DataShelf Hub

BACKEND ENGINEERING

Building Event-Driven Backends

Queues, outboxes and sagas for systems that never lose a message

FREE SAMPLE CHAPTER

Omar El Alaoui
Data & systems engineer

PDF · CH. 01

ABOUT THIS SAMPLE

A free first chapter

What follows is Chapter 1 of *Building Event-Driven Backends*, reproduced in full and provided at no cost by DataShelf Hub. It is unabridged — nothing has been trimmed or summarised. If it's useful to you, the remaining eight chapters continue exactly where this one leaves off.

This sample is for your personal use while you decide whether the book is right for you. Please don't republish or redistribute it — share a link to the book's page instead.

TITLE	Building Event-Driven Backends
AUTHOR	Omar El Alaoui
SHELF	Backend Engineering
FORMAT	PDF, DRM-free, 300 pages
THIS SAMPLE	Chapter 1 of 9 — 14 pages
PUBLISHER	DataShelf Hub · datashelfhub.com

WHAT'S IN THE FULL BOOK

Table of contents

9 chapters · 27 sections

Continued on the next page →

TABLE OF CONTENTS, CONTINUED

Chapters 5–9

CHAPTER ONE

01

When Direct Calls Stop Working

Why the simplest way to connect two services — calling one directly — is also the way that takes both of them down together, and what changes when you stop.

It starts as the obvious choice. Service A needs something from service B, so A calls B, waits for the response, and continues. It's simple to write, simple to trace, simple to explain in a design review. For a while, it's also simply correct — right up until service B has a slow day, and now service A is slow too, and so is everything that calls A, and an incident that started as one service having a bad afternoon has become three teams' problem before anyone has finished their coffee.

This chapter is about the moment that trade-off stops being worth it, and what you're actually signing up for when you decide it has. Event-driven architecture doesn't remove failure from your system — nothing does. It changes where failure is allowed to stop, and that change is worth understanding before you commit to it, because it isn't free and it isn't always the right call.

1.1 Coupling and Cascading Failure

A direct call creates two kinds of coupling at once, and only one of them is obvious. The obvious kind is that A now depends on B's API — a contract problem, and a familiar one. The less obvious kind is temporal coupling: A's request is now blocked for as long as B takes to respond, whether B takes ten milliseconds or ten seconds, whether B succeeds or is in the middle of failing.

That temporal coupling is what turns a local problem into a system-wide one. If B gets slow — a slow query, a garbage-collection pause, a downstream dependency of its own acting up — A's threads or connections start piling up waiting on it. If A has a bounded pool, which it should, that pool exhausts, and now A can't serve any request, including the ones that never needed B at all. The failure didn't stay in B. It propagated, because the call chain made it propagate by construction.

> *"A synchronous call chain isn't just an API contract — it's an agreement to fail together, at whatever speed the slowest link fails at."*

Add a third service C, calling A, which calls B. Now C's users experience B's bad afternoon as C being down, and none of C's engineers have any way to know that from where they're sitting — the failure arrived as "service C times out," full stop. This is cascading failure, and it scales with the depth of your call graph: the more synchronous hops between a request and the service that's actually struggling, the more blast radius that struggle gets for free.

None of this means synchronous calls are wrong. Plenty of interactions genuinely need an immediate answer — you can't checkout a cart without knowing, right now, whether the payment succeeded. The point isn't to eliminate synchronous calls; it's to notice when you've reached for one out of habit, for an interaction that didn't actually need an answer before moving on.

A QUICK TEST: DOES THIS CALL NEED TO BE SYNCHRONOUS?

Needs an answer now	checkout, authentication, anything the user is blocked on
Just needs to happen	send a receipt email, update a search index, notify an analytics pipeline
The tell	if the caller doesn't use the response, it didn't need to wait for one

That second row — work that just needs to happen, eventually, without the caller blocking on it — is where event-driven design earns its complexity. Sending a receipt email doesn't need to hold up the checkout response. Updating a search index doesn't need the write path to wait for it. In both cases, the calling service can record that something happened and move on, letting a separate consumer act on that fact whenever it's ready to.

That's the shift the rest of this chapter walks through: from "call the thing that needs to happen next" to "record what happened, and let interested parties react to it." It sounds like a small change in wording. It is not a small change in what your system is allowed to do when one part of it is having a bad day.

1.2 The Shape of an Event System

Strip away the specific technology and an event-driven system has a consistent shape: a producer records a fact — `OrderPlaced`, `PaymentCaptured`, `InventoryReserved` — onto a durable log or broker, and any number of consumers read that fact independently, on their own schedule, and react to it. The producer doesn't know or care who's listening. Consumers don't block the producer, and they don't block each other.

That independence is the entire point. If the email-sending consumer is down, orders still get placed — emails just send late, once it recovers. If the search-indexing consumer is slow today, checkout doesn't get slow with it. Each consumer's problems stay that consumer's problems, instead of propagating backward through a call chain to the thing that started it all.

SAME INTERACTION, TWO ARCHITECTURES

Synchronous	checkout calls email service, calls search service, calls analytics — all inline, all blocking
Event-driven	checkout emits OrderPlaced once; email, search and analytics consume it independently, whenever they're ready

The trade is real, and this book won't pretend otherwise. You gain independence between producer and consumers, at the cost of immediacy — a consumer might process an event a second later, or a minute later, or an hour later if it's been down. You also take on new problems that a direct call never had: a message can be delivered more than once, messages can arrive out of order, and "did this actually get processed" stops being a question the caller can answer just by looking at a response.

Every remaining chapter in this book is about handling those new problems properly — delivery guarantees in Chapter 3, the transactional outbox that keeps a database write and an event emission consistent in Chapter 4, ordering guarantees in Chapter 6, and dead-letter handling for the messages that just won't process cleanly in Chapter 8. None of them are optional extras; they're the actual engineering an event-driven system requires in exchange for the coupling it removes.

WORTH REMEMBERING

Event-driven design doesn't make failure go away. It changes failure from "everything downstream breaks right now" to "some consumer is behind, and needs to catch up" — a much better failure to have, but only if you've actually built the machinery to recover from it.

1.3 When Not to Go Event-Driven

It's worth saying plainly, this early: not every system benefits from this. A small application with three services and a modest team can lose far more to the operational complexity of running a message broker, reasoning about eventual consistency, and debugging an asynchronous flow than it would ever lose to the coupling problems this chapter describes. Complexity you don't need yet is still complexity you're paying for.

The interactions that most need decoupling share a specific shape: the caller doesn't need an immediate answer, more than one part of the system cares about the same fact, or the downstream work is slow, unreliable, or independently scaled. Checkout confirming payment is none of those — it needs an answer now, from one specific place. Notifying five different systems that an order was placed is exactly that shape.

A reasonable default for a growing system: keep the request path — the calls a user is actually waiting on — synchronous and as simple as possible, and introduce events at the seams where one action needs to fan out to several independent, non-blocking consequences. That's usually a small fraction of your total call graph, and it's the fraction where this chapter's trade-offs pay for themselves fastest.

You don't need to decide your whole architecture on day one. You need to recognise the specific shape — one fact, several independent reactions, none of them blocking the thing that produced it — when it shows up in your system, and reach for the tools in the rest of this book precisely there, rather than everywhere at once.

With that scoped honestly, the next chapter gets concrete: what actually makes a good event, how to name and shape one so it survives your system changing around it, and the versioning discipline that keeps producers and consumers from drifting apart.

§ Chapter Summary

Eight things worth carrying into Chapter 2.

BEFORE YOU MOVE ON

01	A synchronous call chain couples services temporally, not just contractually — a slow dependency makes everything upstream slow too.
02	The tell for "this could be an event instead of a call": the caller doesn't use the response, it just needs the thing to happen.
03	Producers and consumers in an event system are independent — a struggling consumer doesn't propagate its problems backward.
04	That independence isn't free: you trade immediacy for decoupling, and take on duplicate delivery and ordering as new problems to solve.
05	Not every system needs this. Reach for events at the specific seams where one action needs several independent, non-blocking reactions.

BEFORE CHAPTER 2

Pick one synchronous call in a system you work on and ask: does the caller actually use the response, or does it just need the thing to happen? If it's the latter, sketch what the event would be called and who else might eventually want to consume it. Chapter 2 picks up exactly there — designing events that survive your system changing around them.

END OF SAMPLE

Eight chapters still to go.

Chapter 1 draws the line between calls that need an answer and events that just need to happen; the rest of the book covers designing those events properly, the delivery guarantees you actually get, the outbox pattern that keeps writes consistent, and running the whole thing in production without losing a message.