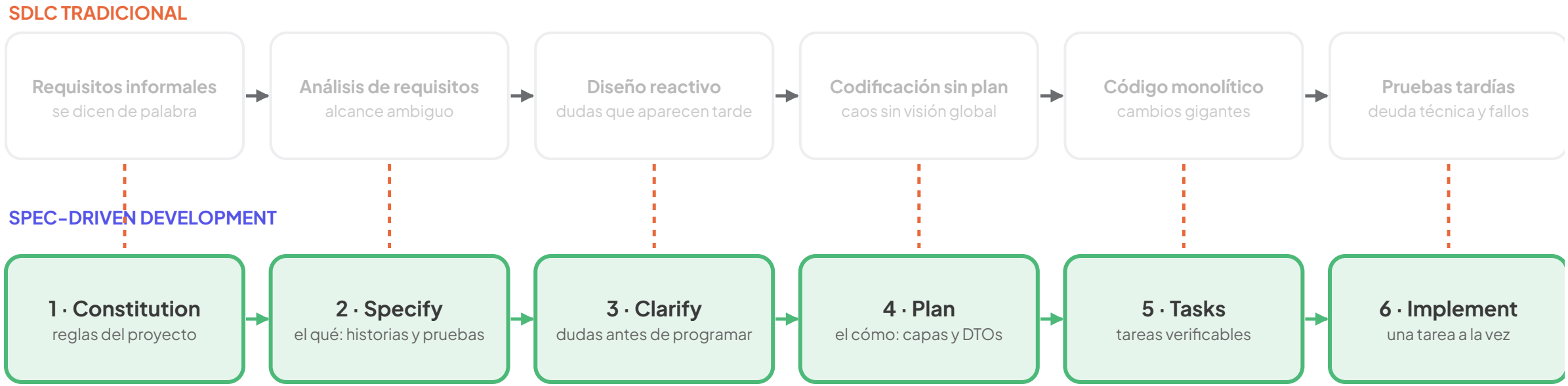
En el SDLC la documentación es un subproducto que se desactualiza al primer cambio. En SDD **la especificación es el motor ejecutable** que guía la generación de código.

El ciclo de vida del SDLC tradicional



Cada etapa arrastra su propio riesgo: lo que no se decide al inicio **se descubre tarde** y sale caro.

SDD: una fase para evitar cada riesgo



SDD no agrega burocracia: **mueve las decisiones al principio**, en documentos que el agente lee y tú revisas.

Estructura del repositorio: el contexto

Árbol de carpetas

```
mi-proyecto-backend/
├── AGENTS.md  <- constitución
├── .agents/
│   ├── memory/  <- restricciones estables
│   │   ├── tech-stack.md
│   │   └── architecture.md
│   ├── specs/  <- el QUÉ
│   │   └── feat-audit-spec.md
│   ├── plans/  <- el CÓMO
│   │   └── feat-audit-plan.md
│   └── tasks/  <- pasos atómicos
│       └── feat-audit-tasks.md
└── src/
```

AGENTS.md
es lo primero que lee el agente: identidad y reglas no negociables

.agents/memory/
versiones y convenciones estables: evitan librerías incompatibles

Estructura del repositorio: la feature

Árbol de carpetas

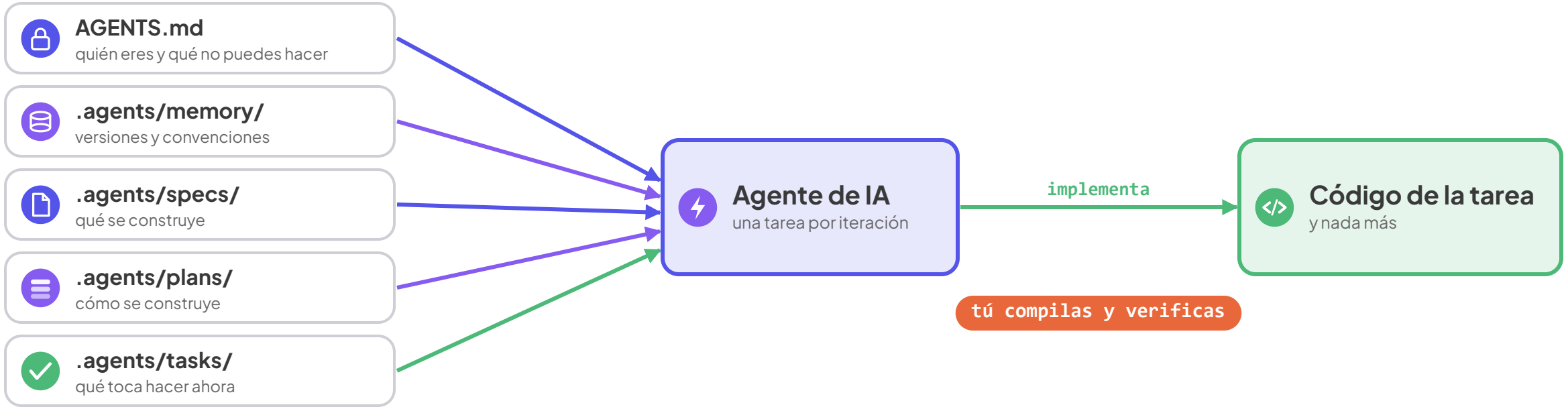
```
mi-proyecto-backend/
├── AGENTS.md  <- constitución
├── .agents/
│   ├── memory/  <- restricciones estables
│   │   ├── tech-stack.md
│   │   └── architecture.md
│   ├── specs/  <- el QUÉ
│   │   └── feat-audit-spec.md
│   ├── plans/  <- el CÓMO
│   │   └── feat-audit-plan.md
│   └── tasks/  <- pasos atómicos
│       └── feat-audit-tasks.md
└── src/
```

specs/
el QUÉ funcional, desde el usuario

plans/
el CÓMO técnico: clases, DTOs, transacciones

tasks/
checklist de pasos atómicos, cada uno verificable

Qué lee el agente y qué produce



El agente **lee todo el contexto del repositorio** antes de actuar, y solo escribe el código de la tarea autorizada.

Universidad ICESI

Paso 1 · AGENTS.md: la constitución

AGENTS.md

```
# Manual de operación del agente

Eres un Ingeniero de Software Senior en
backend con Spring Boot y Java 17.

## Reglas de oro inmutables
1. Capas estrictas: Controller -> Service
   -> Repository. Nunca un repositorio en
   un controlador.
2. DTOs obligatorios: nunca expongas una
   entidad JPA en un endpoint REST.
3. Excepciones de dominio, manejadas en
   @RestControllerAdvice.
4. Una sola tarea por iteración: consulta
   .agents/tasks/ y ejecuta una.
```

Identidad
qué rol cumple el agente

Reglas no negociables
lo que jamás debe hacer

Directorio de contexto
enlaces a la memoria del repositorio

Paso 2 · La memoria permanente

.agents/memory/tech-stack.md

```
# Restricciones técnicas del stack

- Lenguaje: Java 17 (LTS), con Records,
  Text Blocks y Switch Expressions.
- Framework: Spring Boot 3.3.x.
- Seguridad: Spring Security 6 con JWT
  stateless.
- Base de datos: PostgreSQL 16 con Spring
  Data JPA y Hibernate 6.
- Validaciones: Jakarta Validation.
- API: Springdoc OpenAPI v2 (Swagger UI).
```

Versiones exactas
el agente no inventa dependencias

Convenciones estables
se escriben una vez y valen para toda feature

Sin esta memoria, cada sesión con el agente empieza desde cero.

Paso 3 · Specify: la historia de usuario

.agents/specs/feat-audit-spec.md

```
# Especificación: Audit Log

## Historia de usuario
Como administrador del sistema,
quiero que cada acción crítica (login,
creación de cursos, cambio de roles) quede
registrada en un log persistente,
para auditar la seguridad y cumplir normas
de trazabilidad.

## Endpoints requeridos
- POST /api/v1/audit-logs (interno)
- GET  /api/v1/audit-logs (ROLE_ADMIN,
 paginado, orden descendente por fecha)
```

Como / quiero / para
quién, qué y por qué

Endpoints
el contrato HTTP, sin código

La spec describe el **qué**. No dice cómo se implementa.

Paso 3 · Specify: los criterios de aceptación

.agents/specs/feat-audit-spec.md

```
## Payload JSON de entrada
{
  "action": "COURSE_CREATED",
 "userEmail": "profesor@icesi.edu.co",
 "details": "Curso 101 con 3 créditos",
 "ipAddress": "192.168.1.45"
}

## Criterios (Gherkin)
Escenario 1: registro exitoso
 Dado un usuario autenticado,
 Cuando envía un POST con campos válidos,
 Entonces responde 201 Created.
Escenario 2: campos obligatorios
 Dado action vacío o email inválido,
 Entonces responde 400 y la lista de errores.
```

201 Created
el camino feliz, verificable

400 Bad Request
el error esperado, también verificable

Gherkin (*Dado / Cuando / Entonces*) convierte el requisito en una prueba que se puede ejecutar.

Paso 4 · Clarify: preguntar antes de programar

Prompt de Clarify

```
Revisa la especificación
.agents/specs/feat-audit-spec.md a la luz
de AGENTS.md y .agents/memory/.

Hazme de 3 a 5 preguntas sobre casos borde,
índices en base de datos o manejo de
errores que no estén claros.

No generes código todavía.
```

¿Y si falla la base de auditoría?
¿se aborta la transacción o se registra aparte?

¿Se filtrará la búsqueda?
por rango de fechas o por email de usuario

Resolver las dudas en el Markdown **antes de programar** ahorra horas de refactorización.

Paso 5 · Plan: el cómo técnico

.agents/plans/feat-audit-plan.md

```
# Plan técnico: módulo de auditoría

## Componentes
1. Entidad: AuditLog (tabla audit_logs,
 clave UUID, índices created_at y
    user_email).
2. DTOs: CreateAuditLogRequest y
 AuditLogResponse (Records).
3. Repositorio: extiende JpaRepository.
4. Servicio: @Transactional con
 Propagation.REQUIRES_NEW.
5. Controlador: /api/v1/audit-logs con
    @PreAuthorize("hasRole('ADMIN')").

## Decisiones
- REQUIRES_NEW: el log se guarda aunque la
 operación del usuario falle después.
```

Capas
Controller, Service, Repository y DTOs

REQUIRES_NEW
la respuesta a una duda de Clarify

Índices
consultas rápidas para el administrador

Paso 6 · Tasks: tareas atómicas

.agents/tasks/feat-audit-tasks.md

```
# Checklist: módulo de auditoría

- [ ] Tarea 1: persistencia
  Entidad AuditLog y DDL con índice.
  Verificación: compila y crea la tabla.

- [ ] Tarea 2: DTOs y validación
  @NotBlank y @Email en el request.
  Verificación: pruebas unitarias pasan.

- [ ] Tarea 3: repositorio y servicio
  Verificación: prueba de integración.

- [ ] Tarea 4: controlador y seguridad
  Verificación: Postman 201 y 400.
```

Pequeñas y numeradas
una unidad de trabajo cada una

Con criterio de verificación
compila, prueba unitaria, Postman

Independientes
se ejecutan y revisan por separado

Paso 7 · Implement: una tarea a la vez

Prompt de ejecución controlada

```
Lee AGENTS.md, .agents/plans/feat-audit-plan.md
y la Tarea 1 de .agents/tasks/feat-audit-tasks.md.

Implementa ÚNICAMENTE la Tarea 1
(entidad AuditLog y repositorio).
No toques controladores ni servicios.

Cuando termines, muestra el código creado
y detén tu ejecución.
```

1. El agente implementa
solo la tarea autorizada

2. Tú compilas y verificas
sin errores, criterio cumplido

3. Marcas [x] y autorizas la siguiente
el ciclo se repite

Universidad ICESI

Sesión 2

Flutter con Supabase

Universidad ICESI

Postgres

API REST

Seguridad

Usuarios

¿Qué es Supabase?

Una plataforma de **backend como servicio** de código abierto, construida sobre **Postgres**.

- Los datos viven en **tablas SQL**, no en documentos
- Una API REST nace sola de cada tabla
- La seguridad se define con políticas en la base

> Se presenta como la alternativa abierta a Firebase.

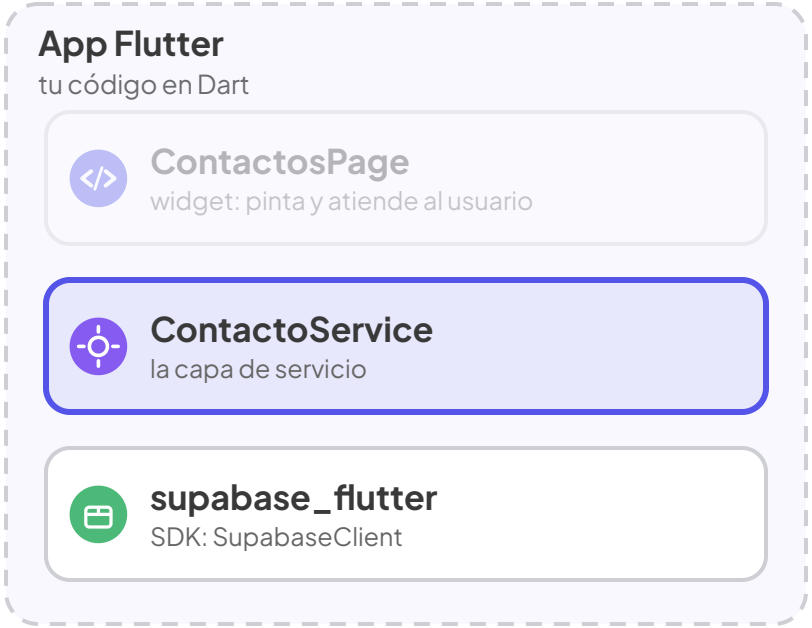
Universidad icesi

Todo lo que una app necesita, alrededor de Postgres



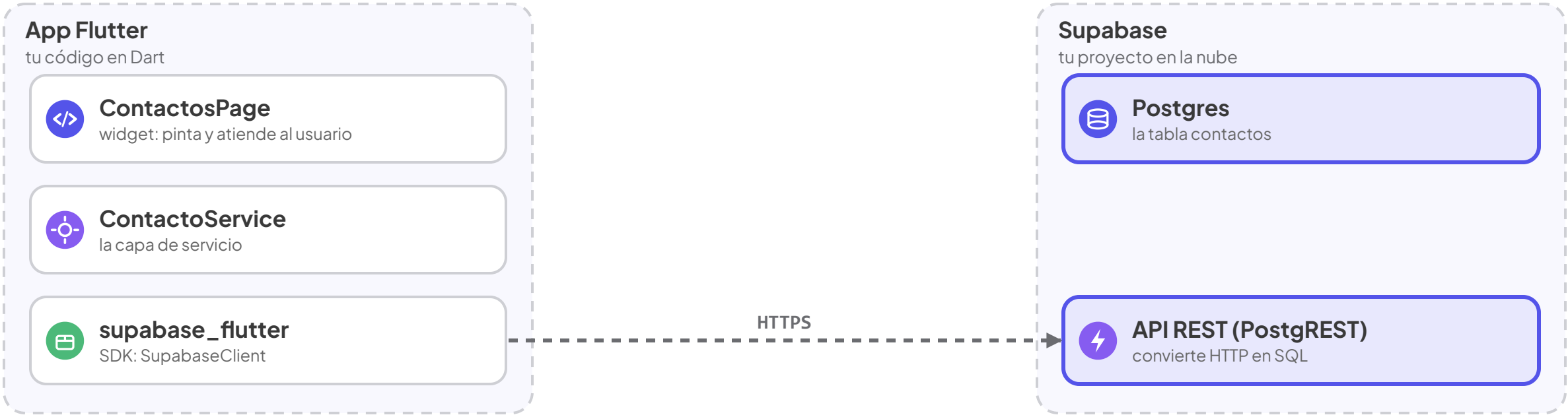
Hoy usamos la **base de datos**, su **API REST** y las **políticas RLS**. Autenticación, Storage y Realtime usan el mismo SDK.

Arquitectura: la interfaz no habla con la base de datos



La página dibuja y atiende al usuario. El **servicio** es la única clase que conoce a Supabase, a través del SDK **supabase_flutter**.

Al otro lado: la API de Supabase y Postgres



Cada proyecto de Supabase expone una **API REST generada automáticamente** (PostgREST) a partir de tus tablas. No escribes ni despliegas ningún servidor.

La petición: de la página a Postgres



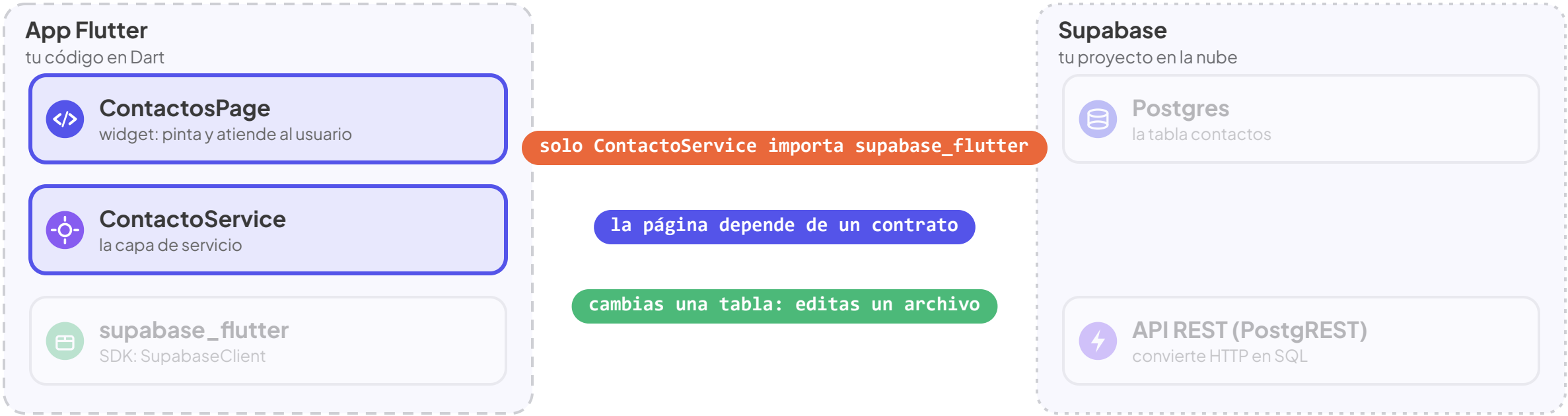
La página pide `listar()`; el servicio usa el SDK; el SDK envía una petición HTTPS; PostgREST la traduce a **SQL** y consulta Postgres.

La respuesta vuelve por el mismo camino



Postgres responde con filas; la API las envía como **JSON con código 200**; el SDK las entrega como datos de Dart y la página se redibuja.

Por qué una capa de servicio



La interfaz depende de un contrato (`listar()`, `crear()`, `borrar()`), no de una tecnología. Cambiar de tabla, de proveedor o probar con datos falsos **no toca los widgets**.

Instalar el SDK de Supabase

Terminal y pubspec.yaml

```
flutter pub add supabase_flutter

# pubspec.yaml (se agrega solo)
dependencies:
  supabase_flutter: ^2.18.3
```

supabase_flutter
el paquete oficial en pub.dev

SupabaseClient
guarda la URL y la llave y arma las peticiones

Supabase.instance.client
disponible en toda la app tras inicializar

Credencial 1 · La llave publishable



En el panel: **Project Settings → API Keys → Publishable key**. Empieza por `sb_publishable_` ; cópiala con el botón de la derecha.

Credencial 2 · La URL del proyecto



En **Integrations → Data API → API URL**. Al SDK le entregas **solo la base** (`https://tu-proyecto.supabase.co`), sin `/rest/v1/`.

Universidad ICESI

Dos llaves, dos destinos

Publishable (sb_publishable_…)

- **Pública por diseño**
  va en la app
- **Por sí sola no da acceso a nada**
  lo decide RLS en cada tabla
- **Antes se llamaba anon key**
  eyJ…, la misma idea

Secret (sb_secret_…)

- **Se salta RLS**
  acceso total a los datos
- **Solo para servidores**
  funciones, workers, backend
- **Nunca en la app ni en el repositorio**

Una app móvil solo debe llevar la llave **publishable**.

Guardar las credenciales fuera del código

env.json, .gitignore y terminal

```
{
  "SUPABASE_URL": "https://tu-proyecto.supabase.co",
  "SUPABASE_PUBLISHABLE_KEY": "sb_publishable_xxx"
}

# .gitignore
env.json

# ejecutar la app
flutter run --dart-define-from-file=env.json
```

env.json
URL y llave fuera del código

.gitignore
el archivo no se sube al repositorio

--dart-define-from-file
las pasa al compilar, sin paquetes extra

Inicializar Supabase una sola vez

lib/main.dart

```dart
const supabaseUrl =
    String.fromEnvironment('SUPABASE_URL');
const supabaseKey =
    String.fromEnvironment('SUPABASE_PUBLISHABLE_KEY');

Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Supabase.initialize(
    url: supabaseUrl,
    publishableKey: supabaseKey,
  );
  runApp(const MyApp());
}
```

String.fromEnvironment
lee lo que pasó --dart-define

await Supabase.initialize
asíncrono: abre el cliente y restaura la sesión

ensureInitialized()
necesario porque main es async

Antes de 2.13 el parámetro se llamaba `anonKey` .

La tabla contactos y su seguridad

SQL Editor de Supabase

```sql
create table public.contactos (
  id bigint generated always as identity
     primary key,
  nombre text not null,
  email text not null,
  created_at timestamptz default now()
);

alter table public.contactos
  enable row level security;

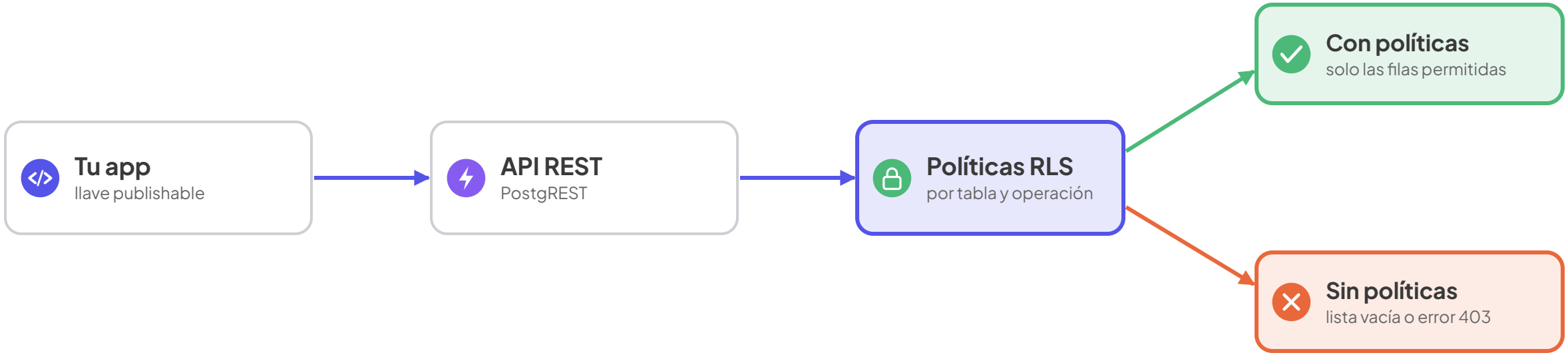
-- solo para el ejercicio de clase
create policy "demo: leer" on public.contactos
  for select to anon using (true);
```

id generado por Postgres
la app nunca lo envía al crear

enable row level security
sin políticas, la API no deja leer ni escribir

Políticas demo abiertas
en una app real se restringen a usuarios

RLS decide qué filas se ven



La API es pública: cualquiera con tu URL y tu llave puede llamarla. **Lo que protege los datos son las políticas**, no esconder la llave.

El modelo: una fila hecha clase

lib/models/contacto.dart

```dart
class Contacto {
  final int? id;
  final String nombre;
  final String email;

  const Contacto({this.id,
    required this.nombre, required this.email});

  factory Contacto.fromMap(Map<String, dynamic> m)
    => Contacto(id: m['id'] as int,
        nombre: m['nombre'] as String,
        email: m['email'] as String);

  Map<String, dynamic> toMap() =>
    {'nombre': nombre, 'email': email};
}
```

fromMap
convierte el Map del SDK en un Contacto

toMap
convierte el Contacto en datos para guardar

id nullable
Postgres lo genera al insertar

El servicio: un método por operación

lib/services/contacto_service.dart

```dart
class ContactoService {
  final _db = Supabase.instance.client;
  static const _tabla = 'contactos';

  Future<List<Contacto>> listar() async {
    final filas = await _db.from(_tabla)
        .select().order('nombre');
    return filas.map(Contacto.fromMap).toList();
  }

  Future<void> crear(Contacto c) =>
      _db.from(_tabla).insert(c.toMap());

  Future<void> borrar(int id) =>
      _db.from(_tabla).delete().eq('id', id);
}
```

Todo devuelve Future
hablar con Supabase es una operación de red

Solo ve Contacto
la interfaz nunca ve Map ni Supabase

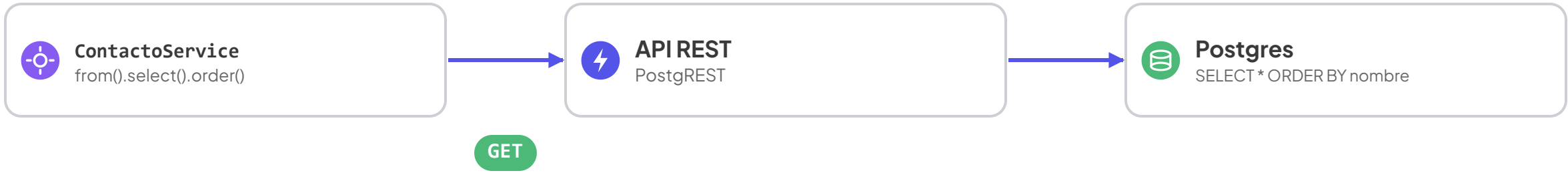
obtener y editar
igual de cortos: select().eq().single() y update().eq()

CRUD: cada consulta es una petición HTTP

Operación	Método del SDK	HTTP	SQL
Listar	`select().order()`	`GET /contactos?order=nombre.asc`	`SELECT … ORDER BY`
Detalle	`select().eq().single()`	`GET /contactos?id=eq.7`	`SELECT … WHERE`
Crear	`insert(datos)`	`POST /contactos + JSON`	`INSERT INTO`
Editar	`update(datos).eq()`	`PATCH /contactos?id=eq.7`	`UPDATE … WHERE`
Borrar	`delete().eq()`	`DELETE /contactos?id=eq.7`	`DELETE … WHERE`

PostgREST traduce la ruta y los parámetros de la URL a SQL. Entender esa traducción te permite depurar.

Leer: listar y ver el detalle

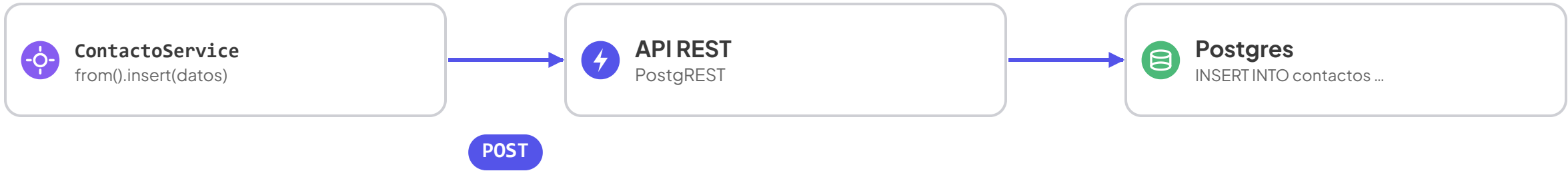


Petición. `GET /rest/v1/contactos?select=*&order=nombre.asc`

Respuesta. `200` con un arreglo JSON, una fila por contacto.

Para el detalle se agrega `.eq('id', 7).single()`: el filtro viaja en la URL (`?id=eq.7`) y `single()` pide **un objeto**; si no existe, lanza una `PostgrestException`.

Crear: insert es un POST

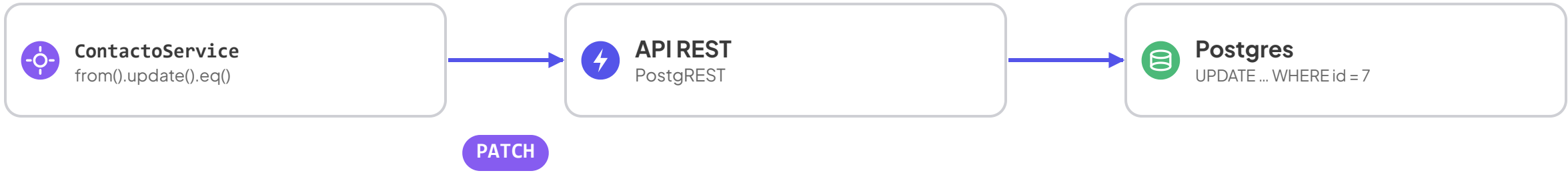


Petición. `POST /rest/v1/contactos`
Cuerpo. `{ "nombre": "Ana", "email": "ana@icesi.edu.co" }`

Respuesta. `201 Created`, sin cuerpo.

La fila nueva viaja como JSON. El `id` y `created_at` los genera Postgres. Para recibir la fila creada, encadena `.select().single()`.

Editar: update es un PATCH



Petición. `PATCH /rest/v1/contactos?id=eq.7`
Cuerpo. `{ "nombre": "Ana Pérez" }`

Respuesta. `204 No Content`: salió bien.

El cuerpo trae **solo las columnas que cambian** y el filtro de la URL elige la fila.

Borrar sin filtro: el error que más duele

delete().eq('id', 7)

`DELETE /contactos?id=eq.7`

Se borra 1 fila
la del id 7

`el filtro elige la fila`

delete() (sin eq)

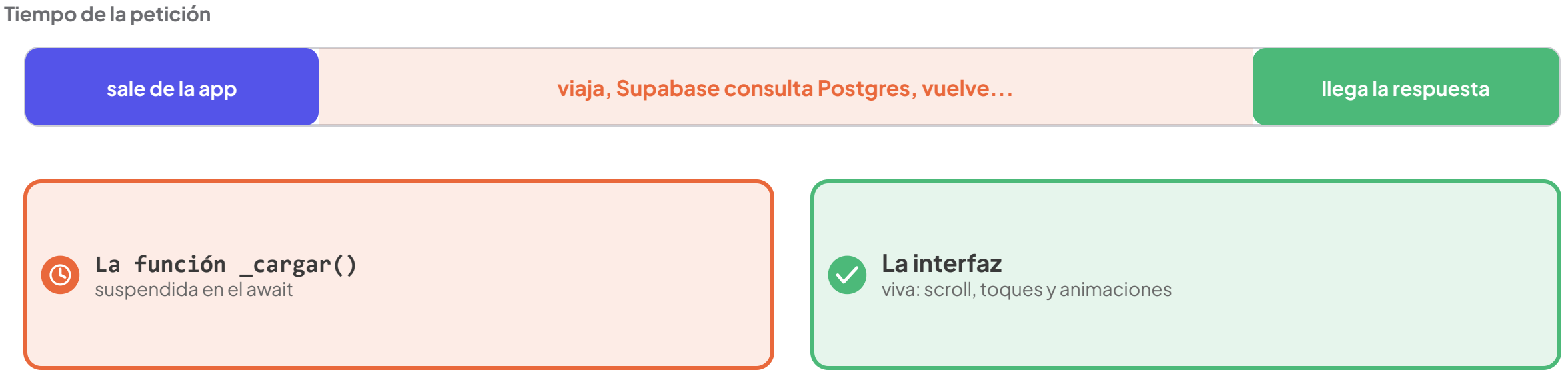
`DELETE /contactos`

Se borran todas las filas
las que la política permita borrar

`sin filtro afecta a todas`

En `update()` y `delete()`, **siempre encadena** `.eq(...)`. Sin filtro, la petición no distingue filas.

Esperar no significa congelarse



El `await` suspende **solo la función que espera**. La app sigue respondiendo, pero `build()` debe decidir qué dibujar mientras no hay datos.

Universidad ICESI

Tres estados, tres vistas

`_cargando`
la petición no ha vuelto

`indicador de carga`

`_error`
la petición falló

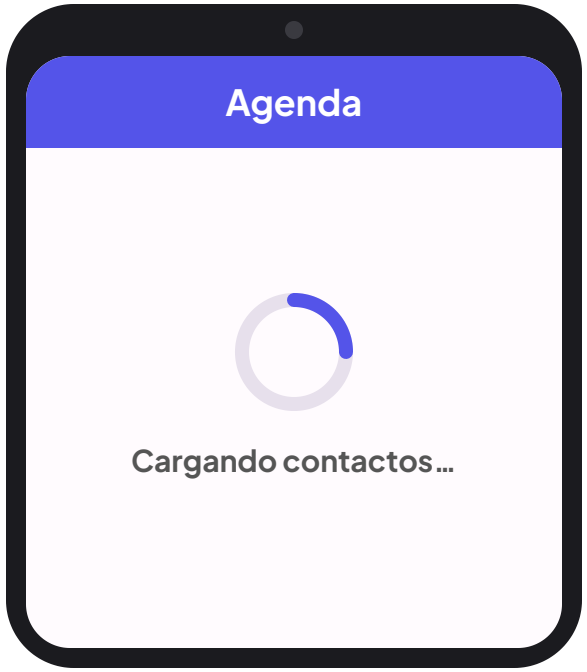
`mensaje y Reintentar`

`_contactos`
la petición volvió bien

`la lista`

El `State` de la página guarda dos banderas y los datos. `build()` **los lee en ese orden** y dibuja una sola cosa.

Cargando: la petición salió

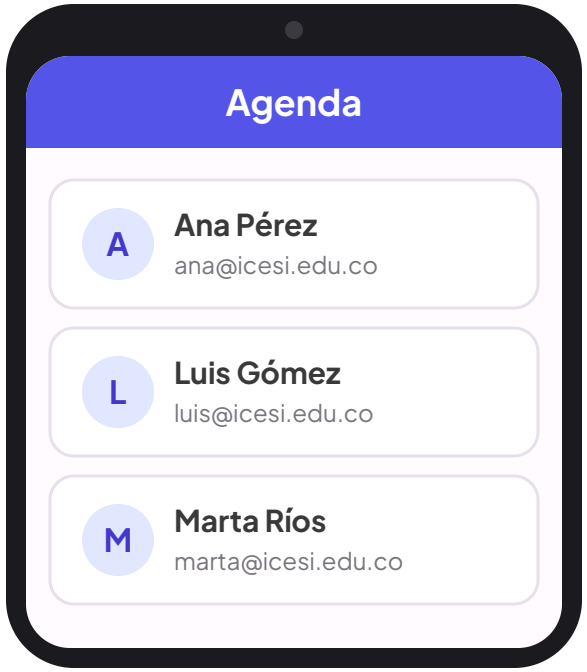


_cargar() en contactos_page.dart

```dart
Future<void> _cargar() async {
  setState(() {
    _cargando = true;
    _error = null;
  });
  try {
    final filas = await _servicio.listar();
    setState(() => _contactos = filas);
  } on PostgrestException catch (e) {
    setState(() => _error = e.message);
  } catch (e) {
    setState(() => _error = 'Sin conexión');
  } finally {
    setState(() => _cargando = false);
  }
}
```

_cargando arranca en true: el primer build() ya dibuja el indicador, no una lista vacía.

Éxito: llegan los datos



_cargar() en contactos_page.dart

```dart
Future<void> _cargar() async {
  setState(() {
    _cargando = true;
    _error = null;
  });
  try {
    final filas = await _servicio.listar();
    setState(() => _contactos = filas);
  } on PostgrestException catch (e) {
    setState(() => _error = e.message);
  } catch (e) {
    setState(() => _error = 'Sin conexión');
  } finally {
    setState(() => _cargando = false);
  }
}
```

Se guardan en _contactos y `finally` apaga el indicador: build() pasa a la tercera rama.

Error de la API: PostgrestException



_cargar() en contactos_page.dart

```dart
Future<void> _cargar() async {
  setState(() {
    _cargando = true;
    _error = null;
  });
  try {
    final filas = await _servicio.listar();
    setState(() => _contactos = filas);
  } on PostgrestException catch (e) {
    setState(() => _error = e.message);
  } catch (e) {
    setState(() => _error = 'Sin conexión');
  } finally {
    setState(() => _cargando = false);
  }
}
```

Supabase respondió con un error (tabla inexistente, RLS, dato inválido). El mensaje llega en `e.message`.

Sin conexión: el catch genérico



_cargar() en contactos_page.dart

```dart
Future<void> _cargar() async {
  setState(() {
    _cargando = true;
    _error = null;
  });
  try {
    final filas = await _servicio.listar();
    setState(() => _contactos = filas);
  } on PostgrestException catch (e) {
    setState(() => _error = e.message);
  } catch (e) {
    setState(() => _error = 'Sin conexión');
  } finally {
    setState(() => _cargando = false);
  }
}
```

Sin red la petición ni sale y el SDK lanza otra excepción: la atrapa el `catch` genérico.

build() elige una vista; mounted evita errores

contactos_page.dart

```dart
Widget _cuerpo() {
  if (_cargando) {
    return const Center(
      child: CircularProgressIndicator());
  }
  if (_error != null) {
    return ErrorVista(_error!,
        onReintentar: _cargar);
  }
  return ListView(children: [...]);
}

// después de cada await:
if (!mounted) return;
```

Primero cargando
sin datos todavía

Luego el error
con su botón para reintentar

if (!mounted) return
si el usuario salió, no llames a setState

`finally` siempre apaga la bandera: sin él, un error deja la app cargando para siempre.

Errores frecuentes y qué revisar

Síntoma	Causa probable
La lista llega vacía aunque hay filas	RLS activado y sin política de select
new row violates row-level security policy	falta la política de insert
Editar o borrar no hace nada y no da error	falta política de update/delete, o el filtro no coincide
Could not find the table …	nombre de tabla mal escrito en from()
Invalid API key o 401	URL o llave equivocadas, o no se pasó env.json
La app se queda cargando para siempre	falta el finally que apaga _cargando