PostgreSQL-Specific Data Types: UUID, JSON/JSONB, and Arrays

UUID (as an alternative to BIGSERIAL, with trade-offs), JSON vs. JSONB, querying JSONB with ->/->>/@>, and arrays (ANY/@>/unnest) -- honestly noting this project's real schema uses none of them, with illustrative examples grounded in this project's own domain. The 2nd lesson in the Advanced PostgreSQL category.

Advanced 30 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/postgresql-specific-data-types

---

"PostgreSQL Data Types" covered the types every relational database has some version of — integers, strings, booleans, timestamps. This lesson covers three that are genuinely PostgreSQL's own: `UUID`, `JSON` / `JSONB`, and arrays. Worth saying plainly up front: this project's own real schema doesn't use any of the three — every table so far has used `BIGSERIAL` primary keys and plain scalar columns. The examples below are illustrative extensions of this project's own domain, not existing columns, which is exactly what makes this a fair test of whether each type is genuinely warranted before reaching for it.

UUID: An Alternative to BIGSERIAL

"Constraints and Keys" already covered `BIGSERIAL PRIMARY KEY` as `BIGINT` plus an auto-incrementing sequence. `UUID` is a different primary-key strategy entirely — a 128-bit value, conventionally written as `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`, generated randomly (or from other inputs, depending on the UUID version) rather than handed out in sequence:

```
CREATE TABLE api_token (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    label VARCHAR(255) NOT NULL
);
```

`gen_random_uuid()` is PostgreSQL's own built-in function for generating a random (version 4) UUID as a column default — the direct UUID equivalent of `BIGSERIAL`'s implicit sequence, generating a fresh id with no application code involved. Nothing in this project's real schema needs this today — every `id` this project has ever generated (`topic.id`, `category.id`, and every other primary key) has been a `BIGSERIAL`, but a hypothetical table like `api_token`

above is exactly the kind of case where a UUID earns its place: an id that might need to be generated by a client before it's ever inserted, or one that shouldn't reveal how many tokens exist by its numeric value.

UUID vs. BIGSERIAL: Trade-offs

A `BIGSERIAL` id is compact (8 bytes), sequential (which keeps a `PRIMARY KEY` index's underlying B-tree — "Indexes and Query Performance with EXPLAIN," later in this category, covers what that means — efficiently ordered), and human-readable, but it's also predictable and reveals information: `topic.id = 42` tells anyone that at least 42 topics have been created, in an order they can guess. A `UUID` is larger (16 bytes), unpredictable, and can safely be generated by a client or a separate service before a row is ever inserted into the database — no round trip needed to learn "what id did this row get," which matters for a system with multiple services generating records independently, a scenario this project's own single-application architecture (documented in CLAUDE.md as a deliberate constraint, not something this course invents) has simply never needed. Neither is universally "better" — `BIGSERIAL` remains the right default for an internal primary key with no external-generation or information-hiding requirement, which is exactly why every table in this project still uses it.

JSON and JSONB

PostgreSQL has two JSON types: `JSON` stores the exact text submitted, byte for byte, re-parsing it every time it's queried; `JSONB` ("B" for binary) stores a parsed, more efficient internal representation, and is what nearly every real PostgreSQL schema reaches for — faster to query, supports indexing (which plain `JSON` doesn't), at the cost of not preserving things like the original key order or duplicate keys.

```
CREATE TABLE user_preference (
    user_id BIGINT PRIMARY KEY,
    settings JSONB NOT NULL DEFAULT '{}'::jsonb
);
```

A `JSONB` column holds structured, semi-variable data directly — no separate table, no rigid, fully-specified column list — useful specifically when different rows genuinely need different shapes of data, which is a real, if narrow, gap: this project's own schema, built entirely from fixed, well-known columns (`title` , `summary` , `published`), has never needed this kind of flexibility, since every field every row needs is already known ahead of time.

Querying JSONB: ->, ->>, and @>

Three operators cover most `JSONB` querying. `->` extracts a value by key, keeping it as `JSONB` ; `->>` extracts the same value as text; `@>` checks whether one `JSONB` value contains another:

```
SELECT settings -> 'theme' FROM user_preference;        -- "dark" (as JSONB)
SELECT settings ->> 'theme' FROM user_preference;        -- dark (as text)
SELECT * FROM user_preference WHERE settings @> '{"theme": "dark"}';
```

`->` vs. `->>` matters the moment the extracted value needs to be compared or displayed as a plain string — `settings ->> 'theme' = 'dark'` compares text to text; `settings -> 'theme' = 'dark'` would compare `JSONB` to a text literal and never match, since they're different types even when they "look" the same. `@>` (the "contains" operator) is what makes filtering rows by a nested key practical without extracting it first — `WHERE settings @> '{"theme": "dark"}'` finds every row whose `settings` JSON has that key-value pair, however much else the object contains.

A Realistic JSONB Example

Extending this project's own real `code_example` table (currently just `title` / `example_name` / `sort_order`) with a hypothetical `metadata JSONB` column shows the pattern concretely:

```
-- Hypothetical extension, not a real column in this project
UPDATE code_example
SET metadata = '{"language": "java", "linesOfCode": 24, "tags": ["records",
"immutability"]}'::jsonb
WHERE example_name = 'PointRecordExample';

SELECT example_name FROM code_example
WHERE metadata @> '{"language": "java"}';
```

This is exactly the kind of shape a relational `code_example` table doesn't naturally offer — a variable-length `tags` list and a handful of optional, code-example-specific facts, without a fixed column for each possible one. Whether this project should actually add such a column is a separate design question (a real, fixed `language` `VARCHAR` column, following "PostgreSQL Data Types"'s own guidance, might well be the better choice if `language` is the only field that ever varies) — the point here is only to show `JSONB` querying against a shape that's plausible for this project's own domain, not to argue this project's real schema is missing it.

Arrays

PostgreSQL lets any column type be declared as an array of that type, using `[]`:

```
CREATE TABLE code_example (
    ...
    tags TEXT[]
);

INSERT INTO code_example (title, example_name, sort_order, tags)
VALUES ('Point Record Example', 'PointRecordExample', 1, ARRAY['records', 'immutability']);

SELECT * FROM code_example WHERE 'records' = ANY(tags);
SELECT * FROM code_example WHERE tags @> ARRAY['records'];
```

`ANY(tags)` checks whether a single value appears anywhere in the array — the array equivalent of `IN`, but against one row's own array column rather than a fixed list of literals. `@>` works on arrays the same way it works on `JSONB` — "does this array contain that array (or that single-element array)."

Querying Arrays

Beyond membership, PostgreSQL supports indexing into an array (`tags[1]`, 1-indexed, not 0-indexed — a real, easy-to-forget difference from Java), array length (`array_length(tags, 1)`), and `unnest(tags)`, which expands an array column into one row per element — useful whenever an array needs to be treated as rows for a moment, joined or aggregated like any other table data. A `tags TEXT[]` column and a separate `tag` table joined through a `topic_tag` link table (the exact `@ManyToMany` pattern "Relationships, Fetching, and the N+1 Problem" already covered on the Java side) solve a similar-looking problem differently — the array is simpler for a small, unstructured list with no need to query "which topics share this tag" efficiently at scale; a real join table wins once tags need their own metadata, need to be renamed in one place, or need genuinely efficient lookup in the other direction.

Common Misconceptions

"`JSON` and `JSONB` are interchangeable, `JSONB` is just newer." Not quite — `JSON` preserves exact input text (including key order and duplicate keys) but re-parses on every read; `JSONB` normalizes and can be indexed, and is the right default for nearly every real use case, but the two aren't drop-in replacements for each other in every scenario. **"A UUID primary key is always more secure than `BIGSERIAL`."** It hides sequential information, which is a real, narrow benefit, but "harder to guess the next id" isn't the same as "secure" — access control still has to happen at the application/authorization layer regardless of key type. **"Storing data as `JSONB` avoids needing to design a schema."** It defers the design decision, not eliminates it — every consumer of that `JSONB` column still needs to agree, informally, on what keys might appear and what they mean, without a `CREATE TABLE` statement or "Constraints and Keys" to enforce any of it.

Best Practices

- Default to `BIGSERIAL` (or `GENERATED ALWAYS AS IDENTITY`, from "Constraints and Keys") for internal primary keys, and reach for `UUID` specifically when a value needs to be generated outside the database, or when hiding sequential information is a real requirement — not as a default "more modern" choice.
- Prefer `JSONB` over plain `JSON` unless there's a specific reason to preserve exact input formatting — nearly every real schema that stores JSON at all uses `JSONB`.
- Reach for a `JSONB` or array column only for genuinely variable, per-row-different data — this project's own schema, built entirely from fixed columns known ahead of time, is a real example of when neither is needed at all.
- Choose a real join table over an array column the moment the "list" needs its own metadata, needs renaming in one place, or needs to be queried efficiently from the other direction (which topics share this tag) — the array stays a good fit for a small, simple, per-row list with none of those needs.

Common Mistakes

- Comparing a `->`-extracted `JSONB` value to a plain text literal (`settings -> 'theme' = 'dark'`) instead of using `->>` — this silently never matches, since `JSONB` and `text` are different types, rather than producing an obvious error.
- Reaching for a `JSONB` column as a way to avoid deciding on a schema, then discovering every query needs to know the exact key names and types by convention alone, with none of the guarantees "Constraints and Keys" already covered for real columns.
- Forgetting PostgreSQL arrays are 1-indexed (`tags[1]` is the first element), and either getting `NULL` or an off-by-one result after assuming 0-indexing out of Java habit.
- Modeling a genuinely relational many-to-many (like tags that need their own description, or need efficient "find all topics with this tag" lookups) as an array column instead of a real join table, then having to work around the array's limitations later instead of using the pattern "Relationships, Fetching, and the N+1 Problem" already covered for exactly this shape of problem.

Summary, Cheat Sheet, and Glossary

Summary

- `UUID` (often generated with `gen_random_uuid()`) is an alternative to `BIGSERIAL` for primary keys — larger and unpredictable rather than compact and sequential, useful specifically when an id must be generated outside the database or shouldn't reveal sequence information.

- `JSONB` (preferred over plain `JSON` in nearly every real case) stores semi-structured, per-row-variable data; `->` / `->>` extract a key as `JSONB`/text respectively, and `@>` checks containment.
- Arrays (`TEXT[]`, etc.) hold a variable-length list directly in one column; `ANY(...)` checks membership, `@>` checks containment, and `unnest(...)` expands an array into rows.
- None of these three appear anywhere in this project's own real schema — every example here is a deliberately labeled, illustrative extension, not an existing column, which is itself the point: reach for these types only when a genuine need for them exists.
- A real join table (already covered in "Relationships, Fetching, and the N+1 Problem") remains the right choice over a `JSONB`/array column the moment the data inside needs its own structure, constraints, or efficient querying from the other direction.

Cheat Sheet

```
id UUID PRIMARY KEY DEFAULT gen_random_uuid()

data JSONB
data -> 'key'           -- as JSONB
data ->> 'key'           -- as text
data @> '{"key": "v"}'  -- contains

tags TEXT[]
'x' = ANY(tags)          -- membership
tags @> ARRAY['x']        -- containment
tags[1]                   -- 1-indexed
unnest(tags)               -- one row per element
```

Glossary

- **UUID**: a 128-bit identifier, typically randomly generated, used as an alternative to a sequential primary key.
- **JSONB**: PostgreSQL's binary, indexable JSON type — the preferred JSON storage type over plain `JSON` in nearly every case.
- **Containment operator (@>)**: checks whether one `JSONB` value or array contains another.
- **unnest()**: a function that expands an array column into one row per element.