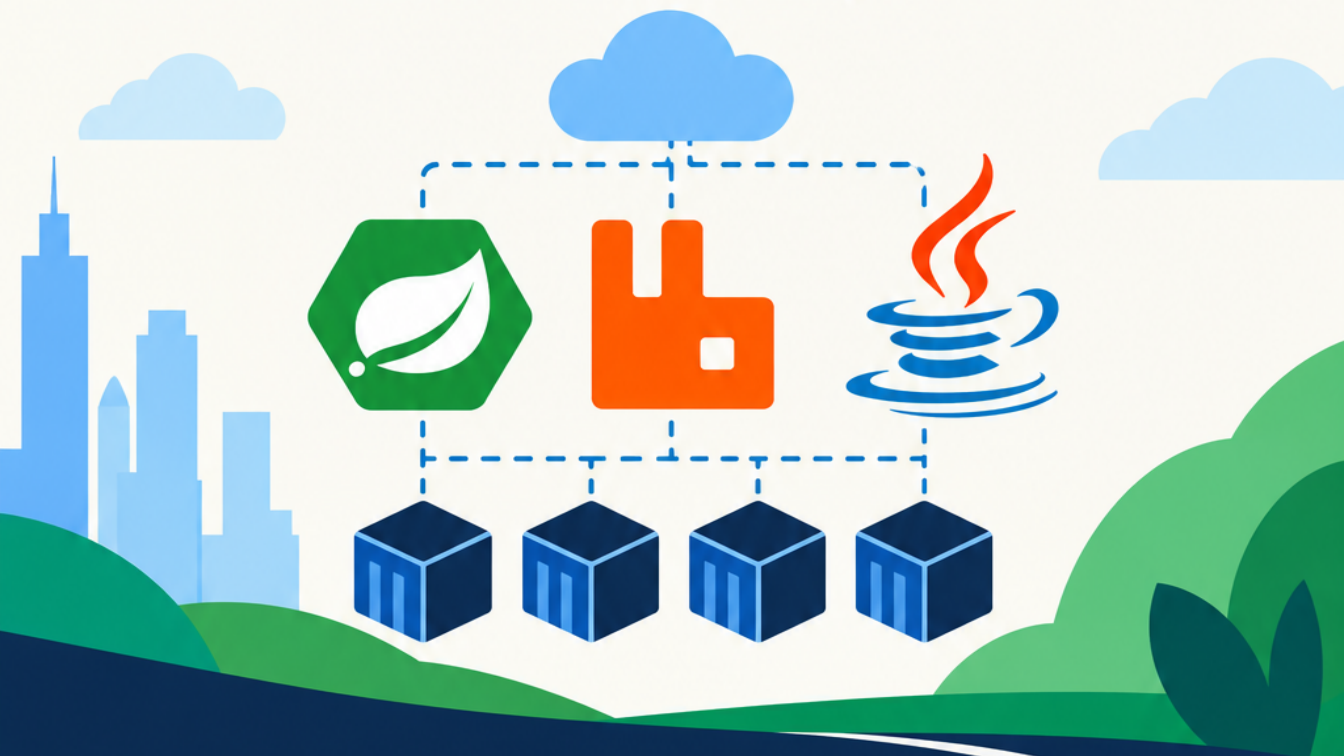


Spring, RabbitMQ, and Modern Java Platform Engineering

Designing, Building, and Operating
Reliable Distributed Systems



Florian Bauer

Spring, RabbitMQ, and Modern Java Platform Engineering

Designing, Building, and Operating Reliable Distributed Systems

Steve Publications

This book is available at

<https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>

This version was published on 2026-09-27



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

Designing, Building, and Operating Reliable Distributed Systems	1
Introduction	2
What This Book Covers	2
How This Book Is Structured	4
Who Should Read This Book	4
The Philosophy Behind This Book	5
Chapter 1: The Reality of Distributed Systems	6
The Moment Everything Changes	6
The Fundamental Tensions: Consistency, Availability, and Partition Tolerance	6
Latency, Concurrency, and Partial Failure	7
Why “Just Add a Message Broker” Is Not an Answer	8
Communication Models: RPC, Messaging, Events, and Their Trade-offs	9
The Operational Dimension: What Production Really Demands	10
Introducing OrderFlow: Our Reference Platform	11
Chapter 2: Modern Java for Platform Engineering	13
Why Java Matters for Distributed Systems	13
Java 21 Features That Matter: Records, Sealed Classes, Pattern Match- ing, Virtual Threads	13
JVM Fundamentals for Production: Memory Model, Garbage Collec- tion, Tuning Basics	19
Build Tooling: Maven, Modularization, and Dependency Management	21
Concurrency in Java: Executors, Virtual Threads, Structured Concur- rency	23
Structured Logging and Observability-Ready Code	23
Defensive Programming: Null Safety, Immutable Data, and Failure Handling	24

Chapter 3: Architectural Principles and Patterns 26

 From Monolith to Distributed: When and Why to Break Apart 26

 Domain-Driven Design: Bounded Contexts, Ubiquitous Language, and Strategic Design 26

 Event-Driven Architecture: Core Concepts and Anti-Patterns 26

 Microservices and Service-Oriented Architecture: Truths and Misconceptions 26

 Synchronous vs Asynchronous: Choosing the Right Communication Style 26

 Designing for Failure: Resilience Patterns and Graceful Degradation . . 27

Chapter 4: Spring Boot for Production Systems 28

 How Spring Boot Works: Auto-Configuration, Starters, and the Application Context 28

 Dependency Injection and the Component Model 28

 Configuration Management: Profiles, Externalization, and Hierarchies . 28

 REST with Spring Web MVC and WebFlux 28

 Actuator: Health, Metrics, and Production Visibility 29

 Lifecycle Management and Graceful Shutdown 29

Chapter 5: Data Access and Persistence with Spring Data 30

 Spring Data JPA: Entities, Repositories, and Query Methods 30

 Transaction Management: Propagation, Isolation, and Distributed Considerations 30

 Database Migrations with Flyway 31

 Connection Pooling and Performance 31

 Choosing Persistence Technologies: Relational vs Document vs Specialized Stores 31

 Persistence Anti-Patterns and Pitfalls 31

Chapter 6: RabbitMQ Fundamentals 32

 RabbitMQ Architecture: Broker, Nodes, Clusters, and the Management Layer 32

 The AMQP Protocol: Concepts, Versioning, and Alternatives 32

 Connections, Channels, and Resource Management 32

 Queues, Exchanges, and Bindings: The Core Data Model 32

 Message Flow and Lifecycle 32

 RabbitMQ Management: UI, HTTP API, and CLI Tools 32

Chapter 7: Exchange Types, Routing, and Topology Design 34

CONTENTS

Direct Exchanges: Point-to-Point Routing	34
Fanout Exchanges: Broadcasting and Multicast	34
Topic Exchanges: Pattern-Based Routing and Hierarchies	34
Headers Exchanges and Use Cases	34
Designing Exchange and Queue Topologies for OrderFlow	34
Virtual Hosts, Namespaces, and Multi-Tenancy	34
Chapter 8: Message Delivery Semantics and Reliability	36
At-Least-Once, At-Most-Once, and Exactly-Once: What They Really Mean	36
Publisher Confirms: Ensuring Messages Reach the Broker	36
Consumer Acknowledgments: Manual vs Automatic, Negative Ac- knowledgments	36
Message Durability: Persistent Messages and Queue Durability	36
Transactional Publishing: AMQP Transactions and Trade-offs	36
Implementing Reliable Delivery in OrderFlow	37
Chapter 9: Advanced Messaging Patterns with RabbitMQ	38
Dead Letter Exchanges and Queues: Handling Failed Messages	38
Message Timeouts and TTL-Based Expiration	38
Delayed Message Processing with Plugins and Alternatives	38
Priority Queues and Work Prioritization	38
Queue Length Limits and Overflow Handling	38
Message Inspection and Redelivery Control	39
Chapter 10: RabbitMQ Clustering, High Availability, and Scaling	40
RabbitMQ Clustering: Architecture, Topologies, and Quorum	40
HA Queues and Quorum Queues: Choosing the Right Model	40
Scaling Out: Adding Nodes, Rebalancing, and Monitoring	40
Broker Upgrades and Maintenance Procedures	40
Disaster Recovery Strategies and Data Backup	40
Performance Tuning: Memory, Disk, and Erlang Considerations	41
Chapter 11: Spring AMQP Fundamentals	42
The Spring AMQP Abstraction: RabbitTemplate, ConnectionFactory, and Channel	42
Configuring RabbitMQ Connections in Spring Boot	42
Sending Messages: RabbitTemplate and Producer Configuration	42
Receiving Messages: RabbitListener and Consumer Configuration . .	42
Message Conversion and Serialization	42

Error Handling Fundamentals: DefaultErrorHandler and Beyond . . .	43
Chapter 12: Building Reliable Producers with Spring AMQP	44
Publisher Confirms and Returns in Spring AMQP	44
ConfirmsCallback and ReturnCallback: Handling Broker Feedback . .	44
Retry Semantics: SimpleMessageConverter and RetryTemplate Inte- gration	44
Correlation IDs and Message Tracing	44
Batching and Bulk Publishing	44
Producer Performance and Throughput Optimization	45
Chapter 13: Building Reliable Consumers with Spring AMQP	46
Acknowledgment Modes: AUTO, MANUAL, and BATCH	46
Concurrency Tuning: Prefetch, Consumer Threads, and Queue Parti- tioning	46
The DefaultMessageListenerContainer Lifecycle and Configuration . .	46
Consumer Error Handling: Retry, Backoff, and DLQ Routing	46
Idempotent Consumer Design	46
Graceful Consumer Shutdown and Message In-Flight Handling	47
Chapter 14: Message Serialization, Schema Evolution, and Compatibility 48	48
JSON Serialization with Jackson: Configuration and Customization . .	48
Avro and Schema Registry for Contract-Governed Messages	48
Protobuf and Performance-Sensitive Scenarios	48
Schema Evolution Strategies: Backward, Forward, and Full Compati- bility	48
Versioning Messages and Supporting Multiple Consumers	48
Message Schema Documentation and Governance	49
Chapter 15: Event-Driven Design and the Outbox Pattern	50
Domain Events and Integration Events: Defining the Distinction	50
The Transactional Outbox Pattern: Theory and Implementation	50
Implementing Outbox with Spring Data and Polling	50
CDC-Based Outbox: Debezium and Change Data Capture	50
Event Publishing in OrderFlow: From Database to RabbitMQ	50
Testing Event-Driven Components	51
Chapter 16: The Inbox Pattern and Idempotent Processing	52
Why Idempotency Matters: Duplicate Detection and Processing	52
The Inbox Pattern: Tracking Received Messages	52

CONTENTS

Implementing Idempotency Keys and State Stores	52
Outbox-Inbox: End-to-End Reliable Communication	52
Handling Partial Failures and Compensating Actions	52
Idempotency in OrderFlow Services	52
Chapter 17: Sagas and Distributed Transactions	54
Why Distributed Transactions Fail: Two-Phase Commit and Its Problems	54
Saga Pattern: Choreography vs Orchestration	54
Implementing Sagas with RabbitMQ and Spring Events	54
Compensating Actions and Rollback Semantics	54
Saga State Machines and Persistence	54
OrderFlow Saga: Implementing an Order Fulfillment Workflow	54
Chapter 18: Message Ordering and Sequence Guarantees	56
When Ordering Matters and When It Does Not	56
FIFO Queues and Single-Consumer Constraints	56
Partitioning by Key for Ordered Substreams	56
Sequence Numbers and Gap Detection	56
Implementing Order Guarantees in RabbitMQ	56
Ordering Trade-offs: Throughput vs Determinism	56
Chapter 19: Concurrency, Backpressure, and Flow Control	58
Understanding Throughput, Latency, and Saturation	58
Backpressure Mechanisms: Prefetch, Queue Limits, and Rate Limiting	58
Spring AMQP Flow Control and Consumer Throttling	58
Circuit Breakers and Bulkheads: Protecting Services	58
Implementing Flow Control in OrderFlow	58
Capacity Planning and Load Testing	58
Chapter 20: Security, Authentication, and Authorization	60
RabbitMQ Security: Users, VHosts, Permissions, and TLS	60
Spring Security with OAuth2 and JWT	60
Service-to-Service Authentication Patterns	60
Secrets Management: Vault, Kubernetes Secrets, and Environment Config	60
Encrypting Data in Transit and at Rest	60
Security Testing and Compliance	60
Chapter 21: Observability – Logging, Metrics, and Tracing	62

CONTENTS

Structured Logging with Micrometer and Logback	62
Metrics with Micrometer: Gauges, Counters, Timers, and Histograms	62
Distributed Tracing with Micrometer Tracing and Brave/Open-Telemetry	62
Correlating Logs, Metrics, and Traces	62
RabbitMQ Metrics and Broker Visibility	62
Implementing Observability in OrderFlow	63
Chapter 22: Containerization, Kubernetes, and Deployment	64
Docker Images for Spring Boot Applications	64
Docker Compose for Local Development and Integration Testing	64
Kubernetes Deployments, Services, and ConfigMaps	64
RabbitMQ on Kubernetes: Helm Charts and Operators	64
Ingress, Networking, and Service Mesh Considerations	64
Deployment Strategies: Rolling, Blue-Green, and Canary	64
Chapter 23: Testing Strategy for Distributed Systems	66
Unit Testing Spring and RabbitMQ Components	66
Integration Testing with Testcontainers: RabbitMQ, Databases, and More	66
Contract Testing with Spring Cloud Contract	66
End-to-End Testing in an Isolated Environment	66
Testing Failure Scenarios: Network Partition, Broker Down, Slow Consumers	66
Test Infrastructure and CI Pipelines	66
Chapter 24: CI/CD, Automation, and Release Engineering	68
CI Pipeline Design: Build, Test, and Quality Gates	68
Container Registry and Image Tagging Strategy	68
CD Strategies and Automated Deployments	68
Environment Promotion and Feature Flags	68
Rollback Procedures and Release Management	68
The OrderFlow CI/CD Pipeline	68
Chapter 25: Troubleshooting, Maintenance, and Long-Term Evolution	70
Common Failure Modes and Their Symptoms	70
RabbitMQ Troubleshooting: Connections, Queues, Memory, and Disk	70
Spring Boot Troubleshooting: Memory Leaks, Thread Pools, and Timeouts	70
Incident Response and Debugging Production Issues	70

Dependency Management and Version Upgrades	70
Technical Debt, Refactoring, and Architectural Evolution	71
Chapter 26: The Production OrderFlow Platform	72
Full Architecture: Services, Events, and Flows	72
Production Deployment: Complete Kubernetes and Infrastructure Configuration	72
Operational Runbook: Monitoring, Scaling, and Incident Response . .	72
Performance Characteristics and Capacity Profiles	72
Evolutionary Roadmap: How the Architecture Could Grow	72
Lessons Learned and Key Takeaways	73
Conclusion	74
References	75

Designing, Building, and Operating Reliable Distributed Systems

This book is a complete technical guide to building distributed systems with Spring Boot, RabbitMQ, and modern Java. It takes you from the fundamental architectural principles that govern reliable distributed applications through the internals of each technology, and into the practices required to operate production-grade platforms at scale. A realistic reference application – an order processing platform called OrderFlow – evolves throughout the book from a simple service into a fully event-driven, containerized, observable microservice architecture. Every chapter includes complete, production-quality code, infrastructure configurations, and detailed explanations of why each architectural choice was made, what trade-offs it introduces, and how it affects the behavior of the overall system.

Introduction

The moment your application stops being a single process running on a single machine, everything changes.

It changes quietly at first. A new database for a new feature. A second service to handle authentication. A background job that runs on a different server. You may not even notice you have become distributed until you do: until the logs stop telling a single coherent story, until latency spikes that have nothing to do with your code start appearing in monitoring dashboards, until a deployment succeeds but one service is now out of sync with another and you spend three hours tracing a missing message through half a dozen components.

This book exists to help you arrive at that moment prepared. Not with dogma or a prescriptive architecture to copy, but with a working understanding of the principles, mechanisms, and trade-offs that govern real distributed systems built with real technologies.

What This Book Covers

We use three core technologies as our foundation: Java 21 as the language, Spring Boot 3.3.x as the application platform, and RabbitMQ 4.x as the message broker. These are not arbitrary choices. Java remains the dominant language for enterprise back-end systems, Spring Boot is the most widely adopted framework for building production Java services, and RabbitMQ is one of the most mature and broadly deployed message brokers in the world. Together they form a platform that thousands of organizations already depend on for their most critical systems.

The book is organized into seven parts that build on each other progressively:

Part I establishes the foundational principles. We examine what makes distributed systems fundamentally different from monolithic applications: partial failure, network latency, consistency versus availability, and the operational realities that production demands. We then cover the modern Java features

that underpin reliable platform development: virtual threads, sealed classes, records, pattern matching, and JVM considerations for production. We conclude with a treatment of architectural patterns – domain-driven design, event-driven architecture, microservices, and the patterns that help us reason about failure.

Part II covers Spring Boot as the application foundation. You will learn how Spring Boot auto-configuration actually works under the hood, how dependency injection shapes your component model, how configuration management and profiles enable environment-specific behavior, and how Actuator gives you visibility into running applications. We examine Spring Data JPA for persistence, transaction management, and the database migration tooling that production systems need.

Part III dives deeply into RabbitMQ. You will understand the broker architecture, the AMQP 0-9-1 protocol model, connections and channels, exchanges and queues, and routing. We cover every exchange type and how to design topologies that scale. We examine message delivery semantics: publisher confirms, consumer acknowledgments, durability, and what “reliable delivery” actually means. We cover dead letter exchanges, delayed messaging, prioritization, clustering, high availability with quorum queues, and the operational procedures required to run RabbitMQ in production.

Part IV focuses on Spring AMQP, the integration layer between your Spring Boot applications and RabbitMQ. You will learn the programming model: RabbitTemplate, RabbitListener, message conversion, error handling. We cover production-grade producer configuration with publisher confirms and returns, and consumer configuration with manual acknowledgments, concurrency tuning, and fault handling. We examine serialization strategies and the critical problem of schema evolution.

Part V is about the distributed application patterns that make messaging powerful. We implement the transactional outbox pattern to ensure consistency between database operations and event publishing. We implement the inbox pattern for idempotent message processing. We cover the saga pattern for multi-service business transactions. We examine message ordering guarantees, backpressure, and flow control mechanisms that prevent cascading failures.

Part VI covers production operations and platform engineering. Security with TLS and OAuth2, observability with metrics and distributed tracing,

containerization and Kubernetes deployment, comprehensive testing strategies with Testcontainers, CI/CD pipelines, and the day-to-day work of troubleshooting and maintaining these systems.

Part VII synthesizes everything into a complete reference architecture. We review the fully evolved OrderFlow platform, walk through the production deployment configuration, and provide an operational runbook for monitoring, scaling, and incident response.

How This Book Is Structured

Each chapter is self-contained enough that you can read it for a specific topic, but the chapters build on each other in sequence. Reading them in order gives you the full progression from principles to practice.

We use a single reference application throughout: OrderFlow, an e-commerce order processing platform. Each chapter adds to this application incrementally. By the end of the book you will have a complete, working distributed platform with multiple services, event-driven communication, reliable messaging, observability, containerized deployment, and production-grade operational tooling.

All code examples are complete and intended to compile and run. Where configuration files, Dockerfiles, or Kubernetes manifests are included, they are full artifacts, not fragments. Dependency versions are explicitly stated so you can reproduce the examples exactly.

Who Should Read This Book

This book is for Java developers and platform engineers who build or operate distributed systems in production. You should be comfortable with Java and understand basic concepts like HTTP, databases, and REST APIs. You do not need prior experience with Spring Boot, RabbitMQ, or microservices architecture, though familiarity with any of these will help you move faster through the early chapters.

If you are already building Spring Boot applications and considering adding RabbitMQ for asynchronous communication, this book will help you do it correctly from the start. If you are already using RabbitMQ but want to

understand its internals, optimize your configuration, or solve specific production problems like message ordering, reliable delivery, or cluster scaling, the RabbitMQ-specific chapters will give you that depth. If you are responsible for operating these systems in production, the chapters on observability, security, testing, Kubernetes, and troubleshooting will serve as a practical reference.

The Philosophy Behind This Book

There is a tension in technical books between being comprehensive and being readable. I have tried to honor both. Every claim here is grounded in either primary documentation or established engineering practice. Every code example has been designed to work with current versions of the technologies it uses. Every architectural recommendation comes with a discussion of trade-offs, not just the ideal scenario.

Technology choices are rarely right or wrong in absolute terms. They are appropriate or inappropriate for a specific context, with specific constraints, specific constraints that include team skills, operational maturity, regulatory requirements, and budget. I will take positions on trade-offs when the evidence is clear, but I will also show you how to think through those trade-offs yourself so you can make informed decisions in your own context.

The hardest problems in distributed systems are not about choosing the right technology. They are about understanding what the technology guarantees, what it does not guarantee, and how to design your application behavior around those constraints. This book focuses on that understanding. Let us get started.

Chapter 1: The Reality of Distributed Systems

The Moment Everything Changes

A single process on a single machine follows the rules you learned in introductory computer science. Memory is shared. Time is sequential. Failure is rare and catastrophic. When you write code, you can reason about what will happen next with confidence.

Add a second machine and the rules disappear.

The second machine may be slower. It may be in a different data center. It may have a different clock. The network between them may delay messages, drop messages, duplicate messages, or reorder them. One machine may crash silently while the other continues running. Neither machine can know, in real time, what state the other machine is in.

These are not edge cases. They are the default behavior of any system that spans more than one machine. Building reliable distributed systems means designing for these conditions as first-class concerns, not as problems to handle with defensive code in a few critical paths.

The Fundamental Tensions: Consistency, Availability, and Partition Tolerance

The CAP theorem is the most widely cited result in distributed systems literature, and also the most widely misunderstood. It states that in the presence of a network partition, a distributed system can provide either consistency or availability, but not both.

Breaking this down carefully is important because the sloppy versions of this theorem lead to bad architectural decisions.

Consistency, in the CAP context, means that every read receives the most recent write or an error. This is what you get from a single-process application with a single database: if you write a value and then read it, you see what you wrote.

Availability means that every request receives a response, without guarantee that the response contains the most recent write. The system never says “I am unavailable.” It might return stale data, but it always responds.

Partition tolerance means the system continues to operate despite arbitrary message loss or delay between nodes. The network can split the cluster into two groups that cannot communicate with each other, and the system keeps running.

The key insight is that partition tolerance is not a choice. In any distributed system spanning multiple machines, the network can and will fail. You cannot design a system that assumes the network will always work. You must tolerate partitions. So the real choice is between consistency and availability during a partition.

In practice, most production systems choose availability during partitions. When the partition heals, they use reconciliation mechanisms to restore consistency. This is the approach used by most distributed databases and message brokers, including RabbitMQ. During a network split, different partitions may accept writes independently. When connectivity is restored, a conflict resolution or reconciliation process ensures the system converges to a consistent state.

Understanding this tension matters because it affects how you design your application logic. If you build on top of a system that prioritizes availability during partitions, your application must be prepared to handle stale reads and eventual consistency. If you choose consistency, your application may experience periods where it cannot process requests at all. There is no free lunch.

Latency, Concurrency, and Partial Failure

Three characteristics define the runtime environment of distributed systems.

Latency: Communication between machines takes time, and that time varies. A call to a service in the same availability zone might take one

millisecond. A call to a service in a different region might take two hundred. A network congestion event might make either call take several seconds. When you make a synchronous call to another service, your thread is blocked for the duration of that call. If you have fifty concurrent requests each waiting for a five-hundred-millisecond call, you need enough thread capacity to handle all fifty waiting threads plus any other work in the system. This simple arithmetic is what drives most concurrency design decisions.

Concurrency: Multiple operations happen simultaneously. Two users place orders for the last item in stock at nearly the same time. Two services update the same record in parallel. The order in which operations complete may differ from the order in which they were initiated. You need mechanisms to coordinate concurrent access: locks, transactions, optimistic concurrency control, or queueing. Each mechanism has trade-offs in terms of performance, complexity, and failure behavior.

Partial failure: In a single-machine system, failure is usually binary. The process either runs or it does not. In a distributed system, failure is granular. The database may be up but slow. One service may be up while another is restarting. The network between two services may be working while the network to a third is degraded. A message broker may accept publishes but fail to deliver to a specific queue. Your system must handle these partial conditions gracefully.

The Fallacies of Distributed Computing, first published in 1994 by Peter Deutsch at Sun Microsystems, remain startlingly relevant. The network is not reliable. Latency is not zero. Bandwidth is not infinite. The network is not secure. The topology does not change. There is one administrator. Transport cost is zero. The network is homogeneous. None of these statements are true in production. Modern systems are more complex than they were in 1994, making these fallacies even more dangerous.

Why “Just Add a Message Broker” Is Not an Answer

When organizations encounter the performance and coupling problems of synchronous, direct service-to-service calls, a common instinct is to add a message broker. “We will decouple everything with messages,” the architecture diagram proclaims, and services are connected by arrows labeled with queue names.

This instinct addresses real problems but introduces new ones if the underlying concepts are not understood.

A message broker is not magic. It is another distributed component with its own failure modes. It can become slow. It can run out of memory. It can partition into multiple independent clusters. Messages can be lost if durability is not configured. Messages can be delivered multiple times if acknowledgments fail. Consumers can fall behind if they process too slowly. The broker does not eliminate distributed systems complexity. It moves some of it from application code into infrastructure, and adds infrastructure-specific complexity on top.

The decision to use a message broker should be driven by specific requirements:

Asynchronous processing when the caller does not need an immediate response. Load leveling when production bursts exceed steady-state processing capacity. Event distribution when multiple independent consumers need to react to the same occurrence. Decoupling when services should not know each other's existence or interfaces.

If none of these apply, adding a message broker increases complexity without benefit. A synchronous HTTP call between services in the same data center may be simpler, faster, and easier to debug than routing the same request through a queue.

Communication Models: RPC, Messaging, Events, and Their Trade-offs

There are three fundamental communication models in distributed systems. Understanding the differences is essential for choosing the right pattern for each situation.

Remote Procedure Call: In the RPC model, one service directly invokes a method or endpoint on another service, waits for a response, and continues. HTTP REST APIs are the most common form. RPC is synchronous and request-response oriented. It is simple to reason about: you send a request, you get a response. The caller and callee are tightly coupled: the caller must know the callee's interface, be aware of its availability, and handle its errors. Latency is additive: if service A calls service B which calls service C, the total latency is the sum of all three calls plus network overhead.

Messaging: In the messaging model, one service sends a message to a queue or topic, and another service consumes it asynchronously. The sender does not wait for a response and may not know which service will process the message. Messaging is inherently asynchronous. It decouples sender and receiver in time: the message can be processed later, even if the receiver is temporarily unavailable. It also decouples them in space: the sender does not need to know the receiver's address. However, messaging introduces complexity: you must handle duplicates, failures, ordering, and the monitoring overhead of an additional component.

Events: In the event model, services publish events describing things that have happened, and any interested services can subscribe to those events. Events are a specific form of messaging where the semantic is “something occurred” rather than “please do something.” Events are typically published once and consumed by multiple subscribers independently. The event publisher does not care who subscribes or how they react. This enables extreme decoupling and composability: you can add new subscribers without changing the publisher. The trade-off is that reasoning about the overall system behavior becomes harder because you cannot trace a single request through the system. Instead you must understand the event flow: which events are published, in what order, and what side effects each subscriber performs.

Most production systems use a combination of all three. A user submits an order through a REST API call. The order service saves it and publishes an event. The payment service, the inventory service, and the notification service each consume that event and perform their own processing. If any step fails, compensating actions may be triggered. Understanding when to use each model is one of the most important skills in distributed system design.

The Operational Dimension: What Production Really Demands

A system that works in development but is impossible to operate in production is not a successful system. The operational requirements often dictate architectural decisions more than the functional requirements do.

Observability: You must be able to see what is happening inside your running system. This means structured logs that can be searched and correlated, metrics that expose key behavioral dimensions, and distributed traces that

follow a request across service boundaries. If you cannot answer “what is slow right now” or “why did this request fail” in minutes rather than hours, your system is not observable enough.

Reliability: The system must continue operating despite component failures. A single database crash should not take down the entire platform. A slow consumer should not block all message processing. A deployment that introduces a bug should be reversible quickly. Reliability is achieved through redundancy, isolation, graceful degradation, and automated recovery mechanisms.

Scalability: The system must handle growth. This includes not just more users and more data, but also more services and more operational complexity. A scalable architecture makes it possible to add new features and new teams without requiring a complete rewrite of existing systems.

Maintainability: The system must be understandable and modifiable over time. Code is written once but read many times. Configuration that is clever but opaque will bite you at 3 AM when a production incident requires a quick fix. Documentation that is out of date is worse than no documentation. Maintainability means choosing clear over clever, standardized over exotic, and explicit over implicit.

Security: The system must protect data and access. This includes authentication and authorization for all interfaces, encryption for data in transit and at rest, proper secrets management, and defense-in-depth against a range of attack vectors. Security is not a feature you add at the end. It is a constraint that shapes design decisions from the start.

These operational demands are not abstract concerns. They are the difference between a system that earns your team’s trust and one that becomes a source of constant firefighting. Every architectural choice we make in the chapters ahead will be evaluated against these criteria.

Introducing OrderFlow: Our Reference Platform

Throughout this book we will develop a reference application called OrderFlow, an e-commerce order processing platform. It is designed to be realistic without being unnecessarily complex. It will start as a simple application and evolve into a fully distributed, event-driven, production-grade platform.

The OrderFlow domain is deliberately chosen because it maps naturally to distributed systems patterns. An order flows through multiple stages: creation, validation, inventory check, payment, fulfillment, and notification. Each stage can be a separate bounded context managed by an independent service. Failures in one stage require specific compensating actions. Multiple services need to react to the same events. The latency and reliability requirements for different stages vary. These characteristics make OrderFlow a useful vehicle for demonstrating real-world trade-offs.

The platform will consist of the following services:

Order Service: Receives order requests, validates them, manages order lifecycle state, and publishes domain events. Payment Service: Processes payments, handles payment failures, and publishes payment status events. Inventory Service: Checks and reserves stock, manages inventory levels, and publishes inventory events. Notification Service: Sends email, SMS, and in-app notifications based on order and payment events. Analytics Service: Aggregates events for reporting, metrics, and business intelligence.

These services communicate through both synchronous REST calls and asynchronous RabbitMQ messages. They use PostgreSQL for persistent storage. They are containerized with Docker and deployed to Kubernetes. They are instrumented with observability tooling. They are tested with a comprehensive automated test suite.

By the end of the book, you will have the complete codebase for OrderFlow, the infrastructure configuration to deploy it, and the understanding to operate it, modify it, and evolve it for your own needs.

We begin with the language and platform foundations that make all of this possible.

Chapter 2: Modern Java for Platform Engineering

Why Java Matters for Distributed Systems

Java remains the dominant language for enterprise back-end systems, and its longevity is not an accident. The ecosystem is vast, the performance is well understood, the tooling is mature, and the language evolves steadily without breaking existing investments. For platform engineering specifically, Java offers strong typing that catches errors at compile time, a garbage collector that handles memory management, a concurrency model that supports massive parallelism, and a runtime that can be tuned and monitored with fine granularity.

Java 21, the current Long Term Support release, introduces features that directly address the needs of modern distributed systems: virtual threads for scalable I/O-bound concurrency, sealed classes and pattern matching for type-safe event and message handling, records for concise data modeling, and structured concurrency for reasoning about concurrent execution. This chapter covers these features and the JVM fundamentals that underpin reliable production systems.

Java 21 Features That Matter: Records, Sealed Classes, Pattern Matching, Virtual Threads

Records

Records were introduced in Java 14 as a preview feature and became standard in Java 16. They solve a simple but pervasive problem: modeling immutable data carriers without boilerplate.

Before records, a typical data class looked like this:

```

1  public class OrderRequest {
2      private final String customerId;
3      private final String productId;
4      private final int quantity;
5      private final String shippingAddress;
6
7      public OrderRequest(String customerId, String productId, int quantity,
8          ↪ String shippingAddress) {
9          this.customerId = customerId;
10         this.productId = productId;
11         this.quantity = quantity;
12         this.shippingAddress = shippingAddress;
13     }
14
15     public String getCustomerId() { return customerId; }
16     public String getProductId() { return productId; }
17     public int getQuantity() { return quantity; }
18     public String getShippingAddress() { return shippingAddress; }
19
20     @Override
21     public boolean equals(Object o) {
22         if (this == o) return true;
23         if (!(o instanceof OrderRequest that)) return false;
24         return quantity == that.quantity
25             && java.util.Objects.equals(customerId, that.customerId)
26             && java.util.Objects.equals(productId, that.productId)
27             && java.util.Objects.equals(shippingAddress, that.shippingAddress);
28     }
29
30     @Override
31     public int hashCode() {
32         return java.util.Objects.hash(customerId, productId, quantity,
33             ↪ shippingAddress);
34     }
35
36     @Override
37     public String toString() {
38         return "OrderRequest{customerId='" + customerId + "', productId='" +
39             ↪ productId
40             + "', quantity=" + quantity + ", shippingAddress='" +
41             ↪ shippingAddress + "'}";
42     }
43 }

```

With records, the same class is:

```
1 public record OrderRequest(  
2     String customerId,  
3     String productId,  
4     int quantity,  
5     String shippingAddress  
6 ) {}
```

The compiler generates the constructor, accessors, equals, hashCode, and toString automatically. Records are implicitly final, so they cannot be subclassed. Components are implicitly private and final. The accessor methods are named after the component: `customerId()` instead of `getCustomerId()`.

For distributed systems, records are ideal for modeling events, commands, and DTOs. Events in an event-driven system are naturally immutable: an `OrderCreated` event that occurred in the past cannot change its facts. Using records makes this immutability explicit and eliminates a whole class of bugs from accidental mutation.

Records integrate seamlessly with Jackson for JSON serialization. A record serializes to a JSON object with field names matching component names. Deserialization works through the canonical constructor. For our `OrderFlow` application, we will use records extensively for event and message types.

Sealed Classes

Sealed classes, finalized in Java 17, allow you to restrict which classes or interfaces may extend or implement a given class or interface. This enables exhaustive pattern matching and makes your type hierarchy explicit and self-documenting.

In an event-driven system, you often have hierarchies of event types:

```
1  public sealed interface OrderEvent permits
2      OrderCreated,
3      OrderValidated,
4      OrderPaymentAuthorized,
5      OrderPaymentCaptured,
6      OrderShipped,
7      OrderDelivered,
8      OrderCancelled
9  {}
10
11  public record OrderCreated(String orderId, String customerId, String productId,
12      ↪ int quantity)
13      implements OrderEvent {}
14
15  public record OrderCancelled(String orderId, String reason)
16      implements OrderEvent {}
17
18  public record OrderValidated(String orderId)
19      implements OrderEvent {}
20
21  public record OrderPaymentAuthorized(String orderId)
22      implements OrderEvent {}
23
24  public record OrderPaymentCaptured(String orderId)
25      implements OrderEvent {}
26
27  public record OrderShipped(String orderId)
28      implements OrderEvent {}
29
30  public record OrderDelivered(String orderId)
31      implements OrderEvent {}
```

The permits clause explicitly lists all allowed subtypes. No other class can implement `OrderEvent`. This makes the hierarchy closed and predictable. Combined with pattern matching in switch statements, the compiler can verify that all cases are handled.

Sealed classes are particularly valuable for event handling, error handling, and state modeling. They replace the fragile instanceof chains and Visitor patterns that clutter older Java codebases.

Pattern Matching for Switch

Pattern matching for switch, finalized in Java 21, allows you to match on types and destructure records directly in switch statements. Combined with sealed classes, it enables exhaustive, compile-time-checked pattern matching.

Consider processing different types of `OrderEvent`:

```

1  public void handleEvent(OrderEvent event) {
2      switch (event) {
3          case OrderCreated created -> {
4              log.info("Order created: {}", created.orderId());
5              reserveInventory(created);
6          }
7          case OrderPaymentAuthorized authorized -> {
8              log.info("Payment authorized for order: {}", authorized.orderId());
9          }
10         case OrderCancelled cancelled -> {
11             log.info("Order cancelled: {} reason: {}", cancelled.orderId(),
12                 ↪ cancelled.reason());
13             releaseInventory(cancelled.orderId());
14         }
15         case OrderValidated validated -> {
16             log.info("Order validated: {}", validated.orderId());
17         }
18         case OrderPaymentCaptured captured -> {
19             log.info("Payment captured for order: {}", captured.orderId());
20         }
21         case OrderShipped shipped -> {
22             log.info("Order shipped: {}", shipped.orderId());
23         }
24         case OrderDelivered delivered -> {
25             log.info("Order delivered: {}", delivered.orderId());
26         }
27         // No default case needed: OrderEvent is sealed and all permitted
28         // subtypes are handled above, so the switch is exhaustive.
29         // (Since Java 21, switches over sealed types must be exhaustive
30         // even in statement form.)
31     }
32 }

```

Because `OrderEvent` is sealed and all permitted subtypes are listed in the switch, the compiler ensures the match is exhaustive. If you add a new event type to the sealed interface, the compiler will flag every switch expression that does not handle it. This prevents subtle bugs where a new event type is silently ignored.

Pattern matching also supports destructuring records directly:

```
1 public double calculateOrderValue(OrderEvent event) {
2     return switch (event) {
3         case OrderCreated(String orderId, String customerId, String productId,
4             ↪ int quantity)
5             when quantity > 0 -> quantity * getUnitPrice();
6         case OrderCancelled(String orderId, String reason) -> 0.0;
7         case OrderEvent other -> throw new IllegalStateException(
8             "Cannot calculate value for: " + other.getClass());
9     };
10 }
```

This pattern is essential for building robust event processors in distributed systems. It ensures that event handling logic is complete, type-safe, and evolves together with the event type hierarchy.

Virtual Threads

Virtual threads, introduced as a standard feature in Java 21 via JEP 444, are perhaps the most significant concurrency improvement in Java's history.

Traditional Java threads map one-to-one with operating system threads. Each thread consumes a stack of typically 1 MB. An application with 500 concurrent threads uses 500 MB of stack space alone. This limit constrains throughput for I/O-bound applications where threads spend most of their time waiting for external systems: databases, HTTP calls, message brokers.

Virtual threads are different. They are lightweight threads managed by the JVM, not the OS. Thousands of virtual threads can run on a small number of OS threads, called carrier threads. When a virtual thread blocks on I/O, the JVM parks it and schedules another virtual thread on the same carrier thread. When the I/O completes, the JVM resumes the original virtual thread. This model is often described as “many-to-many” scheduling.

The key insight is that virtual threads make the thread-per-task model practical again. Instead of designing complex asynchronous or reactive pipelines with callbacks and `CompletableFuture` chains, you can write straightforward blocking code that scales to thousands of concurrent operations.

Creating a virtual thread is simple:

```
1 var executor = Executors.newVirtualThreadPerTaskExecutor();  
2 executor.submit(() -> processOrder(order));
```

Or using the static factory:

```
1 Thread.startVirtualThread(() -> processOrder(order));
```

For Spring Boot applications, enabling virtual threads is straightforward. Setting the following property causes the Tomcat servlet container to use virtual threads:

```
1 spring.threads.virtual=true
```

This single configuration change can dramatically improve throughput for I/O-bound applications. Benchmarks show throughput improvements of 10x to 50x for applications that spend significant time waiting for database queries or external HTTP calls, without changing application code.

There are important caveats. Virtual threads are designed for I/O-bound tasks, not CPU-bound tasks. A CPU-bound virtual thread occupies its carrier thread continuously, providing no benefit over a platform thread. You should not use virtual threads for tight computation loops.

Virtual threads should not be used with synchronized blocks that hold locks for long durations. Because virtual threads are scheduled on carrier threads, a long-running synchronized block in a virtual thread can block the carrier thread and reduce throughput for all virtual threads sharing that carrier. For synchronization, prefer lock-free data structures or concurrent utilities designed for high concurrency.

For our OrderFlow application, we will use virtual threads in services that make blocking calls to databases and other services. This gives us the simplicity of blocking code with the scalability previously only achievable with reactive frameworks.

JVM Fundamentals for Production: Memory Model, Garbage Collection, Tuning Basics

Understanding the JVM is not optional for production Java engineering. The JVM is the environment in which your application runs, and its behavior directly

affects latency, throughput, and reliability.

Memory Model

The JVM heap is divided into regions for different generations of objects. Young generation holds newly allocated objects. Most objects die young, so the young generation is collected frequently with low cost. Old generation holds long-lived objects that survive multiple young generation collections. Metaspace holds class metadata and is not part of the heap.

Key properties for production:

-Xmx: Maximum heap size. In containers, the JVM 8u131+ automatically detects container memory limits and sets the maximum heap proportionally. For explicit control, use `-XX:MaxRAMPercentage=75.0` to set heap to 75% of available container memory. -Xms: Initial heap size. In production, set this equal to -Xmx to avoid the performance cost of heap resizing. -XX:MaxMetaspaceSize: Limit on metaspace. Defaults to unlimited, which is dangerous in long-running applications. Set an explicit limit based on observed usage, typically 256–512 MB for most services.

Garbage Collection

Garbage collection pauses affect latency. A GC pause stops all application threads while the collector reclaims memory. In systems with strict latency requirements, GC pauses must be minimized.

G1 Garbage Collector is the default in Java 11–21 and is suitable for most production workloads. It targets a maximum pause time and divides the heap into regions. Configuration:

- 1 `-XX:+UseG1GC`
- 2 `-XX:MaxGCPauseMillis=200`

The Z Garbage Collector, available as a standard option in Java 21, provides sub-millisecond pauses even for very large heaps (up to terabytes). It uses concurrent marking and compaction with a sophisticated lock-free design. Configuration:

```
1 -XX:+UseZGC
```

ZGC is increasingly used in latency-sensitive applications. However, it can use slightly more CPU and memory than G1. The trade-off is worth it when pause time is more critical than raw throughput.

For our OrderFlow services, we will use G1GC for most services and consider ZGC for latency-critical paths. The choice depends on observed behavior under load.

Thread and Connection Management

The number of threads affects how your application handles concurrent requests. In a traditional thread pool model, if the pool is exhausted, new requests queue and may time out. Virtual threads remove this constraint for I/O-bound tasks.

Connection pools are critical for database and message broker access. Each connection is a finite resource on both the client and server side. For database connections, HikariCP is the recommended pool. Key settings:

```
1 spring.datasource.hikari.maximum-pool-size=20
2 spring.datasource.hikari.minimum-idle=5
3 spring.datasource.hikari.connection-timeout=5000
4 spring.datasource.hikari.idle-timeout=600000
5 spring.datasource.hikari.max-lifetime=1800000
```

For RabbitMQ connections, Spring AMQP uses a `CachingConnectionFactory` that reuses connections and channels. A single connection with multiple channels is the standard pattern.

Build Tooling: Maven, Modularization, and Dependency Management

Maven is the most widely used build tool for Spring Boot applications. Understanding its dependency resolution and project structure is essential for managing complex multi-service architectures.

For OrderFlow, we use a multi-module Maven project. The parent POM manages dependency versions, and each service is a child module. This ensures consistency across services and simplifies upgrades.

The parent pom.xml defines the Spring Boot parent, the dependency management section, and build plugins:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5          http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7
8      <parent>
9          <groupId>org.springframework.boot</groupId>
10         <artifactId>spring-boot-starter-parent</artifactId>
11         <version>3.3.0</version>
12         <relativePath/>
13     </parent>
14
15     <groupId>com.orderflow</groupId>
16     <artifactId>orderflow-platform</artifactId>
17     <version>1.0.0-SNAPSHOT</version>
18     <packaging>pom</packaging>
19     <name>OrderFlow Platform</name>
20
21     <modules>
22         <module>order-service</module>
23         <module>payment-service</module>
24         <module>inventory-service</module>
25         <module>notification-service</module>
26         <module>analytics-service</module>
27     </modules>
28
29     <properties>
30         <java.version>21</java.version>
31         <spring-amqp.version>3.1.0</spring-amqp.version>
32         <postgresql.version>42.7.3</postgresql.version>
33         <flyway.version>10.12.0</flyway.version>
34     </properties>
35
36     <dependencyManagement>
37         <dependencies>
38             <!-- Dependencies with explicit versions managed here -->
39         </dependencies>
40     </dependencyManagement>
41
42     <build>

```

```
43     <plugins>
44         <plugin>
45             <groupId>org.apache.maven.plugins</groupId>
46             <artifactId>maven-compiler-plugin</artifactId>
47             <configuration>
48                 <source>21</source>
49                 <target>21</target>
50             </configuration>
51         </plugin>
52     </plugins>
53 </build>
54 </project>
```

Each service module declares its dependencies without versions, relying on the parent's managed versions. This pattern scales to many services and prevents version drift.

Concurrency in Java: Executors, Virtual Threads, Structured Concurrency

Concurrency is unavoidable in distributed systems. Services handle multiple requests simultaneously. Background tasks process events asynchronously. External calls are made in parallel. Java provides several concurrency mechanisms, each appropriate for different scenarios.

For I/O-bound parallel calls, virtual threads are the default choice in Java 21. For CPU-bound tasks, a fixed thread pool sized to the number of available processors is appropriate:

```
1  var cpuExecutor =
    ↳ Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```

Structured concurrency, available as a preview in Java 21 and later versions, provides a framework for managing related concurrent tasks as a single unit. It ensures that all tasks complete or are cancelled together, simplifying error handling and resource cleanup. While still evolving, structured concurrency addresses one of the hardest problems in concurrent programming: keeping related tasks coordinated.

For our OrderFlow application, we will use virtual threads for request processing and fixed pools for CPU-bound batch operations.

Structured Logging and Observability-Ready Code

Logging is the primary tool for understanding what happened in a distributed system. Structured logging, where log entries are formatted as machine-readable key-value pairs rather than free-form text, makes logs searchable, aggregatable, and analyzable.

Java logging frameworks like SLF4J with Logback support structured logging through MDC (Mapped Diagnostic Context). Trace IDs, span IDs, and request IDs are stored in the MDC and automatically included in every log entry:

```
1 MDC.put("traceId", traceId);
2 try {
3     log.info("Processing order", Map.of("orderId", orderId));
4 } finally {
5     MDC.clear();
6 }
```

With Spring Boot 3.3 and Micrometer Tracing, trace IDs and span IDs are automatically injected into the MDC when tracing is enabled. You do not need to manually manage context propagation.

Log levels must be used consistently:

ERROR: Something failed and needs immediate attention. Action is required. WARN: Something unexpected happened but the system continues operating. Investigation may be needed. INFO: Normal operational events. Request processing, state changes, lifecycle events. DEBUG: Detailed diagnostic information for development and troubleshooting.

Every service should log at the INFO level what it received, what it did, and what it produced. This makes it possible to trace the flow of work through the system from logs alone.

Defensive Programming: Null Safety, Immutable Data, and Failure Handling

Distributed systems amplify the cost of bugs. A null pointer exception in a single service can cascade into timeouts and failures across multiple services. Defensive programming practices reduce these risks.

Null safety: Use `Optional` for values that may be absent. Use `@NonNull` annotations from Jakarta Validation or tools like SpotBugs or Error Prone to catch null-related bugs at compile time. Prefer records with non-nullable components for data that should always have values.

Immutable data: Immutable objects are inherently thread-safe. They cannot be modified after creation, so they can be shared freely between threads without synchronization. Events, commands, and DTOs should be immutable.

Failure handling: In distributed systems, failures are normal. Design your code to handle them gracefully. Use specific exception types rather than generic `RuntimeException`. Implement retry logic for transient failures. Time out external calls to prevent indefinite blocking. Log failures with enough context to diagnose the root cause.

These practices form the foundation for reliable Java services. The next chapter builds on them to examine the architectural patterns that organize these services into coherent distributed systems.

Chapter 3: Architectural Principles and Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

From Monolith to Distributed: When and Why to Break Apart

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Domain-Driven Design: Bounded Contexts, Ubiquitous Language, and Strategic Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Event-Driven Architecture: Core Concepts and Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Microservices and Service-Oriented Architecture: Truths and Misconceptions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Synchronous vs Asynchronous: Choosing the Right Communication Style

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Designing for Failure: Resilience Patterns and Graceful Degradation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 4: Spring Boot for Production Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

How Spring Boot Works: Auto-Configuration, Starters, and the Application Context

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Building an OrderFlow Service

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Dependency Injection and the Component Model

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Configuration Management: Profiles, Externalization, and Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

REST with Spring Web MVC and WebFlux

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Actuator: Health, Metrics, and Production Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Lifecycle Management and Graceful Shutdown

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 5: Data Access and Persistence with Spring Data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Spring Data JPA: Entities, Repositories, and Query Methods

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The N+1 Problem and Solutions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Transaction Management: Propagation, Isolation, and Distributed Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Propagation Behaviors

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Isolation Levels

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Transactional Boundaries in Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Database Migrations with Flyway

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Connection Pooling and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Choosing Persistence Technologies: Relational vs Document vs Specialized Stores

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Persistence Anti-Patterns and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 6: RabbitMQ Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ Architecture: Broker, Nodes, Clusters, and the Management Layer

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The AMQP Protocol: Concepts, Versioning, and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Connections, Channels, and Resource Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Queues, Exchanges, and Bindings: The Core Data Model

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Message Flow and Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ Management: UI, HTTP API, and CLI Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 7: Exchange Types, Routing, and Topology Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Direct Exchanges: Point-to-Point Routing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Fanout Exchanges: Broadcasting and Multicast

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Topic Exchanges: Pattern-Based Routing and Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Headers Exchanges and Use Cases

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Designing Exchange and Queue Topologies for OrderFlow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Virtual Hosts, Namespaces, and Multi-Tenancy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 8: Message Delivery

Semantics and Reliability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

At-Least-Once, At-Most-Once, and Exactly-Once: What They Really Mean

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Publisher Confirms: Ensuring Messages Reach the Broker

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Consumer Acknowledgments: Manual vs Automatic, Negative Acknowledgments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Message Durability: Persistent Messages and Queue Durability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Transactional Publishing: AMQP Transactions and Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Reliable Delivery in OrderFlow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 9: Advanced Messaging Patterns with RabbitMQ

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Dead Letter Exchanges and Queues: Handling Failed Messages

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Message Timeouts and TTL-Based Expiration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Delayed Message Processing with Plugins and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Priority Queues and Work Prioritization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Queue Length Limits and Overflow Handling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Message Inspection and Redelivery Control

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 10: RabbitMQ Clustering, High Availability, and Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ Clustering: Architecture, Topologies, and Quorum

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

HA Queues and Quorum Queues: Choosing the Right Model

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Scaling Out: Adding Nodes, Rebalancing, and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Broker Upgrades and Maintenance Procedures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Disaster Recovery Strategies and Data Backup

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Performance Tuning: Memory, Disk, and Erlang Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 11: Spring AMQP

Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The Spring AMQP Abstraction: RabbitTemplate, ConnectionFactory, and Channel

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Configuring RabbitMQ Connections in Spring Boot

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Sending Messages: RabbitTemplate and Producer Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Receiving Messages: RabbitListener and Consumer Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Message Conversion and Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Error Handling Fundamentals: `DefaultExceptionHandler` and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 12: Building Reliable Producers with Spring AMQP

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Publisher Confirms and Returns in Spring AMQP

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

ConfirmsCallback and ReturnCallback: Handling Broker Feedback

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Retry Semantics: SimpleMessageConverter and RetryTemplate Integration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Correlation IDs and Message Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Batching and Bulk Publishing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Producer Performance and Throughput Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 13: Building Reliable Consumers with Spring AMQP

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Acknowledgment Modes: AUTO, MANUAL, and BATCH

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Concurrency Tuning: Prefetch, Consumer Threads, and Queue Partitioning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The DefaultMessageListenerContainer Lifecycle and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Consumer Error Handling: Retry, Backoff, and DLQ Routing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Idempotent Consumer Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Graceful Consumer Shutdown and Message In-Flight Handling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 14: Message Serialization, Schema Evolution, and Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

JSON Serialization with Jackson: Configuration and Customization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Avro and Schema Registry for Contract-Governed Messages

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Protobuf and Performance-Sensitive Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Schema Evolution Strategies: Backward, Forward, and Full Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Versioning Messages and Supporting Multiple Consumers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Message Schema Documentation and Governance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 15: Event-Driven Design and the Outbox Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Domain Events and Integration Events: Defining the Distinction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The Transactional Outbox Pattern: Theory and Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Outbox with Spring Data and Polling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

CDC-Based Outbox: Debezium and Change Data Capture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Event Publishing in OrderFlow: From Database to RabbitMQ

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Testing Event-Driven Components

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 16: The Inbox Pattern and Idempotent Processing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Why Idempotency Matters: Duplicate Detection and Processing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The Inbox Pattern: Tracking Received Messages

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Idempotency Keys and State Stores

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Outbox-Inbox: End-to-End Reliable Communication

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Handling Partial Failures and Compensating Actions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Idempotency in OrderFlow Services

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 17: Sagas and Distributed Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Why Distributed Transactions Fail: Two-Phase Commit and Its Problems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Saga Pattern: Choreography vs Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Sagas with RabbitMQ and Spring Events

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Compensating Actions and Rollback Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Saga State Machines and Persistence

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

OrderFlow Saga: Implementing an Order Fulfillment Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 18: Message Ordering and Sequence Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

When Ordering Matters and When It Does Not

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

FIFO Queues and Single-Consumer Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Partitioning by Key for Ordered Substreams

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Sequence Numbers and Gap Detection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Order Guarantees in RabbitMQ

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Ordering Trade-offs: Throughput vs Determinism

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 19: Concurrency, Backpressure, and Flow Control

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Understanding Throughput, Latency, and Saturation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Backpressure Mechanisms: Prefetch, Queue Limits, and Rate Limiting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Spring AMQP Flow Control and Consumer Throttling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Circuit Breakers and Bulkheads: Protecting Services

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Flow Control in OrderFlow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Capacity Planning and Load Testing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 20: Security, Authentication, and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ Security: Users, VHosts, Permissions, and TLS

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Spring Security with OAuth2 and JWT

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Service-to-Service Authentication Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Secrets Management: Vault, Kubernetes Secrets, and Environment Config

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Encrypting Data in Transit and at Rest

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Security Testing and Compliance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 21: Observability — Logging, Metrics, and Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Structured Logging with Micrometer and Logback

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Metrics with Micrometer: Gauges, Counters, Timers, and Histograms

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Distributed Tracing with Micrometer Tracing and Brave/OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Correlating Logs, Metrics, and Traces

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ Metrics and Broker Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Implementing Observability in OrderFlow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 22: Containerization, Kubernetes, and Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Docker Images for Spring Boot Applications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Docker Compose for Local Development and Integration Testing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Kubernetes Deployments, Services, and ConfigMaps

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ on Kubernetes: Helm Charts and Operators

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Ingress, Networking, and Service Mesh Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Deployment Strategies: Rolling, Blue-Green, and Canary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 23: Testing Strategy for Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Unit Testing Spring and RabbitMQ Components

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Integration Testing with Testcontainers: RabbitMQ, Databases, and More

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Contract Testing with Spring Cloud Contract

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

End-to-End Testing in an Isolated Environment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Testing Failure Scenarios: Network Partition, Broker Down, Slow Consumers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Test Infrastructure and CI Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 24: CI/CD, Automation, and Release Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

CI Pipeline Design: Build, Test, and Quality Gates

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Container Registry and Image Tagging Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

CD Strategies and Automated Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Environment Promotion and Feature Flags

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Rollback Procedures and Release Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

The OrderFlow CI/CD Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 25: Troubleshooting, Maintenance, and Long-Term Evolution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Common Failure Modes and Their Symptoms

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

RabbitMQ Troubleshooting: Connections, Queues, Memory, and Disk

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Spring Boot Troubleshooting: Memory Leaks, Thread Pools, and Timeouts

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Incident Response and Debugging Production Issues

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Dependency Management and Version Upgrades

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Technical Debt, Refactoring, and Architectural Evolution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Chapter 26: The Production OrderFlow Platform

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Full Architecture: Services, Events, and Flows

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Production Deployment: Complete Kubernetes and Infrastructure Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Operational Runbook: Monitoring, Scaling, and Incident Response

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Performance Characteristics and Capacity Profiles

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Evolutionary Roadmap: How the Architecture Could Grow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Lessons Learned and Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/springrabbitmqandmodernjavaplatformengineering>.