

THE PRACTICAL GUIDE

JSON Data Design

Using, Modeling, and Storing JSON in Modern Databases

POSTGRESQL · MYSQL · SQL SERVER · ORACLE · SQLITE · MONGODB

Kriangkrai

THENEXBOOKS

JSON Data Design

The Practical Guide to Using, Modeling, and Storing JSON in Modern Databases

Kriangkrai

THENEXBOOKS

JSON Data Design
The Practical Guide to Using, Modeling, and Storing JSON in Modern Databases

First Edition, 2026

Copyright © 2026 Kriangkrai. Published by thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the author and publisher assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Database features change between versions; always consult the official documentation for the version you run.

PostgreSQL, MySQL, MariaDB, Microsoft SQL Server, Oracle, SQLite, MongoDB, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Preface

Almost every system built in the last fifteen years speaks JSON somewhere. It travels between browsers and servers, between microservices, through message queues and webhooks. It lives in configuration files, log pipelines, and — increasingly — inside the database itself. Every major relational database now ships a JSON data type, and document databases built entirely around JSON-like documents power some of the largest applications in the world.

Yet JSON is often treated as something too simple to study. Developers learn the syntax in an afternoon and then spend years discovering its sharp edges the hard way: a customer ID silently rounded because it exceeded 2^{53}, a price stored as a float, a "flexible" `jsonb` column that grew into an unqueryable junk drawer, a migration that locked a table for an hour because one key inside a document had to be renamed.

This book exists to close that gap. It is a practical guide to **using JSON well** — in code, in APIs, and above all in databases. You will learn not just *how* to store and query JSON in PostgreSQL, MySQL, SQL Server, Oracle, SQLite, and MongoDB, but *when* you should, when you should not, and how to design JSON data that stays healthy as your system and your team grow.

Who This Book Is For

- **Backend and full-stack developers** who design APIs and database schemas and want to use JSON columns with confidence.
- **Data engineers and analysts** who ingest, flatten, and query semi-structured data.
- **Architects and tech leads** deciding between relational, document, and hybrid designs.
- **Students and career changers** who know basic SQL and one programming language and want production-grade skills.

What You Will Learn

- The JSON standard in depth — including the number, Unicode, and duplicate-key pitfalls that cause real outages.
- How to define contracts with JSON Schema and design clean, consistent JSON APIs.
- A decision framework for choosing between columns, JSON, and separate tables — and proven hybrid modeling patterns.
- Deep, hands-on coverage of JSON in PostgreSQL, MySQL/MariaDB, SQL Server, Oracle, SQLite, and MongoDB.
- Indexing, performance, validation, security, ORM integration, analytics, and safe schema evolution.
- Three complete case studies: a product catalog, multi-tenant custom fields, and an event/audit/webhook platform.

How This Book Is Organized

Part I — JSON Foundations (Chapters 1–5) covers the format itself, working with it in code, JSON Schema, and API design. **Part II — Data Modeling with JSON** (Chapters 6–8) teaches you to think in both relational and document terms and to evolve stored JSON safely. **Part III — JSON in Databases** (Chapters 9–13) is a practical tour of each major database engine. **Part IV — JSON in Production** (Chapters 14–18) addresses performance, integrity, security, application integration, and analytics. **Part V — Case Studies** (Chapters 19–21) brings everything together in realistic end-to-end designs. The appendices provide a cross-database cheat sheet, a JSON Schema quick reference, a production checklist, and a glossary.

You can read the book cover to cover, or jump straight to the database you use. Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Orbit Store

Throughout the book we use **Orbit Store**, a fictional e-commerce and marketplace platform. Merchants (tenants) sell products with very different attributes; customers place orders; payments, shipments, and events flow through the system; and merchants can define their own custom fields. It is exactly the kind of system where JSON is both tremendously useful and easy to misuse — which makes it the perfect teacher.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause bugs, data loss, or outages in production.

Code appears in `monospace`. SQL examples target PostgreSQL unless stated otherwise; version requirements are called out where they matter.

Contents

Preface

PART I — JSON FOUNDATIONS

1 Why JSON Won

2 JSON Syntax in Depth

3 Working with JSON in Code

4 JSON Schema: Contracts for Your Data

5 Designing JSON Documents and APIs

PART II — DATA MODELING WITH JSON

6 Relational vs. Document Thinking

7 Hybrid Modeling Patterns

8 Schema Evolution and Versioning of Stored JSON

PART III — JSON IN DATABASES

9 PostgreSQL: json and jsonb Fundamentals

10 PostgreSQL: Indexing and Querying JSON at Scale

11 MySQL and MariaDB JSON

12 SQL Server, Oracle, and SQLite JSON

13 MongoDB and Document Databases

PART IV — JSON IN PRODUCTION

14 Performance and Storage Internals

15 Data Integrity and Validation in the Database

16 Security for JSON Data

17 ORMs and Application Integration

18 Analytics, Search, and Data Migration

PART V — CASE STUDIES

19 Case Study: E-commerce Product Catalog

20 Case Study: Multi-tenant SaaS Custom Fields and Settings

21 Case Study: Event Logs, Audit Trails, and Webhooks

APPENDICES

A Cross-Database JSON Cheat Sheet

B JSON Schema Quick Reference

C Production Checklist for JSON Data

D Glossary

PART I

JSON Foundations

Before you store JSON anywhere, you need to know the format precisely — its types, its limits, and the conventions that make JSON data predictable. This part builds that foundation, from the grammar to contracts and API design.

1 · Why JSON Won

2 · JSON Syntax in Depth

3 · Working with JSON in Code

4 · JSON Schema: Contracts for Your Data

5 · Designing JSON Documents and APIs

CHAPTER 1 · PART I — JSON FOUNDATIONS

Why JSON Won

JSON started as a convenient subset of JavaScript and became the default data format of the internet. This chapter explains how that happened, where JSON is used today, how it compares honestly with its competitors, and when you should reach for something else. It closes by introducing Orbit Store, the fictional platform we will design, store, and query throughout the book.

A Format That Was Discovered, Not Invented

Douglas Crockford likes to say that he *discovered* JSON rather than invented it. The claim is more than modesty. Every piece of JSON syntax already existed in JavaScript: curly braces for objects, square brackets for arrays, double-quoted strings, numbers, `true`, `false`, and `null`. What Crockford and his colleagues did around 2001 was notice that this small subset of JavaScript's object literal notation was enough to carry almost any structured data between a server and a browser, and then write it down as a standalone format with a name.

At the time, the dominant answer to "how do I send structured data over HTTP?" was XML. XML had schemas, namespaces, transformation languages, and enterprise vendor support. It also had a lot of ceremony. Parsing XML in a browser meant walking a DOM tree, and mapping that tree onto the objects your program actually wanted to use was tedious. JSON, by contrast, *was* the object. A browser could turn a JSON response into live JavaScript data in a single step.

The early history moved roughly like this:

- **2001–2002.** Crockford and colleagues at State Software use the format for browser–server communication. The `json.org` website goes up, presenting the entire grammar as a handful of railroad diagrams.
- **2005.** The term "Ajax" is coined, describing web applications that fetch data in the background. Although the "X" stood for XML, many Ajax applications quickly switched to JSON because it was smaller and easier to consume in JavaScript.
- **2006.** RFC 4627 registers the `application/json` media type and gives JSON its first formal specification at the IETF.
- **2009.** ECMAScript 5 adds native `JSON.parse` and `JSON.stringify` to the language, ending the era of evaluating JSON with `eval()`, a practice that was both slow and dangerous.
- **2013.** Ecma International publishes ECMA-404, "The JSON Data Interchange Syntax", a short standard that defines only the grammar.
- **2014.** RFC 7159 replaces RFC 4627 and, among other changes, allows any JSON value (not only objects and arrays) at the top level.

- **2017.** RFC 8259 replaces RFC 7159 and becomes Internet Standard STD 90. At the same time, ECMA-404 is revised to its 2nd edition. The two documents are intended to describe the same grammar, and RFC 8259 adds the interoperability guidance that the Ecma text leaves out, most notably that JSON exchanged between systems that are not part of a closed ecosystem **must** be encoded in UTF-8.

That last point is worth pausing on. JSON has two "official" definitions, and they play different roles. ECMA-404 is a pure syntax specification: it tells you which sequences of characters are valid JSON texts. RFC 8259 is the one most engineers should keep bookmarked, because it also discusses the things that break interoperability in practice: character encoding, duplicate object keys, number precision, and the limits that implementations are allowed to impose. We will lean on RFC 8259 heavily in Chapter 2.

> **NOTE**
>
> The grammar has been effectively frozen since the beginning. Crockford deliberately gave JSON no version number, arguing that a format without versions cannot fork. More than two decades later, a JSON document written in 2002 is still valid today, and every conforming parser will read it. Stability of that kind is rare, and it is a large part of why JSON is trusted as a long-lived storage format.

Why It Spread

Plenty of formats are simple. JSON won because it combined simplicity with several other properties at exactly the right moment in the history of the web.

It maps to the data structures every language already has

Nearly every mainstream programming language has some notion of a map or dictionary, a list or array, a string, a number, a boolean, and a null or nil value. JSON has exactly those six kinds of values and nothing else. That means a JSON document converts to native data structures without any intermediate object model. In Python it becomes a `dict`; in Go a `map[string]any` or a struct; in Java a `Map`, a record, or a Jackson tree; in C# a class or a `JsonNode`. XML, by contrast, has elements, attributes, text nodes, mixed content, namespaces, and processing instructions, none of which correspond directly to the way programmers model data in memory.

The grammar is tiny

The complete JSON grammar fits on a single page. There are no attributes versus child elements to decide between, no entity references, no DTDs, and no optional features. A competent programmer can write a correct JSON parser in an afternoon, and many have. That low barrier meant that high-quality parsers appeared for every language almost immediately, and those parsers agreed with each other far more often than the parsers of richer formats did.

It is readable by humans

JSON is text that looks like the code developers already write. You can read an API response in a terminal, paste it into a bug report, or diff two versions in a pull request. When something breaks at 2 a.m., being able to `curl` an endpoint and read the answer is priceless.

It rode the REST and mobile waves

JSON arrived as web APIs moved from SOAP toward resource-oriented HTTP, and just before smartphones made every company build an API for its mobile app. Frameworks made it the default response format, and once HTTP clients, gateways, documentation generators, and browser developer tools all assumed JSON, choosing anything else carried a real cost.

Databases met it halfway

The final step was the arrival of JSON inside databases. Document databases such as MongoDB and CouchDB used JSON-like documents as their primary data model. Relational databases responded: PostgreSQL added a `json` type in 9.2 and the binary `jsonb` type in 9.4; MySQL added a native `JSON` type in 5.7; SQL Server, Oracle, and SQLite added JSON functions; and the SQL standard itself gained JSON features in SQL:2016. Today you can model, index, query, and validate JSON in almost any database you are likely to use, which is precisely what Parts II through IV of this book are about.

Where JSON Lives Today

It is hard to find a modern system that does not produce or consume JSON somewhere. The most common places are worth naming, because each has slightly different requirements, and good JSON design depends on knowing which one you are in.

HTTP APIs

The overwhelming majority of public and internal web APIs use JSON request and response bodies. Standards have grown up around this usage: RFC 9457 defines "Problem Details" error responses (`application/problem+json`), RFC 6902 defines JSON Patch for partial updates, and RFC 7396 defines JSON Merge Patch. GraphQL responses are JSON. OpenAPI descriptions are usually written in JSON or YAML and use JSON Schema to describe payloads. Chapter 5 is devoted to designing JSON for APIs.

`GET /v1/orders/ord_7Q2K9 — response body`

```
{
  "id": "ord_7Q2K9",
  "status": "paid",
  "currency": "USD",
  "total": "129.90",
  "createdAt": "2026-03-14T09:26:53Z",
  "customer": { "id": "cus_1042", "email": "ana@example.com" },
  "items": [
    { "sku": "TSHIRT-ORB-M", "quantity": 2, "unitPrice": "24.95" },
```

```
    { "sku": "MUG-ORB-BLK", "quantity": 1, "unitPrice": "80.00" }
  ]
}
```

Configuration

Package manifests (`package.json`), TypeScript compiler settings (`tsconfig.json`), editor settings, cloud infrastructure templates, and countless application config files are JSON or a JSON dialect. Configuration is where JSON's lack of comments hurts most, which is why several tools accept "JSON with comments" variants. We cover those in Chapter 2.

Structured logs

Logging platforms increasingly expect one JSON object per log line. A structured log entry can be filtered and aggregated by field (`level`, `tenantId`, `durationMs`) instead of by fragile regular expressions over free text. The one-object-per-line convention is called JSON Lines or NDJSON, and it is also the standard way to ship large datasets in JSON.

app.log (JSON Lines)

```
{"ts":"2026-03-14T09:26:53.120Z","level":"info","msg":"order paid","orderId":"ord_7Q2K9"}
{"ts":"2026-03-14T09:26:53.348Z","level":"warn","msg":"slow query","durationMs":812}
```

Messaging and events

Message brokers and event streams carry JSON payloads constantly: webhooks sent to merchants, domain events published to a queue, change-data-capture records flowing into a warehouse. The CloudEvents specification, widely used for event metadata, defines a JSON format. In high-throughput pipelines teams sometimes switch to a binary format such as Avro or Protocol Buffers, but JSON remains the default for anything crossing an organizational boundary, because the receiving side can read it without sharing a code generator or schema registry.

Databases and storage

Finally, JSON is a storage format. Document databases store it natively; relational databases store it in dedicated column types; data lakes store it as JSON Lines files; analytics engines query it directly. This is the use case where design decisions have the longest consequences. An API payload lives for milliseconds. A JSON column lives for years, gets read by code that has not been written yet, and must survive changes to the application that produced it. Much of this book is about getting that right.

The Alternatives, Honestly

JSON is not the best format for everything, and pretending otherwise leads to bad engineering. Each competitor below exists because it solves a problem JSON does not solve well. Understanding them makes you better at deciding when JSON is the right tool and at designing JSON that avoids its known weak spots.

XML

XML (a W3C Recommendation since 1998) is older, richer, and far more capable of expressing *documents*, meaning text with embedded markup. It supports attributes, namespaces, mixed content (text interleaved with elements), comments, and an ecosystem of mature standards: XML Schema, XPath, XSLT, XML Signature. For publishing formats, office documents, and many regulated industries (finance messaging, healthcare, government filings), XML remains the right or mandated choice. For plain data records exchanged between programs, however, XML's richness becomes friction: you must decide whether a value is an attribute or an element, deal with namespace prefixes, and translate a tree model into your program's objects.

YAML

YAML optimizes for humans writing configuration by hand. It supports comments, multi-line strings, anchors and aliases for reuse, and an indentation-based syntax without braces. YAML 1.2 was designed so that JSON is (very nearly) a valid subset of YAML. The costs are complexity and surprise. The YAML specification is many times longer than JSON's, indentation errors are silent, and implicit typing has caused famous bugs: under YAML 1.1 rules, the unquoted value `NO` (the country code for Norway) is parsed as the boolean `false`, and values like `1.10` can turn into the number `1.1`. YAML is a fine choice for configuration humans edit, and a poor choice for data machines exchange.

CSV

CSV is the lingua franca of spreadsheets and bulk data exports. It is compact for flat, tabular data because column names appear once in a header rather than in every record. But CSV has no nesting, no types (everything is a string), and only a loose specification: RFC 4180 describes a common form, yet real-world files vary in delimiters, quoting, line endings, and encodings. CSV is perfect for "give me the orders table as a spreadsheet" and unsuitable for an order with nested line items, addresses, and metadata.

Protocol Buffers

Protocol Buffers (Protobuf), open-sourced by Google in 2008, is a schema-first binary format. You define messages in a `.proto` file, generate code for each language, and serialize to a compact binary encoding in which fields are identified by numbers rather than names. The results are small, fast to parse, and strongly typed, and the numbered-field design gives clear rules for evolving schemas without breaking old readers. Protobuf is the foundation of gRPC. The trade-off is that the payload is not self-describing: without the schema, the bytes are close to meaningless, and you cannot inspect a message with `curl` and your eyes.

Avro

Apache Avro is another schema-based binary format, popular in the Hadoop and Kafka ecosystems. Avro schemas are themselves written in JSON, and Avro's defining feature is schema resolution: data written with one version of a schema can be read with another, as long as the changes follow Avro's

compatibility rules. Paired with a schema registry, Avro makes large event pipelines robust against schema drift. Like Protobuf, it is compact and fast but not human-readable, and it requires infrastructure to distribute schemas.

MessagePack and CBOR

MessagePack and CBOR (Concise Binary Object Representation, RFC 8949) are "binary JSON": schemaless, self-describing formats with essentially the same data model as JSON, plus extras JSON lacks, such as a native byte-string type. CBOR also has tags for dates, big numbers, and other extended types, and is used in constrained-device protocols and in security standards such as WebAuthn. They are smaller and faster to parse than JSON text, and they convert to and from JSON easily, but they lose readability and have far less tooling. Several databases use conceptually similar binary encodings internally: PostgreSQL's `jsonb`, MongoDB's BSON, and MySQL's binary JSON storage all trade text for a parsed, binary representation. We study those internals in Chapter 14.

Side by side

Format	Encoding	Schema	Human-readable	Nesting	Comments	Typical sweet spot
JSON	Text (UTF-8)	Optional (JSON Schema)	Yes	Yes	No	APIs, events, stored documents
XML	Text	Optional (XSD, DTD)	Yes, verbose	Yes	Yes	Documents, regulated B2B exchange
YAML	Text	Optional	Very	Yes	Yes	Hand-edited configuration
CSV	Text	None (header row)	Yes	No	No	Flat tabular exports
Protocol Buffers	Binary	Required (.proto)	No	Yes	In schema	Internal RPC (gRPC), mobile
Avro	Binary	Required (JSON schema)	No	Yes	In schema	Event streams, data lakes
MessagePack / CBOR	Binary	None	No	Yes	No	Compact JSON-like payloads, IoT

The columns trade off against one another: readability pushes toward text, size and speed toward binary, safety under evolution toward required schemas. JSON sits at a specific point (readable, self-describing, schema optional) that suits most application developers most of the time. It is also why this book spends so much effort adding back discipline that JSON itself does not impose.

Strengths and Weaknesses

Strengths

- **Universal support.** Every language, database, message broker, log platform, and browser handles JSON out of the box. Choosing JSON never requires justification in a design review.
- **Self-describing.** Field names travel with the data. A consumer can make sense of a payload without prior agreement, which is invaluable for public APIs, webhooks, and debugging.
- **Flexible structure.** Optional fields, nested objects, and arrays of varying length are natural. You can add a field without coordinating a schema deployment across every system.
- **Stable and simple specification.** The grammar has not changed in two decades, and interoperability between parsers is generally excellent for well-behaved data.
- **Excellent tooling.** Pretty-printers, validators, diff tools, `jq`, editor support, JSON Schema, and database functions are all mature.

Weaknesses

- **A thin type system.** There is no date type, no decimal type, no integer type distinct from floating point, no binary type, and no way to express a set or a map with non-string keys. Every system must agree on conventions: ISO 8601 strings for timestamps, strings for money, base64 for bytes. When they do not agree, bugs follow.
- **Number ambiguity.** JSON numbers have arbitrary precision in the text, but many parsers read them as 64-bit floating point. Integers above 2^{53} can silently change value in JavaScript. Chapter 2 treats this in depth, because it causes real production incidents with IDs and money.
- **Verbosity at scale.** Repeating every key in every record costs space and parsing time. In a stream of a billion events, the string `"customerId"` appears a billion times. Compression helps a lot on the wire, and databases with binary JSON storage reduce the cost of parsing but generally still store the keys.
- **No comments.** Crockford removed comments deliberately, having seen people use them to carry parsing directives. That makes JSON a mediocre configuration language.
- **Schema is optional.** Flexibility is a strength until it isn't. Without a schema, nothing stops one producer from sending `"price": 19.9` and another from sending `"price": "19.90"`. The format itself offers no protection against drift.
- **Underspecified edge cases.** Duplicate keys, very large numbers, and invalid Unicode are all handled differently by different parsers. RFC 8259 acknowledges this and I-JSON (RFC 7493) exists to tighten it.

When Not to Use JSON

A practical engineer knows the boundaries of their tools. JSON is usually a good default, but the following situations deserve a second look.

- **High-volume internal service-to-service calls.** When two services owned by the same organization exchange millions of messages per second, the cost of text encoding, repeated keys, and number parsing becomes measurable. Protocol Buffers with gRPC, or another schema-based binary format, is often a better fit, with JSON reserved for the public edge. Measure first: for most services, JSON serialization is a small fraction of request time compared with database and network latency.
- **Large analytical datasets.** For billions of rows queried by column, row-oriented JSON is the wrong shape. Columnar formats such as Apache Parquet or ORC compress far better and allow engines to read only the columns a query needs. JSON Lines is fine for ingestion and interchange; convert to columnar storage for analysis. Chapter 18 returns to this.
- **Binary payloads.** Images, audio, PDFs, and other binary blobs do not belong inside JSON. Base64 encoding inflates data by about one third and forces the whole blob into memory during parsing. Store binaries in object storage and put a URL or content hash in the JSON.
- **Hand-edited configuration with many comments.** If humans will maintain a file and need to explain their choices inline, consider TOML, YAML, or a JSON dialect with comments rather than fighting plain JSON.
- **Documents with mixed content.** Rich text with inline markup, such as legal documents or publishing workflows, is what XML (or HTML) was designed for. Encoding mixed content in JSON is possible but awkward.
- **Data that is really relational.** This is the most important case for this book. If your data has stable structure, strong relationships, and needs referential integrity, storing it as a JSON blob in a database throws away the guarantees a relational schema gives you for free. Chapter 6 develops a decision framework for this, and Chapter 7 shows how to combine relational columns and JSON in the same model.

> **WARNING**
>
> The most expensive JSON mistake is not a syntax error; it is using a JSON column as a way to avoid designing a schema. It feels fast on day one and becomes very slow on day five hundred, when nobody knows which of seven historical document shapes a row contains. JSON is flexible storage, not a substitute for thinking about your data.

Meet Orbit Store

Abstract advice about data modeling is easy to agree with and hard to apply. To keep this book concrete, every chapter draws its examples from a single fictional company: **Orbit Store**, a multi-tenant e-commerce and marketplace platform. Orbit Store lets independent merchants open online shops on a shared platform, much like the hosted storefront services you may have used. Customers browse products, place orders, pay, and receive shipments; merchants manage catalogs, configure their stores, and integrate with their own systems through webhooks.

Orbit Store is deliberately designed to exercise every interesting JSON problem. It has data that clearly belongs in relational tables, data that is naturally document-shaped, and a large gray zone in between where the right choice depends on access patterns and trade-offs.

The core entities

Entity	What it holds	Why it is interesting for JSON
Tenants (merchants)	Store name, plan, settings, custom field definitions	Per-tenant settings and custom fields vary widely; classic JSON territory (Chapter 20)
Customers	Profile, addresses, marketing preferences	Stable core columns plus flexible preferences and tenant-defined custom fields
Products	Title, price, inventory, category-specific attributes	A T-shirt has size and color; a laptop has RAM and CPU; a book has an ISBN (Chapter 19)
Orders and line items	Totals, status, items, addresses at time of purchase	Snapshots of prices and addresses; money precision; mixing relational and embedded data
Payments	Amounts, provider, raw provider responses	Storing third-party payloads whose shape you do not control
Shipments	Carrier, tracking, status history	Nested arrays of events; querying inside arrays
Events and audit logs	Who did what, when, before and after values	High-volume append-only JSON; retention and indexing (Chapter 21)
Webhooks	Subscriptions, delivery attempts, payloads	Versioned outbound JSON contracts and replay

A first look at the data

Here is a product from a merchant selling apparel. The fields common to every product (identifier, tenant, title, price, status) are stable and predictable. The `attributes` object is where categories differ, and the `custom` object holds fields the merchant defined themselves in the Orbit Store admin panel.

product.json

```
{
  "id": "prd_01HQ3M8Z4K",
  "tenantId": "ten_luna_apparel",
  "sku": "TSHIRT-ORB-M",
  "title": "Orbit Logo T-Shirt",
  "category": "apparel/tshirts",
  "price": { "amount": "24.95", "currency": "USD" },
  "status": "active",
  "attributes": {
    "size": "M",
    "color": "midnight",
    "material": "100% organic cotton",
    "fit": "regular"
  },
  "custom": {
```

```
    "collection": "Spring 2026",
    "printedLocally": true
  },
  "tags": ["logo", "organic", "bestseller"],
  "updatedAt": "2026-03-01T12:00:00Z"
}
```

A product from a different merchant, in a different category, has the same outer shape but completely different attributes:

product-laptop.json

```
{
  "id": "prd_01HQ3N2B7P",
  "tenantId": "ten_bytebarn",
  "sku": "LAPTOP-BB-14",
  "title": "ByteBarn 14\" Ultrabook",
  "category": "electronics/laptops",
  "price": { "amount": "1249.00", "currency": "EUR" },
  "status": "active",
  "attributes": {
    "cpu": "8-core",
    "ramGb": 16,
    "storageGb": 512,
    "screenInches": 14.0,
    "ports": ["usb-c", "usb-c", "hdmi"]
  },
  "tags": ["laptop"],
  "updatedAt": "2026-02-18T08:30:00Z"
}
```

Even these two small documents raise the questions this book answers. Should `price` be a relational column rather than a JSON object? (Almost certainly yes, if you sort and filter by it.) Why is the amount a string rather than a number? (Chapter 2 explains.) How do you let merchants search for "all laptops with at least 16 GB of RAM" efficiently? (Chapters 9 through 13 show how in each major database.) How do you stop a buggy import from writing `"ramGb": "sixteen"`? (Chapters 4 and 15.) What happens when the attributes for laptops change next year? (Chapter 8.)

Orders show a different pattern: a mix of references and snapshots. The order refers to the customer by ID, but it copies the shipping address and each item's price at the moment of purchase, because those must not change if the customer later moves or the merchant changes a price.

order.json

```
{
  "id": "ord_7Q2K9",
  "tenantId": "ten_luna_apparel",
  "customerId": "cus_1042",
  "status": "paid",
  "currency": "USD",
  "totals": { "subtotal": "129.90", "shipping": "0.00", "tax": "0.00",
              "grand": "129.90" },
```

```
  "shippingAddress": {
    "name": "Ana Souza",
    "line1": "221 Harbor Street",
    "city": "Lisbon",
    "postalCode": "1100-148",
    "country": "PT"
  },
  "items": [
    { "productId": "prd_01HQ3M8Z4K", "sku": "TSHIRT-ORB-M", "title": "Orbit Logo T-Shirt",
      "quantity": 2, "unitPrice": "24.95" },
    { "productId": "prd_01HQ3P9C2D", "sku": "MUG-ORB-BLK", "title": "Orbit Mug, Black",
      "quantity": 1, "unitPrice": "80.00" }
  ],
  "createdAt": "2026-03-14T09:26:53Z"
}
```

Finally, events. Every meaningful change in Orbit Store emits an event that is written to an audit log and may be delivered to merchants as a webhook. Events are append-only, high-volume, and schema-versioned, which makes them a very different design problem from products and orders.

event.json

```
{
  "id": "evt_01HQ4A0XJ3",
  "type": "order.paid",
  "version": 2,
  "tenantId": "ten_luna_apparel",
  "occurredAt": "2026-03-14T09:26:53.120Z",
  "actor": { "type": "system", "id": "payments-worker" },
  "data": {
    "orderId": "ord_7Q2K9",
    "amount": "129.90",
    "currency": "USD",
    "paymentId": "pay_88XK1"
  }
}
```

Notice some conventions already at work, which the next chapters will justify in detail: identifiers are prefixed strings, timestamps are RFC 3339 strings in UTC, money amounts are decimal strings paired with an ISO 4217 currency code, field names use `camelCase` consistently, and events carry an explicit `version`. None of these are required by JSON. All of them are the result of lessons learned in production.

How This Book Is Organized

The book is divided into five parts that move from the format itself to production systems.

- **Part I — JSON Foundations** (Chapters 1–5). After this introduction, Chapter 2 covers the syntax in precise detail, including the numeric and Unicode edge cases that cause subtle bugs. Chapter 3 shows how to parse and produce JSON correctly in JavaScript/TypeScript, Python,

Java, Go, and C#, plus command-line processing with `jq`. Chapter 4 introduces JSON Schema (draft 2020-12) as a contract for your data. Chapter 5 applies all of this to designing JSON documents and APIs.

- **Part II — Data Modeling with JSON** (Chapters 6–8). How to decide between relational and document modeling, the hybrid patterns that combine them, and how to evolve JSON that is already stored in production without breaking readers.
- **Part III — JSON in Databases** (Chapters 9–13). Deep, practical coverage of PostgreSQL `json` and `jsonb` (two chapters), MySQL and MariaDB, SQL Server, Oracle, SQLite, and MongoDB. Each chapter covers storage, querying, indexing, and the version-specific features you need to know.
- **Part IV — JSON in Production** (Chapters 14–18). Storage internals and performance, integrity and validation inside the database, security, ORM integration, and analytics, search, and migration.
- **Part V — Case Studies** (Chapters 19–21). Three complete designs from Orbit Store: the product catalog, multi-tenant custom fields and settings, and event logs, audit trails, and webhooks.

The appendices provide a cross-database cheat sheet, a JSON Schema quick reference, a production checklist, and a glossary. You can read the book front to back, but the parts are designed to stand on their own. If you already know JSON well and are here for database design, skim Part I (read Chapter 2's section on numbers regardless) and start with Chapter 6.

> **TIP**
>
> Keep a scratch database handy while reading Parts II and III. PostgreSQL, MySQL, and SQLite all run easily on a laptop or in a container, and running the examples yourself, then deliberately breaking them, is the fastest way to build intuition for how JSON behaves inside a database.

A Philosophy for the Rest of the Book

Three ideas color every recommendation that follows. First, **JSON is a boundary format that became a storage format**; storing it for years and querying it with SQL demands more than its designers anticipated. Second, **flexibility must be paid for with discipline elsewhere**: choosing JSON moves responsibility for data quality from the database engine to your schemas, validation, and versioning. Third, **the right answer is usually hybrid**. Orbit Store keeps orders, customers, and payments in relational tables and uses JSON for the parts that genuinely vary. Learning where to draw that line is the central skill this book teaches. Chapter 2 begins with the format itself.