A PRACTICAL GUIDE FOR BACKEND DEVELOPERS

PostgreSQL for Everything

Build a Multi-Tenant B2B SaaS, from Schema to Production

EVERY ROW BELONGS TO A TENANT



Agostinho Sousa

PostgreSQL 18 · PostgREST 16 · Python

Free sample · two complete chapters

A PRACTICAL GUIDE FOR BACKEND DEVELOPERS

PostgreSQL for Everything

Build a Multi-Tenant B2B SaaS, from Schema to Production

Agostinho Sousa

First Edition · Version 1.0

Free sample · preface and two complete chapters

PostgreSQL for Everything
Build a Multi-Tenant B2B SaaS, from Schema to Production

Copyright © 2026 Agostinho Sousa. All rights reserved.

First Edition, version 1.0, September 2026.

No part of this book may be reproduced or distributed without the written permission of the author, except for brief quotations in reviews. The companion source code is provided for use in your own learning and projects under the terms included with it.

The author has made every effort to ensure the accuracy of the information in this book. The software, configurations and procedures are provided as is, without warranty of any kind. Measurements describe the recorded experiments only. Test any procedure in your own environment before relying on it with production data.

PostgreSQL® and Postgres® are registered trademarks of the PostgreSQL Community Association. PostgREST, Python and other names are the property of their respective owners. This book is an independent publication and is not affiliated with or endorsed by any of them.

Typeset in Source Serif 4, Inter and JetBrains Mono.

Contents

The chapters in grey are part of the full book. This free sample contains the preface and two complete chapters.

Who This Book Is For 6

Preface 9

PART I **A Foundation We Can Trust**

1 Two Companies, One Application 12

2 The First Executable Slice

3 Who Owns This Row?

4 Identity Is Not Permission

5 An API Is a Contract

PART II **Business Rules Under Pressure**

6 An Order Is More Than a Row

7 Two Orders, One Item Left 19

8 The Customer Clicked Twice

9 Cancel, Release, Reverse

PART III **More Than Relational Storage**

10 Work That Survives a Restart

11 Tell the Outside World

12 Find the Product the User Meant

13 Every Customer Wants Another Field

14 Who Changed This Order?

15 Reports Without Guesswork

PART IV **Operate, Measure, and Evolve**

16 One Tenant Gets Busy

17 Change the Schema While the Product Lives

18 Restore Before You Need To

19 Ship a System You Can Explain

20 When PostgreSQL Is No Longer Enough

REFERENCE **Appendices**

A SQL and PL/pgSQL Conventions

B Database Roles and Privileges

C API Examples and Error Conventions

D Reproducing the Experiments

E Glossary and Further Reading

Who This Book Is For

You can already build an API. You know tables, primary keys, joins and the everyday SQL an ORM writes for you. What you have not yet had is a project that forces the harder questions:

- How do you stop one customer's request from ever reading another customer's rows?
- What happens when two people try to buy the last item at the same moment?
- What should the server do when a client retries a request that already succeeded?
- How does work survive a crashed worker, and how do you notify another system without lying about what happened?
- How do you change the schema while old clients are still calling you?
- How do you know that a backup will actually give you a working product back?

This book answers those questions by building one application from an empty database to a release you can hand to someone else. It is written for backend and full-stack developers who are comfortable in a terminal, can read Python and SQL, and want to understand the data layer of a real business system rather than copy another CRUD tutorial.

You do not need to be a database administrator or a PL/pgSQL expert. Every PostgreSQL feature is introduced at the moment the application needs it, and always with a test that shows what it does and what it does not guarantee.

What You Will Build

The running project is **Ledger & Stock**, a sales and inventory application delivered as a multi-tenant SaaS. Two companies, Northstar and Harbor, share the same application and the same database. Their people sign in, select an organization and work only with that organization's customers, products, orders and stock.

By the last chapter the system has:

- **Tenant isolation enforced by the database.** Row-level security, membership-based authorization and composite keys that keep every relationship inside its tenant.
- **A contract-first HTTP API.** PostgREST in front of a dedicated `api` schema, with explicit commands, versioned edits and structured errors.
- **Correct behaviour under concurrency.** Stock reservations that cannot oversell, consistent lock ordering and optimistic version checks.
- **Safe retries.** Idempotency keys whose saved results can be replayed without repeating a stock movement or an order.
- **Durable background work.** A PostgreSQL job queue with leases, fencing tokens, bounded retries and a separate worker.
- **Reliable integrations.** A transactional outbox and signed webhooks, with a receiver that deduplicates repeated deliveries.
- **Search, flexible fields, audit and reports.** Full-text and typo-tolerant product search, validated JSONB custom fields, an audit trail with a real actor model, and operational reports.
- **Operations you can explain.** Request budgets, live schema migrations, a tested backup-and-restore rehearsal, a restricted operator snapshot and a verifiable release package.

What You Will Be Able to Do

After working through the book you will be able to:

1. Design a multi-tenant schema whose integrity does not depend on every query remembering a `WHERE` clause.
2. Put a business rule in the right place: constraint, policy, transaction, API command or worker.
3. Reproduce a race condition on purpose, fix it and prove the fix with two real clients.
4. Make commands retry-safe and explain exactly which failures a retry can and cannot repair.
5. Run background jobs and outbound events without pretending that two systems commit together.

6. Evolve a live schema in small, reversible steps and verify each one through the API.
7. Rehearse a restore that checks permissions and business behaviour, not just row counts.
8. Recognize, from measurements, when a responsibility should move out of PostgreSQL.

What You Need

- A macOS or Linux machine where you can install software.
- **PostgreSQL 18** with the `pg_trgm` extension and the standard client tools, **PostgREST 16** and **Python 3.10 or newer**. The code was verified end to end with PostgreSQL 18.6 and PostgREST 16.3.
- Basic SQL, basic HTTP and comfort reading short Python scripts. No framework, ORM or Python package is required to run the examples.

The book is also useful on its own if you only want to read. Every chapter explains the design, shows the important code and reports what the tests demonstrated.

Preface

A customer places an order for the last item in stock. At almost the same moment, another customer places an order for that same item.

Both requests look valid when they arrive. Both users expect an answer. Somewhere in the system, a decision must be made about which order can proceed.

Now add a second company using the same application. Its customers, products, and stock must remain separate from the first company's data. A request is retried after a timeout. A background process stops halfway through a task. Someone asks who changed an order yesterday.

These are the kinds of problems we will solve in this book.

Our application will be small enough to understand, but its behavior will have to survive more than a successful demonstration. We will build it around PostgreSQL, expose its primary HTTP API through PostgREST, and use a small worker for work that belongs outside a database transaction.

What “Everything” Means

The title proposes an architectural question: how much of a business application's data infrastructure can PostgreSQL reasonably provide?

We will use it for relational storage and integrity, tenant isolation, transactional business operations, persistent background jobs, search, flexible attributes, audit records, and reporting. Each use must earn its place by solving a problem in the application.

Consolidation has a cost. Workloads share resources. Database functions need tests and version control. A slow report can compete with an order being placed. A queue introduces cleanup and retry concerns. Moving a responsibility into PostgreSQL does not make that responsibility disappear.

We will keep those costs visible. Where a separate service would offer a better boundary or a required capability, we will explain the reason. Authentication, external communication, and the user interface will still have their own components.

The aim is to give you a basis for choosing a small architecture deliberately, and for recognizing when it needs to change.

Who This Book Is For

You are a backend or full-stack developer who can already build a simple API. You understand tables, primary keys, joins, and basic SQL statements. You may have used PostgreSQL primarily through an ORM.

You now need to make decisions that a CRUD tutorial leaves open:

- Where should a business rule be enforced?
- How do you keep one organization's data out of another organization's requests?
- What happens when two valid operations cannot both succeed?
- Which parts of a failed operation can be retried?
- How do you know a backup is useful?

We will introduce the PostgreSQL-specific concepts needed to answer these questions. You do not need to arrive as a database administrator or a PL/pgSQL expert. You should be comfortable reading code, working in a terminal, and following an HTTP request into the data layer.

The Application

Our running example is **Ledger & Stock**, a small sales and inventory system for businesses that sell physical products. Despite its name, the application does not implement a general accounting ledger.

Each business subscribes to the same SaaS application. Its staff manage customers and products, create orders, reserve inventory, record shipments, and correct erroneous dispatches. They also need a useful history of changes and a view of current sales and stock.

This gives us a connected set of problems. Search helps staff find products. Background jobs deliver order notifications. Reports summarize the orders we already store. Audit records explain the changes we already make.

The project deliberately excludes payroll, manufacturing, full accounting, tax-compliant invoicing, payment recording, and payment processing. Those domains have rules substantial enough to deserve their own treatment.

How to Read This Book

Read the implementation chapters in order on your first pass. Each builds on decisions and code introduced earlier.

The recurring pattern will be:

1. **A business problem.** What must the application do, and what can go wrong?
2. **A design.** Which guarantees belong in the data model, the transaction, the API, or the worker?
3. **An implementation.** A small addition to the same application.
4. **An experiment.** A test that challenges the guarantee, including failures where they matter.
5. **A boundary.** What does the solution assume, and when would we change it?

The companion code contains an executable checkpoint for every implementation chapter, with its migrations, demonstration data and tests. Each checkpoint was run end to end against real PostgreSQL and PostgREST processes, and its results are recorded next to the code.

We will use performance measurements to answer specific questions. Any published measurements will include the conditions needed to interpret and reproduce them. A number from one machine is evidence about that experiment, not a promise about every deployment.

By the end, you should be able to trace a business operation from its HTTP request to its database effects, explain why those effects remain correct under the tested failure conditions, and identify what the system still cannot guarantee.

CHAPTER 1

1 Two Companies, One Application

Northstar Office Supplies sells desks, lamps, and other equipment to small businesses. Its staff use a spreadsheet to track available stock and another to track orders. When someone reserves the last desk, the two spreadsheets do not always agree.

Harbor Workshop sells tools and workshop equipment. It has a similar problem and wants the same software. It has no reason to see Northstar's customers, orders, or prices.

We will give both companies access to one application: Ledger & Stock.

This chapter establishes the product and the rules that will guide its implementation. It contains design examples, not an executable checkpoint. In the next chapter, we will turn the first slice into a running system.

1.1 Define the Tenant

A **tenant** is an organization that uses the application and owns a distinct set of business data. Northstar is one tenant. Harbor is another.

A user is a person with an authenticated identity. A **membership** connects that person to an organization and records their role within it.

These are separate concepts. A consultant might work for both companies. That person should be able to select either organization, but an action performed for Northstar must not accidentally use Harbor's records.

For the main business API, every request will operate in one active tenant context. The server must verify that the authenticated user has an active membership in the selected organization. Sending an organization identifier in a request expresses a choice; it does not establish permission.

We will begin with a shared database and shared business tables. Tenant-owned rows will carry an organization identifier. This keeps the first implementation approachable and lets us study isolation in the model, permissions, and API together.

The choice also means tenants share database resources and a recovery environment. We will return to those consequences when we test uneven workloads and recovery procedures.

1.2 Choose One Complete Business Journey

The application's central workflow is:

Create a customer → draft an order → confirm and reserve → ship → correct an erroneous shipment when necessary.

Payment recording and payment processing are outside this implementation. The example concentrates on the order and inventory lifecycle.

For inventory, we will start with one stock location per tenant and whole-number item quantities. A product belongs to one tenant. These choices remove units-of-measure conversion and warehouse allocation from the first model.

An order will progress through a small set of business states:

State	Meaning
Draft	An owner may change the order; it holds no stock reservation.
Confirmed	Stock has been reserved for all order lines.
Shipped	The reserved goods have left inventory.
Cancelled	The order will not be fulfilled; any reservation has been released.

For this first version, confirmation succeeds for the entire order or fails. We will not support partial reservations, partial shipments, or backorders. A future payment module would need its own state and reconciliation rules.

These rules make the project smaller while preserving the difficult question: can we keep a promise about stock when requests overlap?

1.3 State the Guarantees Before Writing Endpoints

A useful design begins with statements we can try to disprove.

A customer belongs to the same tenant as its order

Suppose an order is owned by Northstar but refers to a customer owned by Harbor. Even if the application normally filters customer lists correctly, the stored relationship is invalid.

Our model must reject that relationship. Later, we will use tenant-aware keys and foreign keys so the database can check ownership across related records.

A member can act only within an authorized organization

Knowing an order identifier must not grant access to it. We will check this through the actual API, using the same restricted role as ordinary application requests.

Tests executed only as a database administrator would tell us little about this boundary. PostgreSQL's row security rules treat privileged roles differently, so our test identity is part of the test itself. PostgreSQL: Row Security Policies

Confirmation cannot reserve unavailable stock

For a product, we will distinguish:

- **On hand:** units physically held by the business according to recorded inventory movements.
- **Reserved:** units committed to confirmed, unshipped orders.
- **Available:** on-hand units minus reserved units.

If Northstar has one lamp on hand and none reserved, it has one lamp available. Two overlapping requests may both want it. At most one may successfully reserve that unit.

Checking availability in a browser and submitting an update later cannot establish this guarantee. The competing operations must be coordinated where the stock decision is made.

A failed confirmation leaves no partial confirmation

If an order contains a lamp and a desk, but the desk is unavailable, our initial product rules require the whole confirmation to fail. The lamp must not remain reserved by the failed attempt.

The order state, reservations, and event recording therefore belong to one transaction.

A repeated request does not create a second business effect

A client may time out after the server commits. The user sees no answer and tries again.

We will define how confirmation recognizes a repeated operation and how it responds. The later idempotency chapter will cover both the key used to identify a retry and what happens if that key is reused with different input.

1.4 Give Each Component a Job

Our initial architecture has five components:

Component	Responsibility
Frontend	Present the workflow, collect input, and display results.
Identity provider	Authenticate people and issue tokens the API can verify.
PostgREST	Translate authorized HTTP requests into database operations.
PostgreSQL	Store data, enforce integrity and access rules, and execute transactional business operations.
Worker	Perform background tasks and communicate with external systems.

PostgREST can validate JWTs issued by an external identity service. Our application will still need its own membership and permission rules: authentication tells us who is making a request; authorization determines what that identity may do. PostgREST: Authentication

The worker will use narrowly scoped permissions appropriate to its tasks. It will not inherit unrestricted access simply because it runs in the background.

1.5 Design a Business Operation

Consider an API design that asks the frontend to perform four requests:

1. Check available stock.
2. Create reservations.
3. Change the order state.
4. Create a notification task.

A failure between those requests could leave a reservation without a confirmed order. Another request could change availability after the first check.

Instead, we will expose one business command:

```
POST /rpc/confirm_order
Content-Type: application/json
Authorization: Bearer <access-token>
X-Tenant-Id: 10000000-0000-0000-0000-000000000001
Idempotency-Key: cb5c9631-5d82-4fbd-a06b-57b88b411e43

{
  "order_id": "4ae53e21-6d84-4c60-902d-b37e52feb0f3",
  "expected_version": 1
}
```

This previews the keyed contract completed in Chapter 8. Use the actual order ID and current version; earlier checkpoints introduce the operation before adding its retry key.

The database function behind the command will verify the caller's permissions, check the order's state, coordinate access to inventory, establish reservations, and record the resulting event. PostgREST executes a request inside a database transaction, which gives this operation a boundary within which its database changes can commit or roll back together. PostgREST: Transactions

The operation will record durable notification work for the worker to process after commit. Our implemented external-delivery example will use a signed webhook.

This separation matters when external systems fail. An unavailable webhook receiver should not determine whether a valid stock reservation commits. Conversely, commit-

ting an order does not prove that the receiver has accepted its event. We will represent and test those outcomes separately.

1.6 Keep the Public Contract Deliberate

Our intended schema layout is:

Schema	Intended contents
`app`	Business tables and internal business functions.
`api`	Deliberately exposed resources and callable operations.
`private`	Internal authorization helpers and background-processing details.

A schema name does not provide security by itself. Exposure configuration, object privileges, function execution permissions, and row security must agree.

We will give ordinary requests access to the intended API surface. Simple edits, such as changing a product description, can use a resource endpoint. Actions that cross several rules, such as confirming an order, will have a named operation.

Views and privileged functions need particular care because their execution context can change how access checks behave. We will choose those settings explicitly and verify their behavior through cross-tenant tests before treating them as safe API resources.

1.7 Build Evidence Alongside Features

The companion implementation will use two named tenants from the beginning. A single-tenant fixture would make many isolation mistakes invisible.

For each significant guarantee, we will keep a corresponding experiment:

Guarantee	Experiment
Tenant isolation	Ask a Northstar user to read and modify Harbor's records.
Valid ownership relationships	Attempt to attach a Harbor customer to a Northstar order.
No overselling	Confirm two orders concurrently against one available unit.
Atomic confirmation	Fail one line of a multi-line order and inspect all resulting state.

Guarantee	Experiment
Safe retries	Repeat a confirmation with the same idempotency key.
Recoverable background work	Stop a worker during a task and observe recovery and possible redelivery.

These experiments have not been run yet. They define the evidence we need as we implement the application. When a chapter claims that a guarantee holds, its checkpoint should make that claim reproducible.

1.8 Know What This Design Does Not Prove

A shared schema does not isolate each tenant's CPU, memory, storage, or backup lifecycle. Row security does not defend against every action available to a database administrator. A durable task record does not make an external side effect occur exactly once.

These boundaries affect the product we can offer. We will carry them forward into operational chapters, where we can examine workload controls, privileged access, recovery, and delivery semantics with a running application.

For now, we have enough definition to begin: two companies, one workflow, explicit tenant ownership, and business guarantees that can be challenged by tests.

In Chapter 2, we will start PostgreSQL and PostgREST, create our first versioned migration, and make the first request against the application.

CHAPTER 7

7 Two Orders, One Item Left

> This is a complete chapter from the full book. It builds on the tenant model, API contract and order aggregate of the earlier chapters.

Northstar has one desk available. Two customers each order one desk. Both orders are valid drafts, and both requests can read the same available quantity before either confirmation commits.

Only one order can receive a reservation.

This is where a stock display becomes a stock promise. A read tells us what was visible at that moment. Confirmation must make a decision that remains correct when another transaction wants the same unit.

In this chapter, we introduce receipt movements, product balances, and reservations. The checkpoint (`examples/chapter-07/README.md`) reproduces the last-unit race using two real HTTP clients against PostgreSQL and PostgREST.

7.1 Distinguish Physical Stock from Promised Stock

For each product in an organization, we track:

Quantity	Meaning
On hand	Recorded units physically held by the business.
Reserved	Units committed to confirmed, unshipped orders.

Quantity	Meaning
Available	On hand minus reserved.

If one desk is on hand and none is reserved, one is available. Confirming a one-desk order changes reserved to one and available to zero. On hand remains one: nothing has shipped.

This distinction prevents confirmation from pretending that a physical movement occurred. Shipping will reduce physical stock later. For now, the only physical movements we support are incoming receipts.

We keep one logical stock location per tenant and use whole item quantities. Multiple warehouses, transfers, damaged stock, and unit conversion would add separate allocation rules.

7.2 Store the Balance and Its Explanation

The model has three new tables:

- `inventory_balances`: one current balance per tenant/product pair.
- `inventory_movements`: positive receipt records in this checkpoint.
- `stock_reservations`: quantities committed by confirmed order lines.

The balance table contains a stored generated value for available stock and a check:

```
available bigint
GENERATED ALWAYS AS (on_hand - reserved) STORED,

CONSTRAINT inventory_reservation_bounds
CHECK (reserved >= 0 AND reserved <= on_hand)
```

We also require nonnegative on-hand stock. These row-local constraints reject an impossible balance regardless of which ordinary write attempts to store it.

A reservation references both its inventory balance and the matching tenant/order/product combination in `order_lines`. Its primary key permits one reservation per order and product.

The balance makes the contested quantity a small, lockable row. The movement and reservation records explain how the application reached it. This does create values that must remain consistent across tables. Our commands update them transactionally, and reconciliation tests compare them afterward.

The database does not infer cross-table equality simply because the tables have meaningful names. Arbitrary privileged SQL could still update a balance without the corresponding record.

7.3 Receive Stock as One Operation

The incoming-stock command is:

```
POST /rpc/receive_stock

{
  "product_id": "50000000-0000-0000-0000-000000000001",
  "quantity": 3,
  "reference": "Delivery 001"
}
```

The normal authentication, tenant, and JSON headers apply. An owner supplies a visible product, a positive quantity, and a short reference. The reference is descriptive; it is not unique and does not deduplicate a delivery.

The command creates a zero balance if none exists, increments on-hand stock, and appends the receipt movement in the same transaction. The update adds to the current stored quantity; it does not write a previously calculated absolute balance from a browser.

The stock row's update coordinates with concurrent updates to the same row. Its reservations remain unchanged, so the newly received units increase availability.

This first movement model contains positive receipts only. We will extend its semantics when shipping and reversal enter the application. It would be misleading to describe the current table as a complete inventory ledger.

7.4 Why a Separate Availability Check Is Insufficient

Consider this sequence:

Time	Request A	Request B
1	Reads available = 1	
2		Reads available = 1
3	Decides its order can proceed	
4		Decides its order can proceed

Both decisions were based on a valid observation. The observations did not reserve anything.

Adding a transaction around each request does not automatically make every read exclude competing writers. Under PostgreSQL's default READ COMMITTED isolation, we need an explicit coordination step for this operation. See Transaction Isolation.

We will lock the balance before checking and increasing reserved stock. A second confirmation must wait for that row and then decide using the value it can lock, rather than its earlier client-side observation.

The availability displayed in the UI remains useful information. It is not an authorization to bypass the confirmation check.

7.5 Keep the Order Lock and Add Product Locks

The confirmation body contains the order and its expected version:

```
{
  "order_id": "60000000-0000-0000-0000-000000000001",
  "expected_version": 1
}
```

The function starts with the same draft-order lock used by editing and draft cancellation. This establishes that the caller is an owner, the order is visible and still a draft, and its version matches.

It also checks that at least one line exists. Public creation already requires lines, but the confirmation boundary should reject an empty header left by an import or administrative operation rather than confirm it accidentally.

For each product, the command executes a locking read:

```
SELECT b.* INTO balance
FROM app.inventory_balances AS b
WHERE b.organization_id = private.current_tenant_id()
  AND b.product_id = line.product_id
FOR UPDATE;
```

If the balance is missing or available stock is less than the line quantity, the command raises a stock conflict. Otherwise it increases reserved stock and inserts the reservation.

The row lock is held until transaction completion. Under the tested isolation level, a competing locking read waits and can then return the updated row. See Explicit Locking.

Once every line has been reserved, the function sets the order to confirmed and increments its version. The response uses the order document introduced in Chapter 6.

7.6 Acquire Product Locks in a Consistent Order

One order might list a desk before a lamp; another might list the lamp before the desk. If their confirmations locked products in document order, they could acquire opposite locks and then wait for each other.

The function processes lines using `ORDER BY product_id`, so every confirmation follows the same UUID ordering for balance locks. Document presentation order does not determine lock order.

This addresses a known source of deadlocks between these confirmations. It is not a claim that the system can never deadlock. Future receipt, shipment, cancellation, or maintenance operations must be examined together with this lock hierarchy.

The current path is: lock one order header, then its product balances in a consistent order. Receipt changes only one product balance and does not subsequently acquire an order lock.

We do not implement a generic deadlock retry handler here. Adding retries without deciding which operations are safe to repeat would move the problem into request semantics.

7.7 Reserve Every Line or None

Northstar's opening fixture has one desk and three lamps. Create an order for one desk and four lamps.

In product UUID order, confirmation first reserves the desk. It then reaches the lamp and discovers the shortage. At that point, some work has already occurred inside the transaction.

The exception aborts the request. The test verifies all four outcomes:

- The desk's reserved counter returns to zero.
- No reservation row remains for the failed order.
- The order stays in draft.
- Its version remains unchanged.

This is a concrete all-or-nothing reservation test. We are not only validating availability before any write; we are proving the observed rollback after an earlier line was processed.

The same transaction includes the final status update. A caller should never receive a committed confirmed order whose lines were only partly reserved through this command.

7.8 Run the Last-Unit Race

The central experiment creates two distinct drafts, each requesting the one available desk.

Each client first reads the stock endpoint and confirms that it sees one available unit. A barrier then releases both confirmation attempts. PostgREST has two request-pool connections, so this experiment is not configured to serialize all requests through one connection.

The observed result is:

Outcome	Winning order	Other order
HTTP result	200	409
Order state	Confirmed	Draft
Order version	2	1
Reserved desk quantity	1	0

The shared balance is on hand 1, reserved 1, available 0. The reservation belongs to exactly the winning order.

Which request wins is deliberately unspecified. The test asserts the business outcome, not fairness or a specific scheduler interleaving.

We also submit two confirmations for the same draft. The order-header lock and state check allow one confirmation, while the other conflicts. The reservation is not duplicated.

7.9 Handle the Resulting State Honestly

An insufficient-stock response does not modify the draft. After a receipt adds stock, the same draft can be submitted again using its unchanged version.

An already confirmed order behaves differently: repeating confirmation produces a non-draft conflict. It does not replay the original success response. Chapter 8 will distinguish a repeated logical request from a genuinely new attempt.

Confirmed lines are no longer editable through the draft command. Draft cancellation also refuses confirmed orders, because it would need to release reservations. That workflow is intentionally postponed until we implement release and reversal in Chapter 9.

Physical stock remains on hand after confirmation. No receipt or outbound movement is created merely because a unit is promised to an order.

7.10 Reconcile the Redundant Values

The final test compares the stored balances with their supporting records:

```
on_hand = sum(receipt quantities)
reserved = sum(reservation quantities)
available = on_hand - reserved
```

It also verifies that every confirmed order line has a reservation of the same quantity.

The first equation is specific to this receipt-only checkpoint. Once shipments exist, the movement model must include their effect. The equality is not a permanent definition that ignores future outbound stock.

These checks help distinguish a nonnegative balance from a correctly explained balance. A reserved value can satisfy its bounds and still disagree with reservation records if a write path updates only one representation.

The API does not expose balance, movement, or reservation writes directly. All three views are readable under tenant policies, while mutations go through commands. The existing trusted-SQL boundary remains important: reconciliation is evidence about the implemented paths, not protection against arbitrary administrator changes.

7.11 Try the Checkpoint

Run:

```
python3 examples/chapter-07/run.py
```

The runner executes the retained SQL checks and 156 HTTP-driven checks across authentication, customer editing, orders, and inventory, followed by the inventory SQL reconciliation. Scenario counts are not the number of individual requests: several tests read before and after a command.

The verification record (`examples/chapter-07/VERIFICATION.md`) records the environment and results. The README includes a manual exercise: receive one desk, then confirm the seeded two-desk order. Its resulting stock is on hand 2, reserved 2, available 0.

There are no partial reservations, shipments, multiple warehouses, or idempotency keys yet. In particular, sending the same receipt twice increases stock twice. A reference string does not change that behavior.

We can now make a stock promise under competing demand. Next we must handle a different kind of repetition: the customer clicked twice, or the network lost a successful response.

END OF THE FREE SAMPLE

Eighteen chapters to go

You have just read the design chapter and the concurrency chapter. The rest of the book takes Ledger & Stock from that point to a release another engineer can run and operate: idempotent commands, cancellation and shipment corrections, a durable job queue, signed webhooks from a transactional outbox, search, JSONB custom fields, an audit trail, reports, request budgets, live schema migrations and a backup-and-restore rehearsal that checks permissions and behaviour.

The complete edition

- 20 chapters and 5 reference appendices.
- The complete companion code: one self-contained, runnable checkpoint per implementation chapter.
- A final release with 22 ordered migrations, workers, an operator snapshot and a release packager, covered by more than 500 test scenarios.
- The recorded verification results and measurement reports behind every claim in the book.

Get it at the book's page

PostgreSQL for Everything · free sample of the first edition · version 1.0