# DB Seçimi

## PROMPT

Saas Projesi için PostgreSQL RDBMS kullanmayı düşünüyorum. Bu fikri değerlendirir misin?

## YANIT

PostgreSQL, bir SaaS projesi için gayet güçlü ve mantıklı bir tercih olabilir, ancak “en iyi seçim” olup olmadığı senin **iş yüküne, ölçeklenme planına, ekibinin tecrübesine** ve **mimari hedeflerine** bağlı.

Avantajları:

1. **Stabilite ve Olgunluk** – PostgreSQL onlarca yıldır geliştiriliyor, kararlı ve endüstride güvenilir.
2. **Zengin Özellik Seti** – JSONB desteği, CTE’ler, window functions, full-text search, spatial (PostGIS) gibi modern özellikleri var. SaaS’ta farklı veri tipleri ve raporlama ihtiyaçları için çok uygun.
3. **ACID Uyumu** – Veri bütünlüğü kritikse (abonelikler, faturalar, kullanıcı verileri) sağlam.
4. **Uzantılar** – PostGIS, pg\_partman, TimescaleDB gibi SaaS’ta işinize yarayabilecek güçlü eklentiler var.
5. **Bulut Uyumlu** – AWS RDS, GCP Cloud SQL, Azure Database gibi yönetilen servislerde kolayca çalışıyor.

Dikkat edilmesi gerekenler:

* **Yatay Ölçeklenme** – PostgreSQL tek başına yatay olarak çok kolay ölçeklenmez. Okuma yükünü dağıtmak için read replica kullanılır, ama yazma ölçeklenmesi için sharding veya Citus gibi çözümler gerekebilir.
* **Bakım** – Index yönetimi, vacuum/auto-vacuum tuning, query optimizasyonu gibi operasyonel işler gerekebilir.
* **Maliyet** – Yönetilen servisler (RDS, Aurora PostgreSQL) başlangıçta uygun olabilir ama veri büyüdükçe maliyet artabilir.

Alternatif olarak:

* **MySQL/MariaDB**: Daha basit kurulum ve bakım, ama özellik seti PostgreSQL kadar geniş değil.
* **NewSQL çözümleri** (CockroachDB, YugabyteDB): PostgreSQL benzeri SQL ama doğal yatay ölçeklenme.
* **NoSQL** (MongoDB, DynamoDB): Eğer veri ilişkileri zayıfsa veya tamamen esnek şema isteniyorsa düşünülebilir.

📌 SaaS için PostgreSQL’i seçmek mantıklı çünkü **çoğu SaaS uygulamasında güçlü veri tutarlılığı, esnek sorgulama ve güvenilirlik** gerekiyor. Eğer çok büyük ölçek (milyonlarca aktif kullanıcı, global dağıtım) planlıyorsan, PostgreSQL’i **Citus gibi bir dağıtık eklentiyle** veya **yönetilen bir versiyonla** kullanmak daha uzun ömürlü olur.

# PostgreSQL Base&Auditable

## PROMPT

Postgresqlde oluşturacağım varlıklar için snake\_case kullanmayı düşünüyorum.   
Ayrıca her tablo için;

* id (BIGSERIAL) ,
* public\_id (UUID) ,
* is\_active (boolean)
* created\_by (integer) (ForeignKey => user.id)
* created\_date (timestamp with time zone)
* updated\_by (integer) (ForeignKey => user.id)
* updated\_date (timestamp with time zone)
* deleted\_by (integer) (ForeignKey => user.id)
* deleted\_date (timestamp with time zone)
* tenant\_id (integer) (ForeignKey => tenant.id) “subs” şemasında “Tenant, Plans, Futures” tablolar hariç her tabloda (public şemasındaki countries tablosu da hariç)

Yukarıdakiler soft-delete, denetlenebilirlik, güvenlik vb. Açısından uygun mu?

## YANIT

Senin düşündüğün yapı, **çok kiracılı (multi-tenant)** bir SaaS projesi için oldukça olgun bir “audit + soft delete” standardı gibi görünüyor.  
Avantajlarını, dikkat etmen gereken noktaları ve birkaç iyileştirme önerisini şöyle özetleyebilirim:

**1. snake\_case Kullanımı**

* **Artı:** PostgreSQL ekosisteminde ve SQL standartlarında yaygın, okunabilirliği yüksek.
* **Eksi:** Kod tarafında (özellikle ORM’lerde) camelCase kullanıyorsan mapping yapman gerekir.
* **Öneri:** ORM mapping ayarlarını baştan doğru kurarsan bu sorun olmaz.

**2. Ortak Alanlar (Audit + Soft Delete)**

**Kapsam:**

* id (BIGSERIAL) → PK olarak iyi. Ama ileride sharding düşünürsen *snowflake ID* tarzı dağıtık ID’lere geçmek zor olabilir. UUID de düşünebilirsin.
* public\_id (UUID) → Harici API’lerde kullanmak için çok iyi (güvenlik + tahmin edilemezlik).
* is\_active (boolean) + deleted\_date → Soft delete için mantıklı. Ancak is\_active ile deleted\_date aynı amaca hizmet edebilir. Çifte kontrol gereksiz karmaşıklık yaratabilir.
* created\_by / updated\_by / deleted\_by → Audit trail açısından çok faydalı.
* created\_date / updated\_date / deleted\_date → Zaman damgaları, zaman dilimi farklarını doğru yönetmek için timestamp with time zone seçimin doğru.
* tenant\_id → Multi-tenancy için şart.

**3. Potansiyel İyileştirmeler**

1. **Soft Delete Kolaylığı**
   * is\_active + deleted\_date yerine sadece deleted\_date kullanmak daha net.
   * Sorgularda "WHERE deleted\_date IS NULL" filtrelemesi standart hale gelir.
2. **Multi-Tenant Güvenlik**
   * tenant\_id ForeignKey'inin **her sorguda otomatik filtrelenmesi** için ORM’de global scope / Row Level Security (RLS) kullanabilirsin.
   * PostgreSQL RLS, SaaS projelerinde kiracılar arası veri sızmasını engellemekte altın standarttır.
3. **created\_by / updated\_by / deleted\_by ForeignKey’leri**
   * Bunlar null olabilir (ör. sistem otomasyonu kayıt oluşturursa).
   * NOT VALID constraint ile başlayıp sonradan veri oturunca doğrulama ekleyebilirsin.
4. **UUID Performansı**
   * public\_id için uuid\_generate\_v4() iyi, ama index performansı için uuid-ossp veya gen\_random\_uuid() ile üretilmiş *v4* UUID’leri **clustered olmayan index** olarak tutmanı öneririm.
5. **Ortak Alanlar İçin Inheritance / Composite Type**
   * PostgreSQL’de **table inheritance** veya **composite type** kullanarak bu alanları tüm tablolara kolayca ekleyebilirsin.
   * Örneğin:
   * CREATE TABLE base\_audit\_fields (
   * id BIGSERIAL PRIMARY KEY,
   * public\_id UUID NOT NULL DEFAULT gen\_random\_uuid(),
   * created\_by INT,
   * created\_date TIMESTAMPTZ DEFAULT now(),
   * updated\_by INT,
   * updated\_date TIMESTAMPTZ,
   * deleted\_by INT,
   * deleted\_date TIMESTAMPTZ
   * );

Sonra:

CREATE TABLE orders (

tenant\_id INT NOT NULL,

product\_id INT NOT NULL

) INHERITS (base\_audit\_fields);

✅ **Sonuç:** Yapın mantıklı, özellikle SaaS ölçeğinde audit ve soft delete için sağlam bir temel oluşturmuşsun.  
🔍 Ama küçük dokunuşlarla:

* is\_active yerine sadece deleted\_date kullanmak,
* Row Level Security’yi baştan kurgulamak,
* UUID indeks performansını düşünmek,  
  gelecekte bakım ve ölçeklenmede sana ciddi kolaylık sağlar.

# Primary Key Seçimi ve Tenat ID

## PROMPT 1

Tablolarda id alanını integer mi yapmalıyım, guid mi?

## YANIT 1

Bu, SaaS mimarilerinde çok sık tartışılan bir konu ve tek doğru yok — ama PostgreSQL tarafında **integer (serial/bigserial)** ile **UUID (GUID)** arasındaki farkları hem teknik hem iş mantığı açısından değerlendirelim.

**1. Integer (SERIAL/BIGSERIAL)**

**Artıları**

* **Daha hızlı**: 64-bit integer karşılaştırma, index ve join işlemlerinde UUID’den daha hızlıdır (~2–3 kat daha az CPU/memory kullanır).
* **Daha küçük index boyutu**: bigserial indexi, UUID indexinden %40–50 daha küçük olur → RAM verimliliği.
* **Okunabilir**: ID değerleri küçük sayılarla artar → debug, log, manuel veri incelemede daha anlaşılır.
* **Otomatik artış**: serial/bigserial veya GENERATED ALWAYS AS IDENTITY ile kolay yönetim.

**Eksileri**

* **Tahmin edilebilir**: ID dizileri art arda olduğu için dış sistemlerden veya saldırganlardan “kaç kayıt var” veya “tahmin edilebilir URL” riski.
* **Merge/replikasyon zor**: Birden fazla node’dan veri geldiğinde ID çakışma riski → sequence ayarı gerekebilir.

**2. UUID (GUID)**

**Artıları**

* **Dağıtık sistemler için ideal**: Birden çok veri kaynağında ID üretildiğinde çakışma ihtimali yok.
* **Tahmin edilemez**: ID değerleri rastgele göründüğü için dışarıya bilgi sızdırmaz.
* **Kopyalama/taşıma kolay**: Farklı ortamdan gelen verileri kolayca merge edebilirsin.

**Eksileri**

* **Daha büyük boyut (16 byte)**: Indexler ve foreign key ilişkileri daha fazla disk/RAM kaplar, query performansı düşebilir.
* **Okunabilirlik düşük**: d90b35c1-1423-4c0a-bcd2-b9289b0d8a3f gibi ID’ler loglarda ve debug’da zor okunur.
* **Random insert penalty**: Random UUID kullanılırsa index sıralaması bozulur → insert’lerde page split maliyeti artar. (UUIDv4 yerine UUIDv1/v7 ile bu azaltılabilir.)

**3. SaaS bağlamında önerim**

* **Tek DB, merkezi sequence**: Eğer tüm müşteriler aynı PostgreSQL instance’ında (multi-tenant DB) ve ID’leri sadece dahili sistem kullanıyorsa → **BIGSERIAL** en verimli seçim.
* **Dağıtık/çoklu node veya offline client senkronizasyon**: Eğer ID’ler farklı sistemlerde üretilecekse, veya offline mod desteklenecekse → **UUID** tercih et.
* **Güvenlik endişesi** (ID tahmin edilmesin) varsa: BIGSERIAL kullanıp “public” alanlarda *public\_id* gibi ayrı bir UUID alanı tutabilirsin.

💡 **Best practice hybrid model**:  
PostgreSQL’de PK olarak bigserial tut, ama uygulamada dış dünyaya **UUID** yansıt.

* Avantaj: DB performansı iyi kalır, dış API’lerde güvenli, tahmin edilemez ID olur.
* Örnek:

CREATE TABLE users (

id BIGSERIAL PRIMARY KEY,

public\_id UUID NOT NULL DEFAULT gen\_random\_uuid(),

name TEXT NOT NULL

);

CREATE UNIQUE INDEX idx\_users\_public\_id ON users(public\_id);

## PROMPT 2

SaaS’ta tek DB var, tenantid ile müşterileri ayırıyorum

## YANIT 2

Doğru, BIGSERIAL PostgreSQL’e özel bir **şeker kaplama** (syntactic sugar).  
Ama bu seni tamamen PostgreSQL’e zincirlemez, çünkü BIGSERIAL aslında **BIGINT + sequence** tanımıdır.

**1. BIGSERIAL Nedir?**

PostgreSQL’de:

id BIGSERIAL PRIMARY KEY

şu anlama gelir:

id BIGINT NOT NULL DEFAULT nextval('id\_seq'),

PRIMARY KEY (id)

ve ayrıca PostgreSQL otomatik olarak id\_seq isimli sequence oluşturur.

**2. Taşınabilirlik (Portability) Meselesi**

* **MySQL / MariaDB** → BIGINT AUTO\_INCREMENT ile aynı işlevi görür.
* **SQL Server** → BIGINT IDENTITY(1,1) kullanılır.
* **Oracle** → NUMBER(19) + sequence + trigger veya IDENTITY (12c+) ile yapılır.

Yani **mantık olarak** çoğu veritabanı yönetim sisteminde karşılığı var, sadece yazım şekli değişiyor.

**3. Gelecekte DB değiştirme planı varsa**

Eğer vendor lock-in riskini azaltmak istiyorsan:

* BIGSERIAL yerine **standart SQL-92 GENERATED ALWAYS AS IDENTITY** kullanabilirsin.
* PostgreSQL 10+ sürümlerinde bu destekleniyor ve diğer veritabanlarında da benzer şekilde çalışıyor.

Örnek (taşınabilir versiyon):

id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY

Bu tanım:

* PostgreSQL → Sequence kullanarak çalışır
* SQL Server → IDENTITY olarak dönüşebilir
* MySQL → AUTO\_INCREMENT olarak dönüşebilir

**4. SaaS bağlamında tavsiye**

* **Kısa vadede** PostgreSQL kullanacaksan BIGSERIAL hem kolay hem hızlı.
* **Orta-uzun vadede** DB taşınma ihtimalin varsa GENERATED AS IDENTITY daha güvenli bir seçim.
* UUID’yi tamamen zorunlu kılan bir durum yoksa, sayısal ID + tenant\_id yapısı yine en performanslı model.