

PostgreSQL Data Types

INTEGER/BIGINT/BIGSERIAL, VARCHAR/TEXT, BOOLEAN, ve DATE/TIMESTAMP /TIMESTAMPTZ -- her birinin bu projenin kendi gerçek Topic/TopicTranslation/Question entity'lerindeki karşılık gelen Java alan türüne (Long/String/boolean/LocalDateTime) nasıl eşlendiği. PostgreSQL Foundations kategorisinin 4.'sü.

Başlangıç 20 dk

Oluşturulma tarihi: 5 Ekim 2026

Çevrimiçi oku: <https://www.learnforgex.com/tr/topics/postgresql-data-types>

"Databases, Schemas, Tables, and Basic SQL Syntax" dersi, bu projenin kendi CREATE TABLE ifadelerindeki BIGSERIAL, VARCHAR(255), ve TIMESTAMP 'i, her birinin gerçekte neyi garanti ettiğini ya da Java'da nasıl bir Long, bir String, ya da bir LocalDateTime e dönüştüğünü sormadan okudu. Bu ders bunu cevaplıyor -- uydurma kolonlarla değil, bu projenin kendi gerçek kolonlarıyla.

Sayısal Türler: integer, bigint, ve Bu Projenin BIGSERIAL Seçimi

PostgreSQL'in her gün kullanılan iki tamsayı türü INTEGER (4 byte, yaklaşık $\pm 2,1$ milyar) ve BIGINT 'tir (8 byte, yaklaşık $\pm 9,2$ kentilyon). V1__init_schema.sql 'deki her id kolonu için kullanılan BIGSERIAL, ayrı bir depolama türü DEĞİLDİR -- bir sonraki değeri üreten otomatik oluşturulmuş bir sequence artı BIGINT 'tir, ve bu tam olarak bu projenin kendi Topic.id sindeki GenerationType.IDENTITY 'nin arkasındaki şeydir:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

BIGINT / BIGSERIAL, Java'nın Long ına eşlenir, INTEGER / SERIAL ise Integer a -- bu projenin topic.estimated_minutes kolonu (düz INTEGER, otomatik üretim yok) tam olarak Topic.estimatedMinutes in neden Long değil Integer olarak tanımlandığının nedenidir: küçük, elle girilmiş bir sayı, asla bir identity kolonu değil, bu yüzden BIGINT 'in ekstra menzili hiçbir şey kazandırmaz. Bir tablo asla milyar satıra yaklaşmayacak olsa bile her primary key için BIGINT / BIGSERIAL seçmek, korunmaya değer savunmacı bir varsayılandır -- bir primary key'in türünü, foreign key'ler ona referans verdikten sonra değiştirmek, BIGINT 'in baştan maliyet ettiği birkaç ekstra byte'tan çok daha yıkıcıdır.

Metin Türleri: varchar vs. text

PostgreSQL'in üç string türü var, ve aradaki fark görüldüğünden daha küçük: `VARCHAR(n)`, `n` karakterlik bir maksimum uzunluk zorunlu kılar; uzunluksuz `VARCHAR` ile `TEXT`, işlevsel olarak özdeş, sınırsız uzunlukta string'lerdir. İçeride, PostgreSQL'ün de aynı şekilde saklar ve uzunluğu sınırlı bir `VARCHAR` a kıyasla `TEXT` e hiçbir performans cezası uygulamaz -- `TEXT` in daha yavaş, ayrı saklanan bir tür olduğu bazı diğer veritabanlarının aksine.

Bu projenin kendi kolonları her iki seçimin de bilinçli olarak kullanıldığını gösteriyor: `topic.slug VARCHAR(255)` (`v1__init_schema.sql` 'den), URL'lerde ve dosya yollarında kullanılan, sınırsız bir uzunluğun bir özellik değil bir hata olacağı bir değeri sınırlar; buna karşılık `topic_translation.summary`, Java'da açık bir `columnDefinition` ile eşlenir:

```
@Column(columnDefinition = "TEXT")
private String summary;
```

Bir ders özetinin doğal bir uzunluk tavanı yoktur, bu yüzden `TEXT` dürüst seçimdir -- daha sonra çarpılacak keyfi bir sınır yok. İkisi de Java'nın `String` ine eşlenir; `columnDefinition` ı ya da migration'ın kendisini kontrol etmeden, türün kendisi tek başına hangisine baktığınızı asla söylemez.

boolean

`BOOLEAN`, tam olarak `true`, `false`, ya da `NULL` saklar -- bazı veritabanlarının aksine, `0 / 1` tamsayı ikamesi yoktur. Bu projenin `topic_translation.published` kolonu, gerçek, düz bir `BOOLEAN NOT NULL` 'dur, Java'nın primitive `boolean` ına doğrudan eşlenir:

```
@Column(nullable = false)
private boolean published;
```

Buradaki `NOT NULL` in pratikte isteğe bağlı OLMADIĞINA dikkat et -- Java'da bir primitive `boolean` alan asla `null` tutamaz, bu yüzden kolon buna izin verseydi, veritabanından okunan bir `NULL` değer gidebileceği geçerli hiçbir yer olmazdı. Bu, genel olarak hatırlanmaya değer bir desenin ilk, somut bir görünümü: bir `NOT NULL` kolon ile null-olamayan bir Java türü (kutulanmış bir `Boolean` ya da `Integer` in aksine bir primitive) uyuşmalı, aksi hâlde bir satırı okumak uygulama mantığıyla hiçbir ilgisi olmayan şekillerde başarısız olabilir.

Tarih ve Zaman Türleri: date, timestamp, ve timestampz

PostgreSQL'in yaygın kullanılan üç zamansal türü var. `DATE`, yalnızca takvim tarihini saklar -- zaman bileşeni yok. `TIMESTAMP` (`TIMESTAMP WITHOUT TIME ZONE` nin kısaltması), hiçbir zaman dilimi eklenmemiş bir tarih ve saat saklar -- yazıldığı gibi, saf bir zaman noktası. `TIMESTAMPZ` (`TIMESTAMP WITH TIME ZONE`), PostgreSQL'in içeride her zaman UTC'ye normalleştirdiği,

bağlanan istemcinin hangi zaman diliminde olduğuna göre dönüştürdüğü bir zaman noktası saklar.

Bu projenin kendi `question` tablosu (daha sonraki bir migration'da, `V1__init_schema.sql` 'den sonra eklendi) düz `TIMESTAMP` kullanır:

```
ALTER TABLE question
  ADD COLUMN created_at TIMESTAMP NOT NULL DEFAULT now(),
  ADD COLUMN updated_at TIMESTAMP NOT NULL DEFAULT now();
```

`java.time.LocalDateTime` e eşlenir -- tesadüf değil, bu tür de kendi zaman dilimini taşımaz:

```
@Column(name = "created_at", nullable = false)
private LocalDateTime createdAt;
```

`TIMESTAMP` için `LocalDateTime` , ve (bu proje kullansaydı) `TIMESTAMPTZ` için `java.time.Instant` ya da `OffsetDateTime` , her yönde doğal eşleşmedir -- bunları karıştırmak (bir `TIMESTAMPTZ` kolonunu `LocalDateTime` e eşlemek gibi), PostgreSQL'in takip ettiği zaman dilimi bilgisini sessizce atar. Yukarıda görülen `now()` , mevcut transaction'ın zaman damgasını döndüren yerleşik bir PostgreSQL fonksiyonudur -- uygulama tarafından sağlanan bir değer değil, bu yüzden `created_at / updated_at` , herhangi bir Java kod yolundan bağımsız olarak doğrudan SQL ile eklenen satırlar için bile gerçek bir varsayılan alır.

psql ile Gerçek Bir Kolonun Türünü Okumak

`\d <table>` ("PostgreSQL'e Bağlanmak"ta ve "Databases, Schemas, Tables, and Basic SQL Syntax"ta zaten kullanılan), bir migration dosyası hiç açmadan bir kolonun gerçek türünü kontrol etmenin en hızlı yoludur:

```
learning=# \d topic_translation
               Table "public.topic_translation"
   Column      |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----
 id            | bigint                |           | not null |
 topic_id      | bigint                |           | not null |
 language      | character varying(5)  |           | not null |
 title         | character varying(255)|           | not null |
 summary       | text                  |           |          |
 seo_title     | character varying(255)|           |          |
 seo_description | character varying(500)|           |          |
 published     | boolean               |           | not null |
```

`psql` 'in `VARCHAR(255)` yerine `character varying(255)` , `BIGINT` yerine `bigint` raporladığına dikkat et -- PostgreSQL'in içsel tür adları küçük harflidir ve bazen bildirmek için kullanılan SQL anahtar kelimesinden daha uzundur; ikisi de birebir aynı türe işaret eder.

SQL Türünden Java Alanına: Hibernate Aradaki Boşluğu Nasıl Kapatır

Yukarıda gösterilen her eşleme -- `BIGINT ↔ Long` , `VARCHAR / TEXT ↔ String` , `BOOLEAN ↔ boolean` , `TIMESTAMP ↔ LocalDateTime` -- `Topic` , `TopicTranslation` , ya da `Question` 'da hiçbir açık tür-dönüşüm annotation'ı olmadan Hibernate tarafından otomatik olarak uygulanır. Bu, "JPA, Hibernate ve Spring Data JPA"nın zaten kapsadığı JDBC driver'ı ve Hibernate dialect katmanının doğrudan bir sonucudur -- bu ders yeni bir mekanizma tanıtıyor, yalnızca bu projenin gerçek Java alan türlerinin her birinin o katmanın diğer tarafında hangi PostgreSQL türüne karşılık geldiğini somut olarak adlandırıyor. Bu projenin çıkarıma güvenmek yerine bir eşlemeyi açık yaptığı tek yer tam olarak yukarıdaki `TEXT` durumu (`columnDefinition = "TEXT"`) -- çünkü Hibernate'in çıplak bir `String` alan için kendi varsayılanı `TEXT` değil, uzunluğu sınırlı bir `VARCHAR` dır, bu yüzden sınırsız bir kolon bilinçli olarak istenmelidir.

Yaygın Yanlış Anlamalar

" `VARCHAR(255)` , `TEXT` **ten daha hızlıdır.**" PostgreSQL'de değil -- içeride özdeş, özdeş performansla; uzunluk sınırı saf bir veri-bütünlüğü seçimidir, bir performans seçimi değil. "**Bir `BOOLEAN` kolon 0 ya da 1 olabilir.**" Hayır -- PostgreSQL'in `BOOLEAN` ı gerçek, üç-değerli bir türdür (`true / false / NULL`); `0 / 1` tamsayıdır, tamamen farklı bir tür, bazı istemci kütüphaneleri bunları gevşek bir girdi olarak kabul etse bile. "**`TIMESTAMP` , zaman dilimi işlemlerini içerir.**" Tam tersi -- düz `TIMESTAMP` 'in açıkça hiçbir zaman dilimi yoktur; `TIMESTAMPTZ` , `TIMESTAMP` in daha sık başvuru alanı olması rağmen, zaman dilimi olan taraftır.

Best Practices

- Bir tablo küçük kalacak olsa bile primary key'ler için varsayılan olarak `BIGINT / BIGSERIAL` kullan -- bu proje `v1__init_schema.sql` 'de bunu tek tip yapıyor, ve baştan anlamlı bir şeye mal olmazken daha sonra yıkıcı bir migration'ı önlüyor.
- Gerçekten sınırsız içerik için (`topic_translation.summary` gibi) `TEXT` i tercih et, ve `VARCHAR(n)` i gerçek, anlamlı bir üst sınırı olan değerler için (bir URL'de kullanılan `slug` gibi) sakla -- uzunluk sınırının bir tahmini değil bir iş kuralını ifade etmesine izin ver.
- Bir `NOT NULL` kolonu bir Java primitive'yle (`boolean` , `int` , `long`) ve null-olabilir bir kolonu karşılık gelen kutulanmış türle (`Boolean` , `Integer` , `Long`) eşleştir -- bu projenin `published` alanı, o kuralın primitive tarafının temiz bir örneği.
- Herhangi bir zaman dilimiyle karşılaşabilecek bir projedeki yeni bir zaman-takibi kolonu için düz `TIMESTAMP` yerine `TIMESTAMPTZ` i seç -- bu projenin `question.created_at / updated_at` için düz `TIMESTAMP` kullanımını, düşünmeden kopyalanacak bir desen değil, farkında olunmaya değer gerçek, mevcut bir trade-off olarak ele al.

Yaygın Hatalar

- Uzunluksuz `VARCHAR` ın `TEXT` ten farklı davrandığını varsaymak -- özellikle PostgreSQL'de davranmıyorlar; bu varsayımı başka bir veritabanından taşımak gereksiz mikro-optimizasyona yol açar.
- Bir `TIMESTAMP` kolonunu `LocalDateTime` e (ya da bir `TIMESTAMP` kolonunu `Instant` / `OffsetDateTime` e) eşlemek -- ikisi de derlenir ve genellikle hatasız çalışır bile, ama okuma ya da yazma anında zaman dilimi bilgisini sessizce kaybeder ya da uydurur.
- Bir Java primitive alanının null-olabilir bir kolondan okunan bir `NULL` ı tutamayacağını unutmak -- hata, gerçekte onu tetikleyen kolon tanımından uzakta, Hibernate/JDBC katmanında checked olmayan bir exception olarak yüzeye çıkar.
- "Yer kazanmak için" bir primary key için `INTEGER` seçmek, sonra tablo büyüdükçe gerçek menzil sınırlarına (ya da yıkıcı bir tür-genişletme migration'ına) çarpmak -- kazanılan byte'lar, özellikle bir `id` kolonu için riski nadiren haklı çıkarır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `INTEGER` / `BIGINT` , Java'nın `Integer` / `Long` ına eşlenir; `BIGSERIAL` , bu projenin kendi primary key'lerindeki `GenerationType.IDENTITY` 'nin arkasındaki, otomatik artan bir sequence artı `BIGINT` 'tir.
- `VARCHAR(n)` , uzunluksuz `VARCHAR` , ve `TEXT` , PostgreSQL'de özdeş saklanır ve hepsi Java'nın `String` ine eşlenir -- yalnızca uzunluk sınırı farklıdır, ve bu bir performans seçimini değil gerçek bir kısıtı ifade etmelidir.
- `BOOLEAN` , gerçek üç-değerli bir türdür (`true` / `false` / `NULL`), yalnızca kolon `NOT NULL` olduğunda bir Java primitive `boolean` ına temiz şekilde eşlenir.
- `TIMESTAMP` hiçbir zaman dilimi taşımaz ve `LocalDateTime` e eşlenir; `TIMESTAMP` taşır (içeride UTC'ye normalleştirilir) ve `Instant` / `OffsetDateTime` e eşlenir -- bu projenin kendi `question` tablosu düz `TIMESTAMP` kullanır.
- Burada gösterilen her SQL-Java tür eşlemesi, Hibernate'in JDBC dialect katmanı ("JPA, Hibernate ve Spring Data JPA"da zaten kapsanan) tarafından otomatik olarak işlenir -- bu projenin kendi entity'lerinde hiçbir yerde açık dönüşüm kodu yok.

Cheat Sheet

<code>INTEGER</code> / <code>SERIAL</code>	↔ <code>Integer</code>
<code>BIGINT</code> / <code>BIGSERIAL</code>	↔ <code>Long</code>
<code>VARCHAR(n)</code> / <code>TEXT</code>	↔ <code>String</code>
<code>BOOLEAN</code>	↔ <code>boolean</code> (<code>NOT NULL</code> ise) / <code>Boolean</code> (null-olabilirse)
<code>DATE</code>	↔ <code>LocalDate</code>
<code>TIMESTAMP</code>	↔ <code>LocalDateTime</code>
<code>TIMESTAMP</code> / <code>TZ</code>	↔ <code>Instant</code> / <code>OffsetDateTime</code>

```
\d <table>      -- bir tablonun gerçek kolon türlerini gör (psql)
```

Terimler Sözlüğü

- **BIGSERIAL**: PostgreSQL'in örtük olarak oluşturup yönettiği, otomatik artan bir sequence artı `BIGINT` -- `GenerationType.IDENTITY` 'nin arkasındaki SQL-seviyesi mekanizma.
- **TEXT**: PostgreSQL'de içeride `VARCHAR` la özdeş saklanan, sınırsız uzunlukta bir string türü.
- **TIMESTAMPTZ**: içeride UTC'ye normalleştirilen ve bağlanan istemci için dönüştürülen, zaman dilimi bilgisini takip eden bir zaman damgası türü.
- **now()**: mevcut transaction'ın zaman damgasını döndüren, doğrudan bir kolon `DEFAULT` 'u olarak kullanılabilen yerleşik bir PostgreSQL fonksiyonu.