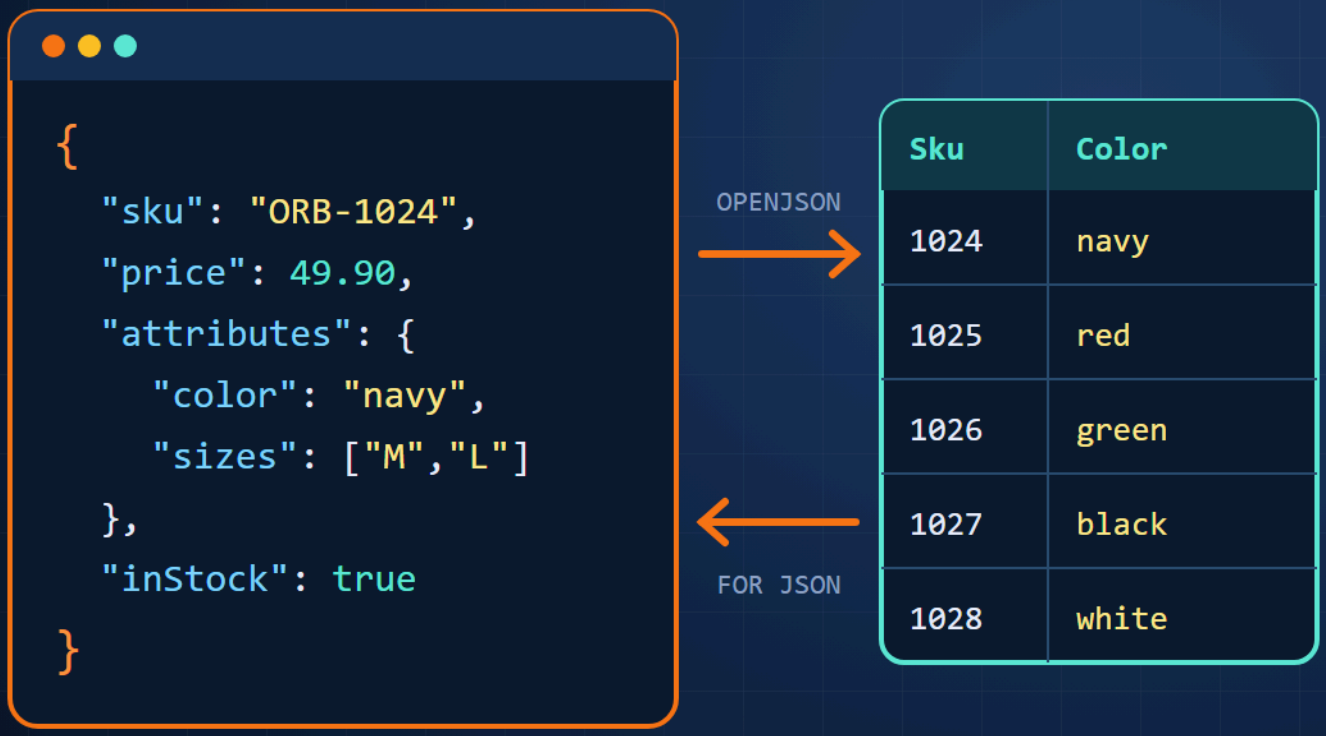


JSON_VALUE · JSON_QUERY · JSON_MODIFY · ISJSON

THE COMPLETE PRACTICAL GUIDE

Working with JSON in SQL Server

Store, query, index, and produce JSON with T-SQL

For SQL Server, Azure SQL Database,
and Azure SQL Managed Instance

OPENJSON · FOR JSON · INDEXING · json TYPE · APIs

THENEXBOOKS

Working with JSON in SQL Server

The Complete Practical Guide to Storing, Querying, Indexing, and Producing JSON with T-SQL
For SQL Server, Azure SQL Database, and Azure SQL Managed Instance

First Edition

THENEXBOOKS

Working with JSON in SQL Server
The Complete Practical Guide to Storing, Querying, Indexing, and Producing JSON with T-SQL

First Edition, 2026

Copyright © 2026 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 publisher, 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 publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. SQL Server and Azure SQL evolve quickly, and several JSON features described in this book were in preview at the time of writing; always consult the official documentation for the version, edition, and compatibility level you use.

Microsoft, SQL Server, Azure, Azure SQL, Entity Framework, and .NET are trademarks of the Microsoft group of companies. Node.js, Python, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by Microsoft Corporation.

Orbit Store, the running example in this book, is fictional. Any resemblance to real companies or products is coincidental.

Preface

Almost every modern application speaks JSON. Web and mobile clients send it, payment and shipping providers post it to your webhooks, partner systems exchange it, logging and analytics pipelines emit it, and AI assistants call tools with JSON arguments and expect JSON back. Sooner or later, that JSON arrives at your relational database — and you have to decide what to do with it.

SQL Server has been able to work with JSON since SQL Server 2016, and the feature set has grown steadily since then: constructor functions, path-existence tests, stricter validation, and, in Azure SQL and SQL Server 2025, a native `json` data type with aggregate functions and new indexing options. Used well, these features let you keep strong relational guarantees where they matter while embracing flexible documents where they help. Used carelessly, they produce slow queries, unindexable predicates, and documents nobody can validate or evolve.

This book is a complete, practical guide to doing it well. We start with the fundamentals of JSON and how SQL Server stores and validates it, move through every function for reading, producing, and modifying JSON, and then spend a large part of the book on what matters in production: indexing, query tuning, bulk loading, data modeling, application integration, security, testing, and operations. We explain *why* each technique works — and where it breaks — not just *how* to type it.

Who This Book Is For

- **Database developers** who write T-SQL every day and need to ingest, query, and return JSON reliably.
- **Backend and full-stack developers** who call SQL Server from .NET, Node.js, or Python and want to push JSON work to the right layer.
- **DBAs and performance engineers** who must keep JSON-heavy workloads fast, indexed, secure, and maintainable.
- **Architects and data engineers** deciding between relational columns, JSON documents, or a hybrid of both.

What You Will Learn

- JSON essentials for SQL developers, storage options (`nvarchar(max)`, UTF-8 `varchar(max)`, and the native `json` type), and validation with `ISJSON` and constraints.
- Path expressions, `JSON_VALUE`, `JSON_QUERY`, and `OPENJSON` — from simple lookups to multi-level shredding of nested arrays.
- Producing JSON with `FOR JSON`, `JSON_OBJECT`, `JSON_ARRAY`, and the JSON aggregate functions, and building API-ready documents.
- Changing documents safely with `JSON_MODIFY`.

- Indexing JSON with computed columns, understanding JSON indexes, tuning queries, and bulk loading and exporting files.
- Hybrid data modeling, JSON parameters for stored procedures, and integration with EF Core, Dapper, Node.js, and Python.
- Auditing, security, privacy, cloud integrations, testing, and two complete case studies — including an AI shopping assistant.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why JSON belongs in SQL Server, how to store and validate it, and how path expressions work. **Part II — Reading JSON** (Chapters 6–10) covers extracting values, shredding documents with `OPENJSON`, and searching inside them. **Part III — Producing and Changing JSON** (Chapters 11–15) turns relational data into JSON and modifies existing documents. **Part IV — Performance, Indexing, and Modeling** (Chapters 16–20) makes JSON workloads fast and well designed. **Part V — Application Integration** (Chapters 21–25) connects the database to application code, auditing, security, and the cloud. **Part VI — Production and Case Studies** (Chapters 26–28) brings everything together. The appendices provide a cheat sheet, templates and a function reference, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Orbit Store

Throughout the book we work with **Orbit Store**, a fictional e-commerce marketplace. Merchants list products whose attributes vary widely by category; customers keep profiles with preferences and addresses; orders flow through payment and shipping providers that send JSON webhooks; and an AI shopping assistant calls the database through carefully guarded stored procedures. The `OrbitStore` database uses the schemas `catalog`, `sales`, `ops`, and `audit`, and it is the perfect place to see when JSON helps and when a plain column is better.

Versions and Conventions

Examples target SQL Server 2016 and later unless a section says otherwise. Features introduced in SQL Server 2022, and features available in Azure SQL and SQL Server 2025, are marked where they appear. Features that were still in preview at the time of writing are labeled as preview, and their syntax may change.

> **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`. T-SQL keywords are written in uppercase, and sample JSON documents are kept short enough to read on the page.

Contents

Preface

PART I — FOUNDATIONS

1 Why JSON in SQL Server: Relational Meets Document

2 JSON Essentials for SQL Developers

3 Storing JSON: nvarchar(max), varchar(max), and the Native json Type

4 Validating JSON: ISJSON, CHECK Constraints, and Data Quality

5 JSON Path Expressions in Depth: Lax, Strict, Arrays, and Escaping

PART II — READING JSON

6 Extracting Values: JSON_VALUE and JSON_QUERY

7 Shredding JSON with OPENJSON: The Default Schema

8 OPENJSON with Explicit Schemas and CROSS APPLY

9 Nested Arrays, Hierarchies, and Multi-Level Shredding

10 Existence, Membership, and Search: JSON_PATH_EXISTS, JSON_CONTAINS, and Patterns

PART III — PRODUCING AND CHANGING JSON

11 FOR JSON PATH: Shaping Query Output

12 FOR JSON AUTO and Output Options: ROOT, INCLUDE_NULL_VALUES, WITHOUT_ARRAY_WRAPPER

13 Constructing JSON: JSON_OBJECT, JSON_ARRAY, JSON_OBJECTAGG, and JSON_ARRAYAGG

14 Building API-Ready Documents from Relational Data

15 Modifying JSON: JSON_MODIFY and the json Type's modify Method

PART IV — PERFORMANCE, INDEXING, AND MODELING

16 Indexing JSON with Computed Columns

17 The Native json Type and JSON Indexes in Depth

18 Query Tuning for JSON Workloads

19 Bulk Loading and Exporting JSON: OPENROWSET, BULK INSERT, bcp, and Files

20 Hybrid Data Modeling: Columns, JSON, or Both?

PART V — APPLICATION INTEGRATION

21 JSON Parameters: Passing Sets and Documents to Stored Procedures

22 JSON from Application Code: .NET, EF Core, Dapper, Node.js, and Python

23 Auditing, Change Capture, and Event Payloads with JSON

24 Security, Privacy, and Governance for JSON Data

25 JSON in Azure SQL and the Cloud: REST Endpoints, Data API builder, and Integrations

PART VI — PRODUCTION AND CASE STUDIES

26 Testing, Schema Evolution, and Operating JSON in Production

27 Case Study: The Orbit Store Product Catalog with Flexible Attributes

28 Case Study: The Orbit Store Order API and AI Shopping Assistant

APPENDICES

A T-SQL JSON Cheat Sheet

B Templates and Function Reference

C Production Checklist

D Glossary

PART I

Foundations

Why JSON belongs in a relational engine, what JSON really is from a SQL developer's point of view, how SQL Server stores and validates it, and how path expressions address every value inside a document.

1 · Why JSON in SQL Server
2 · JSON Essentials for SQL Developers
3 · Storing JSON
4 · Validating JSON
5 · JSON Path Expressions in Depth

CHAPTER 1 · PART I — FOUNDATIONS

Why JSON in SQL Server: Relational Meets Document

JSON has become the common language of APIs, mobile apps, event streams, and AI services, and sooner or later it arrives at your database. This chapter explains why SQL Server speaks JSON, when storing JSON in a relational engine is a smart decision and when it is a trap, how the feature set evolved from SQL Server 2016 to the native `json` type, and how to build the Orbit Store database used throughout this book.

The Problem JSON Solves

Relational databases are built on a simple promise: decide the shape of your data up front, express it as tables and columns, and the engine will enforce that shape, index it, and join it efficiently for decades. That promise is still the best foundation for most business data. Orders have dates, customers have email addresses, order lines have quantities and prices. These facts are stable, well understood, and queried constantly, so they belong in typed columns.

But modern applications also produce data that does not fit neatly into a fixed set of columns. Consider Orbit Store, the fictional e-commerce marketplace we will build throughout this book. Orbit Store hosts thousands of independent merchants. A merchant selling running shoes wants to record sizes, widths, drop height, and cushioning level. A merchant selling laptops cares about CPU, RAM, storage, screen size, and ports. A merchant selling coffee wants roast level, origin, and tasting notes. Each merchant also has its own settings: return windows, shipping zones, notification preferences, branding colors, integration keys.

You could try to model all of this relationally. The first attempt is usually a very wide `Product` table with hundreds of nullable columns, most of them empty for any given row. The second attempt is the entity-attribute-value (EAV) pattern: a narrow table of `(ProductId, AttributeName, AttributeValue)` rows. EAV is flexible, but every value becomes a string, reassembling a product requires pivoting many rows, and simple questions such as "show me blue shirts in size M under 30 dollars" turn into self-joins that are painful to write and hard to optimize.

Meanwhile, the data arrives from the outside world already shaped as JSON. The storefront posts a shopping cart as a JSON document. The payment provider sends a webhook whose body is JSON. The shipping partner returns tracking events as a JSON array. The AI shopping assistant wants product details as JSON so it can reason over them. If the database cannot accept, store, query, and produce JSON, every one of those integrations needs a layer of translation code, and that code becomes a source of bugs, latency, and duplicated business rules.

JSON support in SQL Server exists to close that gap. It lets you keep the stable, heavily queried core of your data in relational columns while storing the variable, sparse, or externally defined parts as JSON documents, and it gives you T-SQL tools to move smoothly between the two worlds.

Relational Meets Document

Document databases popularized the idea of storing a whole business object, such as an order with its lines and shipping details, as one self-describing document. The appeal is obvious: the document matches the object your application already has in memory, there is no mapping layer, and adding a new field requires no schema migration. The cost is equally real: without a schema, the database cannot guarantee that every order has a total, cross-document queries and joins are harder, and data duplicated across documents drifts out of sync.

SQL Server's approach is not to become a document database. Instead, it treats JSON as a first-class *format* that relational queries can read and write. In practice this gives you a spectrum of choices rather than a binary decision:

- **Fully relational.** Every attribute is a typed column. JSON appears only at the edges, when you produce API responses with `FOR JSON` or accept parameters with `OPENJSON`.
- **Hybrid.** Core attributes are columns; a JSON column holds the flexible remainder. This is the sweet spot for most systems and the model Orbit Store uses.
- **Document-centric.** A table holds little more than a key, a few timestamps, and a JSON payload. This fits event logs, webhook archives, audit trails, and caches of external data.

The key insight is that you can choose per table, and even per column, where you sit on that spectrum. You can also move over time: a JSON property that proves important can be promoted to a real column, and a rarely used column can be folded back into a JSON bag. Chapter 20 is devoted to these modeling decisions; for now, it is enough to know that JSON in SQL Server is a tool for the hybrid middle ground, not a replacement for relational design.

What SQL Server gives you that a separate document store does not

Teams sometimes consider running a document database alongside SQL Server just for the flexible data. That can be the right call at very large scale or for specialized workloads, but it has costs that are easy to underestimate. With JSON inside SQL Server you keep:

- **One transaction boundary.** Inserting an order row and its JSON metadata happens atomically. There is no dual-write problem and no eventual consistency between two stores.
- **One security model.** Logins, roles, row-level security, auditing, and encryption apply to JSON columns just as they apply to every other column. Chapter 24 covers the details.
- **One backup and recovery story.** Point-in-time restore brings back the relational rows and their JSON documents together.

- **Joins across both worlds.** A query can join `sales.[Order]` to `catalog.Product` and filter on a property buried in the product's JSON attributes, in a single statement, with a single execution plan.
- **Mature tooling.** Query Store, execution plans, Extended Events, and the rest of your operational toolkit work on JSON queries without extra infrastructure.

When JSON Belongs in the Database

The following patterns are strong candidates for JSON storage. You will see each of them in the Orbit Store schema.

Variable or sparse attributes

Product attributes are the classic example. Every product has a SKU, a name, a price, and a merchant; those are columns. Beyond that, attributes vary by category and by merchant. Storing them as a JSON object in `catalog.Product.Attributes` keeps the table narrow and lets merchants add attributes without a schema change. You can still index the handful of attributes that are searched frequently, as Chapter 16 shows.

Configuration and preferences

Merchant settings and customer preferences change shape as the product evolves. Today a merchant has a return window and a list of shipping zones; next quarter the product team adds a "gift wrapping" option with its own nested settings. These values are usually read as a whole by the application and rarely queried across rows, which is exactly the access pattern JSON handles well.

External payloads you do not control

When a payment provider sends a webhook, its structure is defined by the provider and may change without notice. Storing the raw payload in `ops.WebhookEvent.Payload` preserves exactly what was received, which is invaluable for debugging and reprocessing. You can shred the fields you need into relational tables, and keep the original for audit.

Audit trails and change history

An audit table that captures "before" and "after" images of changed rows needs to hold rows from many different tables. Serializing each image as JSON gives one generic table, `audit.ChangeLog`, instead of one history table per source table. Chapter 23 builds this pattern in full.

API input and output

Even in a fully relational design, JSON is the natural format for passing a set of rows into a stored procedure (for example, an order with ten lines) and for returning nested results to an API. Chapter 14 and Chapter 21 focus on these edge-of-the-database uses.

When JSON Does Not Belong

JSON is not free. Every time you read a property, SQL Server must parse text (or, with the native type, navigate a binary structure) to find it. The engine cannot enforce foreign keys inside a document, cannot keep statistics on nested values unless you expose them, and cannot stop two documents from disagreeing about the same fact. Watch for these warning signs:

- **Core business facts hidden in JSON.** If every query filters on `$.status` or sums `$.total`, those values should be columns. Hiding them in a document makes the most common queries the most expensive.
- **Relationships inside documents.** If a JSON document contains a customer ID that must match a real customer, you want a foreign key, and foreign keys need columns.
- **Frequent partial updates to large documents.** Changing one property in a large text document rewrites the value. With many concurrent writers, this becomes a hotspot.
- **Using JSON to avoid design.** "We will just put it in JSON and figure it out later" is how teams end up with a table of unqueryable blobs and no one who knows what is inside them.

Data	Typical choice	Why
Order date, status, total	Columns	Filtered, sorted, aggregated
Product specs by category	JSON	Varies per product
Customer email	Column	Unique key, login lookup
Customer preferences	JSON	Read as a unit, evolves
Webhook body	JSON	External, must be preserved
Order lines	Table	Joined, aggregated, keyed
Before/after row images	JSON	Generic across tables

> **TIP**
>
> A useful rule of thumb: if a value participates in a join, a uniqueness rule, a foreign key, or the `WHERE` clause of your hottest queries, make it a column. If it is mostly read and written as part of a larger whole, JSON is a reasonable home. When in doubt, start hybrid and promote properties to columns as usage patterns become clear.

How SQL Server Represents JSON

For most of its history, SQL Server had no dedicated JSON data type. Starting with SQL Server 2016, JSON was stored as ordinary text, typically in an `nvarchar(max)` column, and a set of built-in functions understood that text. This design was deliberate: text columns work with every existing driver, ORM, replication feature, and tool, so adopting JSON required no client changes. The trade-off

is that a text column does not know it contains JSON. You must add a `CHECK` constraint if you want the engine to reject malformed documents (Chapter 4), and each function call parses the text to find what it needs.

Azure SQL Database, Azure SQL Managed Instance, and SQL Server 2025 add a native `json` data type. A `json` column stores documents in an optimized binary format, validates them on the way in, and allows the engine to read properties without reparsing the whole document from text. Existing functions such as `JSON_VALUE` and `OPENJSON` accept both text and `json` inputs, so most of the queries in this book work the same way against either storage choice. Chapter 3 compares the storage options in detail, and Chapter 17 goes deep into the native type.

The important conceptual point is that SQL Server separates *storage* from *functions*. Whether a document lives in `nvarchar(max)`, `varchar(max)`, or `json`, you use the same family of functions to validate it, extract values, shred it into rows, modify it, and build new documents.

Feature Timeline by Version

JSON support arrived in stages. Knowing which features exist in which version saves you from writing a query in development that fails on an older production server. The table below summarizes the milestones; the rest of the book notes version requirements whenever a feature is introduced.

Version	JSON capabilities added
SQL Server 2016 (13.x)	`ISJSON`, `JSON_VALUE`, `JSON_QUERY`, `JSON_MODIFY`, `OPENJSON`, `FOR JSON`
SQL Server 2017 (14.x)	Path argument may be a variable
SQL Server 2019 (15.x)	UTF-8 collations for `varchar` storage
SQL Server 2022 (16.x)	`JSON_OBJECT`, `JSON_ARRAY`, `JSON_PATH_EXISTS`, `ISJSON` type argument
Azure SQL, SQL Server 2025	Native `json` type, `JSON_OBJECTAGG`, `JSON_ARRAYAGG`
Preview (at time of writing)	JSON indexes, `JSON_CONTAINS`, `.modify()`, `RETURNING`

A few details deserve emphasis. `OPENJSON` requires the database compatibility level to be 130 or higher, even on a newer server. A database restored from an older system and left at compatibility level 120 will reject `OPENJSON` with a syntax error, which surprises many teams after an upgrade. The other SQL Server 2016 functions do not have this restriction.

UTF-8 collations, introduced in SQL Server 2019, are not a JSON feature as such, but they matter for JSON storage because they allow a `varchar(max)` column to hold the full Unicode range. Chapter 3 discusses when that is a better choice than `nvarchar(max)`.

The features in the last row are described as preview features at the time of writing: the syntax may change before general availability, and they may not be enabled on every service tier or build. We show them where they are relevant, keep those examples short, and label them clearly.

> **NOTE**
>
> Azure SQL Database and Azure SQL Managed Instance are updated continuously, and SQL Server 2025 features can depend on the database compatibility level. Before relying on the native `json` type or the aggregate functions, check the official documentation for your exact version, service tier, and compatibility level.

You can check what you are working with in a few seconds:

check-environment.sql

```
SELECT
    SERVERPROPERTY('ProductVersion') AS ProductVersion,   -- e.g. 16.0.x = SQL Server 2022
    SERVERPROPERTY('Edition')        AS Edition,
    SERVERPROPERTY('EngineEdition')  AS EngineEdition;    -- 5 = Azure SQL Database

SELECT name, compatibility_level
FROM sys.databases
WHERE name = DB_NAME();

-- Raise the level if OPENJSON reports a syntax error (test your workload first!)
-- ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 160;
```

> **WARNING**
>
> Changing the compatibility level affects more than JSON: it can change query optimizer behavior for your entire workload. Treat it as a planned change, test on a copy of production, and use Query Store to catch plan regressions.

A Tour of the Feature Set

Before we set up the sample database, let us take a quick tour of what you can do. Each capability below gets at least one full chapter later; the goal here is to see the whole landscape at once. The examples use a small inline document so you can run them anywhere; run each snippet in the same batch as the `DECLARE` that defines `@doc`.

Validate

`ISJSON` returns 1 when its input is well-formed JSON and 0 when it is not. In SQL Server 2022 and later, a second argument checks the kind of JSON value as well.

```
DECLARE @doc nvarchar(max) = N'{
  "sku": "TSH-ORB-001",
  "color": "navy",
  "sizes": ["S", "M", "L"],
  "specs": { "material": "cotton", "weightGrams": 180 }
}';
```

```
SELECT ISJSON(@doc)          AS IsValid,     -- 1
       ISJSON(@doc, OBJECT)  AS IsObject,    -- 1 (SQL Server 2022+)
       ISJSON(N'{"a": }')    AS IsBroken;    -- 0
```

Read

`JSON_VALUE` extracts a scalar (a string, number, or Boolean) and returns it as `nvarchar(4000)`. `JSON_QUERY` extracts an object or array and returns it as JSON text. `OPENJSON` turns a document into rows, either with a generic key/value/type schema or with a schema you declare.

```
SELECT JSON_VALUE(@doc, '$.color')             AS Color,        -- navy
       JSON_VALUE(@doc, '$.specs.weightGrams') AS WeightGrams,  -- 180 (as text)
       JSON_QUERY(@doc, '$.sizes')             AS Sizes;        -- ["S", "M", "L"]

SELECT s.[key] AS Position, s.value AS Size
FROM OPENJSON(@doc, '$.sizes') AS s;

SELECT p.*
FROM OPENJSON(@doc)
WITH (
    Sku       varchar(40)   '$.sku',
    Color     nvarchar(30)  '$.color',
    Material  nvarchar(50)  '$.specs.material',
    WeightG   int           '$.specs.weightGrams'
) AS p;
```

The last query returns one row with four typed columns. That single pattern, `OPENJSON ... WITH`, is the workhorse for loading JSON into relational tables, and Chapters 7 through 9 explore it thoroughly.

Produce

`FOR JSON` formats the result of any query as JSON. With `PATH` mode, column aliases containing dots create nested objects. SQL Server 2022 adds `JSON_OBJECT` and `JSON_ARRAY` for building documents expression by expression.

```
SELECT 1001           AS [orderId],
       'shipped'      AS [status],
       N'Ada Lopez'   AS [customer.name],
       N'Lisbon'      AS [customer.city]
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER;
-- {"orderId":1001,"status":"shipped","customer":{"name":"Ada Lopez","city":"Lisbon"}}

SELECT JSON_OBJECT('orderId': 1001,
                   'tags': JSON_ARRAY('gift', 'express')) AS Doc;
-- {"orderId":1001,"tags":["gift","express"]}
```

Modify

`JSON_MODIFY` returns a new version of a document with one property set, replaced, removed, or appended to. It does not change the stored value by itself; you use it inside an `UPDATE`.

```
SELECT JSON_MODIFY(@doc, '$.color', 'forest green')       AS Recolored;
SELECT JSON_MODIFY(@doc, 'append $.sizes', 'XL')          AS WithXL;
SELECT JSON_MODIFY(@doc, '$.specs.weightGrams', NULL)     AS NoWeight;
-- In lax mode, setting a property to NULL removes it from the object.
```

Index

SQL Server cannot index the inside of a text column directly, but it can index a computed column that extracts a property. The optimizer then matches queries that use the same expression to the index. Chapter 16 covers this technique; Chapter 17 introduces JSON indexes on the native type.

```
-- Run after creating Orbit Store (next section)
ALTER TABLE catalog.Product
    ADD Color AS CAST(JSON_VALUE(Attributes, '$.color') AS nvarchar(50));

CREATE INDEX IX_Product_Color ON catalog.Product (Color);
```

Store natively

Where the native type is available, you declare a column as `json` and SQL Server validates and stores the document in its binary format. Queries look the same as they do against text.

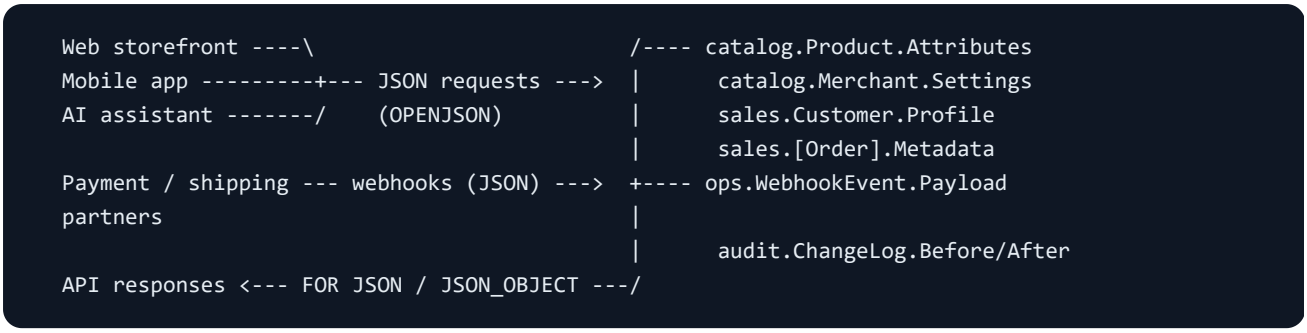
```
-- Azure SQL and SQL Server 2025
CREATE TABLE ops.WebhookEventNative (
    EventId     bigint IDENTITY PRIMARY KEY,
    Source      varchar(50) NOT NULL,
    ReceivedAt  datetime2(3) NOT NULL DEFAULT SYSUTCDATETIME(),
    Payload     json NOT NULL
);
```

Setting Up the Orbit Store Database

Every chapter in this book uses the same running example, so it is worth taking a few minutes to create it now. Orbit Store is a marketplace: merchants list products, customers place orders, and a set of applications talk to the database. There is a web storefront, a mobile app, partner integrations for payments and shipping, and an AI shopping assistant that answers questions such as "find me a waterproof jacket under 120 dollars."

The database has four schemas. `catalog` holds merchants, products, and variants. `sales` holds customers, orders, and order lines. `ops` holds operational data such as incoming webhooks. `audit` holds change history. The design is deliberately hybrid: identifiers, prices, dates, and statuses are typed columns, while attributes, settings, profiles, addresses, metadata, and payloads are JSON.

The following sketch shows where JSON flows in and out of the database. Arrows pointing into SQL Server are documents that arrive and must be validated and stored; arrows pointing out are documents that queries produce for callers.



In other words, JSON is both an input format that must be checked and shredded and an output format that must be shaped precisely. Both directions get equal attention in this book.

The script below uses `nvarchar(max)` for every JSON column because that works on all versions from SQL Server 2016 onward. Each JSON column has a lightweight `CHECK` constraint using `ISJSON` so that malformed documents are rejected at the door. Chapter 4 explains these constraints and shows stricter alternatives.

orbitstore-setup.sql

```
CREATE DATABASE OrbitStore;
GO
USE OrbitStore;
GO
CREATE SCHEMA catalog;
GO
CREATE SCHEMA sales;
GO
CREATE SCHEMA ops;
GO
CREATE SCHEMA audit;
GO

CREATE TABLE catalog.Merchant (
    MerchantId  int IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_Merchant PRIMARY KEY,
    Name        nvarchar(200) NOT NULL,
    Settings    nvarchar(max) NULL
        CONSTRAINT CK_Merchant_Settings CHECK (ISJSON(Settings) = 1)
);

CREATE TABLE catalog.Product (
    ProductId   int IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_Product PRIMARY KEY,
    MerchantId  int NOT NULL
        CONSTRAINT FK_Product_Merchant REFERENCES catalog.Merchant (MerchantId),
    Sku         varchar(40) NOT NULL
        CONSTRAINT UQ_Product_Sku UNIQUE,
    Name        nvarchar(200) NOT NULL,
    Price       decimal(10,2) NOT NULL,
    Attributes  nvarchar(max) NULL
        CONSTRAINT CK_Product_Attributes CHECK (ISJSON(Attributes) = 1)
);
```

```
CREATE TABLE catalog.ProductVariant (
    VariantId   int IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_ProductVariant PRIMARY KEY,
    ProductId   int NOT NULL
        CONSTRAINT FK_Variant_Product REFERENCES catalog.Product (ProductId),
    Sku         varchar(40) NOT NULL
        CONSTRAINT UQ_Variant_Sku UNIQUE,
    Options     nvarchar(max) NULL
        CONSTRAINT CK_Variant_Options CHECK (ISJSON(Options) = 1)
);

CREATE TABLE sales.Customer (
    CustomerId  int IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_Customer PRIMARY KEY,
    Email       nvarchar(320) NOT NULL
        CONSTRAINT UQ_Customer_Email UNIQUE,
    Profile     nvarchar(max) NULL
        CONSTRAINT CK_Customer_Profile CHECK (ISJSON(Profile) = 1)
);

CREATE TABLE sales.[Order] (
    OrderId         int IDENTITY(1000,1) NOT NULL
        CONSTRAINT PK_Order PRIMARY KEY,
    CustomerId      int NOT NULL
        CONSTRAINT FK_Order_Customer REFERENCES sales.Customer (CustomerId),
    OrderDate       datetime2(0) NOT NULL
        CONSTRAINT DF_Order_OrderDate DEFAULT SYSUTCDATETIME(),
    Status          varchar(20) NOT NULL,
    Total           decimal(12,2) NOT NULL,
    ShippingAddress nvarchar(max) NULL
        CONSTRAINT CK_Order_ShipAddr CHECK (ISJSON(ShippingAddress) = 1),
    Metadata        nvarchar(max) NULL
        CONSTRAINT CK_Order_Metadata CHECK (ISJSON(Metadata) = 1)
);

CREATE TABLE sales.OrderLine (
    OrderId     int NOT NULL
        CONSTRAINT FK_OrderLine_Order REFERENCES sales.[Order] (OrderId),
    [LineNo]    int NOT NULL,       -- LINENO is a reserved word: keep the brackets
    ProductId   int NOT NULL
        CONSTRAINT FK_OrderLine_Product REFERENCES catalog.Product (ProductId),
    Quantity    int NOT NULL,
    UnitPrice   decimal(10,2) NOT NULL,
    CONSTRAINT PK_OrderLine PRIMARY KEY (OrderId, [LineNo])
);

CREATE TABLE ops.WebhookEvent (
    EventId     bigint IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_WebhookEvent PRIMARY KEY,
    Source      varchar(50) NOT NULL,
    ReceivedAt  datetime2(3) NOT NULL
        CONSTRAINT DF_Webhook_ReceivedAt DEFAULT SYSUTCDATETIME(),
    Payload     nvarchar(max) NOT NULL
        CONSTRAINT CK_Webhook_Payload CHECK (ISJSON(Payload) = 1)
);
```

```
CREATE TABLE audit.ChangeLog (
    ChangeId    bigint IDENTITY(1,1) NOT NULL
        CONSTRAINT PK_ChangeLog PRIMARY KEY,
    TableName   sysname NOT NULL,
    KeyValue    nvarchar(100) NOT NULL,
    ChangedAt   datetime2(3) NOT NULL
        CONSTRAINT DF_ChangeLog_ChangedAt DEFAULT SYSUTCDATETIME(),
    Before      nvarchar(max) NULL
        CONSTRAINT CK_ChangeLog_Before CHECK (ISJSON(Before) = 1),
    After       nvarchar(max) NULL
        CONSTRAINT CK_ChangeLog_After CHECK (ISJSON(After) = 1)
);
GO
```

Notice that the `CHECK` constraints allow `NULL`. `ISJSON(NULL)` returns `NULL`, and a `CHECK` constraint only rejects a row when its condition evaluates to false, so a missing document passes. If a document is mandatory, declare the column `NOT NULL`, as we did for the webhook payload.

Also note the brackets around `[Order]` and `[LineNo]`. `ORDER` and `LINENO` are reserved words in T-SQL, so both names must always be delimited. We keep them because they are the natural names for these tables, and the brackets are a small price for readability.

Loading sample data

Next, add a small but realistic data set. The documents are intentionally varied: the shoe has sizes and a drop measurement, the laptop has nested specs, and the coffee has an array of tasting notes. This variety is exactly why the attributes live in JSON.

orbitstore-sample-data.sql

```
USE OrbitStore;
GO
INSERT INTO catalog.Merchant (Name, Settings) VALUES
(N'Trailhead Outfitters', N'{
  "returnWindowDays": 30,
  "shipping": { "zones": ["EU", "UK"], "freeOver": 75.00 },
  "notifications": { "email": true, "sms": false }
}'),
(N'Bytewise Electronics', N'{
  "returnWindowDays": 14,
  "shipping": { "zones": ["EU", "US"], "freeOver": null },
  "notifications": { "email": true, "sms": true }
}'),
(N'Roast & Ritual', N'{
  "returnWindowDays": 0,
  "shipping": { "zones": ["EU"], "freeOver": 40.00 },
  "subscriptions": { "enabled": true, "intervalsDays": [14, 28] }
}');

INSERT INTO catalog.Product (MerchantId, Sku, Name, Price, Attributes) VALUES
(1, 'TRL-SHOE-042', N'Ridge Runner Trail Shoe', 129.00, N'{
  "category": "footwear",
  "color": "slate",
```