The Addison-Wesley Signature Series

A Vaughn Vernon Signature Book

Message- and Event-Driven Systems with CQRS and Event Sourcing

Web and Cloud Architecture

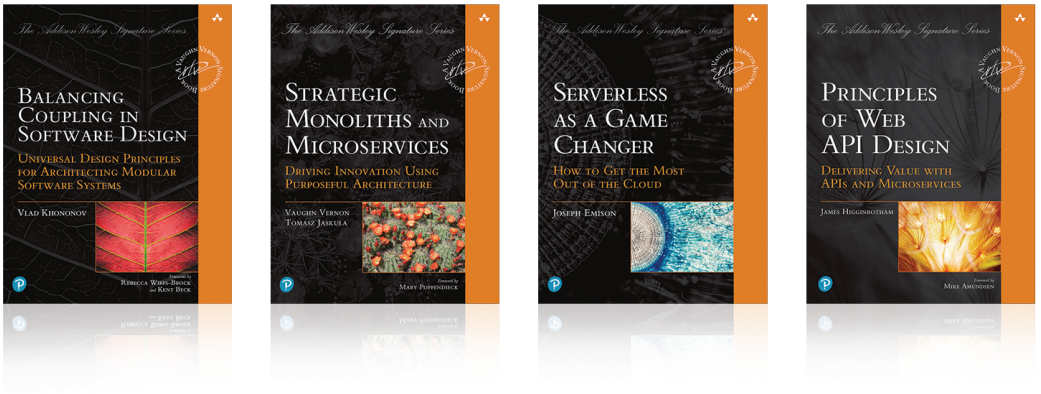
Alex Lawrence



FREE SAMPLE CHAPTER

Message- and Event-Driven Systems with CQRS and Event Sourcing

Pearson Addison-Wesley
Signature Series



The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Vaughn Vernon is a champion of simplifying software architecture and development, with an emphasis on reactive methods. He has a unique ability to teach and lead with Domain-Driven Design using lightweight tools to unveil unimagined value. He helps organizations achieve competitive advantages using enduring tools such as architectures, patterns, and approaches, and through partnerships between business stakeholders and software developers.

Vaughn's Signature Series guides readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

Visit **informit.com/awss/vernon** for a complete list of available publications.

Pearson

informIT

Message- and Event-Driven Systems with CQRS and Event Sourcing

Web and Cloud Architecture

Alex Lawrence

Addison-Wesley
Hoboken, New Jersey

Cover image: Wirestock Creators/Shutterstock

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Please contact us with concerns about any potential bias at www.pearson.com/en-us/report-bias.html.

Visit us on the Web: informit.com

Library of Congress Control Number: 2026942257

Copyright © 2027 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/en-us/global-permission-granting.html.

ISBN-13: 978-0-13-799861-6
ISBN-10: 0-13-799861-9

$PrintCode

To my wife, my children, and all my family
and friends—each of you has supported me
in your own way.

This page intentionally left blank

Contents

Series Editor Foreword . . . . . **xvii**
Preface . . . . . **xxi**

Chapter 1: Business Strategy Meets Message-Driven Systems **1**
Why Businesses Benefit 2
How Domain-Driven Scenarios Are Solved 3
Message-Driven Architecture 6
Loose Coupling 9
Relationship to Reactive Architecture 10
Relation to Monoliths and Microservices 10
Monoliths 11
Big Ball of Mud Versus Modular Monolith 11
Microservices 12
Domain-Driven Running Example 12
The Business and the Problem 13
The Idea for a First Product 13
Strategic and Tactical Design 14
Technology Stack 14
Modeling Messaging Systems 15
Domain Storytelling 16
EventStorming 17
Context Mapping 21
Summary 22
References 23

Chapter 2: Anatomy of Message-Driven Systems **25**
Message Broker 26
Brokers with Publish–Subscribe and Message Queues 27
Message Logs 28
Broker-less Messaging 30

Using RabbitMQ as a Broker 30
Introducing Messaging to WasteWarriors 33
Synchronous Communication 34
Encapsulating External Communication 35
The Structure of Messages 36
Message Identifier 36
Header 37
Message Type 38
Body 39
Message Categories 39
Commands 40
Queries 41
Responses 41
Events 42
Commands Versus Events 42
Mapping Message Categories to Topics 43
Extending the Messaging at WasteWarriors 43
Serialization, Schemas, and Types in Code 48
Serialization Formats 48
Schemas and Validation 51
Interfaces and Classes 54
Schemas and Type Usage at WasteWarriors 57
Guarantees in Distribution 57
Delivery Guarantees 57
Message Broker Requirements 58
Reliable Message Publishing 60
Ordering of Messages 65
Accept the Uncertainty 66
Ordered Consumption of Message Streams 67
Dealing with Duplicates 67
Deduplicating Messages 68
Idempotent Consumers 69
Error Handling 71
“Stop the World” 71
Dropping Messages 72
Retries 72

Dead-Letter Queue 73
Deciding on a Combined Strategy 73
Summary 76
References 76

Chapter 3: In-Context Architectures 79

The Purpose of Layers 80
Manifestations of Layers 81
Dependency Inversion 82
Dependency Injection and Containers 85
What About JavaScript? 88
Ports and Adapters 89
What Is Considered External? 91
Pros and Cons 92
Generic Layouts and Guided Designs 92
The Four Standard Layers 95
Domain Model Layer 96
Application Layer 97
Infrastructure Layer 101
User Interface Layer 105
Architecture for the Running Example 106
Summary 107
References 108

Chapter 4: Tactical Building Blocks 109

Value Objects 110
Equality by Values 113
Immutability 114
Side-Effect–Free Behavior 115
Identities 116
UUIDs and Other Formats 116
Where to Generate Values 117
Identities Are Value Objects 117
Entities 119
Value Containers 121
Behavior and State Changes 121
Entity Relationships 123

Domain Services 124
Repositories 126
Documents Versus Relational Data 127
Basic Functionalities 129
Custom Queries 131
Concurrency Control 133
Domain Events 136
Naming, Structure, and Definition 138
Creating Events 139
Message Journal and Publishing 141
Consuming Events 145
Aggregates and Eventual Consistency 147
Structure and Content 147
Transactional Consistency 150
Protecting Invariants 151
Relation to Persistence 153
Design for Concurrency 153
Eventual Consistency 154
Summary 162
References 163

Chapter 5: CQRS 165

When Business Software Should Segregate Responsibilities 166
Food Offer Reviews 168
Commands and Queries 170
Command-Query-Separation 170
Use Case Segregation 171
Message Categories 173
The Focus of CQRS 174
Command Model and Query Models 175
Command Model 176
Query Model 178
Model Synchronization 180
On-Demand Computation 180
Synchronous Update 183
Event-Based Synchronization 187
Technical Segregation 190
Implementation Parts 191
Model Persistence 192

Runtime Services 193
Use of Technologies 195
Initializing and Evolving Query Models 195
Data Migrations 196
Derive from Command Model State 197
Rebuild from Event Journal 199
Eventual Consistency in the User Interface 201
Progress Indication 201
Optimistic UI 203
Dos and Don'ts 203
Summary 205
References 206

Chapter 6: Event Sourcing 207

Your Business as Auditable History 208
Event-Sourced Food Offers 210
Modeling State as Events 211
Modeling Terminal States 214
The Relation to Domain Events 215
Event-Sourced Command Model 215
Object-Oriented Style 216
Functional Style 219
The Case for Child Entities 222
Event Store 222
Serialization Format 223
Store Topology 224
Stream Revisions 225
Global and Grouped Streams 226
Basic Operations 227
Optimistic Concurrency 230
Reactive Stream Consumption 232
Comparison of Store Technologies 234
PostgreSQL 234
KurrentDB 243
What About Apache Kafka? 245
Choosing a Technology 246
Query Model Projections 246
Combine Data from Multiple Sources 248
Design and Implementation 249

On-Demand Projections 250
Synchronous Projections 252
Reactive Projections 253
Query Models on the Command Side 257
Processing and Publishing Events 259
Local Processing of Events 259
Publishing Domain Events 261
Data Deletion 265
Summary 266
References 267

Chapter 7: Long-Running Processes 269

What Is a Process? 270
Processes in Software 270
Long-Lived and Distributed Transactions 271
Sagas 273
The Process of Reserving and Buying Food 274
Message-Driven Processes 278
Compensating Commands and Events 280
Domain-Specific Errors as Facts 280
Dealing with Technical Errors 280
Message-Driven Food Reservation Process 281
Choreography 284
Parallel Paths and Stateful Decisions 285
Mini-Choreographies 285
Implementation Design 286
Choreographed Food Reservations 286
Orchestration 290
Who Owns the Process? 291
Translation and Routing 291
Stateless Orchestration 292
Stateful Orchestration 295
Correlation of Messages 305
Traces with Causation and Correlation IDs 306
Process Coordination Through Correlation 306
Improving the Food Reservation Process 307

Process as a Domain Model Component. 308
Who Owns the Orchestration? 309
Separate Technical Process State 309
All the Same Patterns 310
Introducing the Order Aggregate 310
Evolution of Long-Running Processes. 315
Process Evolution with Downtime 316
Process Evolution Through Versioning 316
Isolating Changes Through Subprocesses 317
Improving the Order Process. 317
Summary 318
References 319

Chapter 8: Evolution of Messages and Message Streams 321

Purpose and Scope Determine Life Cycle 322
Message Purpose 322
Message Scope 324
Managing Stores in WasteWarriors. 326
Relation of Data, Serialization, Schemas, and Types 327
Introducing Schemas to WasteWarriors. 329
Types of Message Changes. 332
Adding and Removing Fields. 332
Updating the Data Type of a Field. 333
Renaming or Moving a Field 333
Merging and Splitting Fields 333
Merging and Splitting Message Types. 334
Renaming a Message Type 336
Message Changes for Stores in WasteWarriors 336
Adaptation Techniques 338
Tolerant Structure and Consumption. 338
Explicit Versioning 340
Upcasting 342
Modification of Persistent Messages. 344
Copy-Transform of Message Streams. 345
Evolution of Persistent Message Streams. 347
Identities for Derived Entities 348
References to Source Streams 351
Separating Stores and Businesses 351

Schema Registries 354
Version Control Repository 354
Shared Artifacts 355
Runtime Service 356
Schema Management at WasteWarriors 357
Summary 357
References 358

Chapter 9: Composing Data from Different Sources 359

Global Message Journal 360
Choosing the Right Technology 361
Topology of a Global Journal 363
Importing Past Messages 364
Schemas and Evolution 367
Consumption of Messages 368
Data Deletion 369
Replicating Query Models 370
Choosing a Messaging Technology 372
Stream Topology and Event Types 374
Publishing State Events 376
Replicating and Composing Data 385
Optimization and Deletion of Streams 388
Notes on Message Evolution 389
Synchronous On-Demand Composition 389
Timeouts and Partial Responses 393
Automatic Retries 396
Lazy Caching of Data 397
The "Right Way" to Compose Query Models 399
Summary 399
References 400

Chapter 10: Message-Driven User Interfaces 401

Frontend Architecture 402
The Frontend of WasteWarriors 402
Messaging Technologies and Protocols 406
Plain HTTP 407
Server-Sent Events 409
WebSockets 412

Messaging Protocols 414
Choosing the Right Technologies and Protocols. 418
Executing Commands and Queries 418
Data Composition for the UI 419
Querying Food Offer Overviews and Placing Orders 420
Reacting to Events 424
Commutative and Idempotent Reactions 424
Reacting to Food Offer Changes 425
Building Reactive Query Models 428
Dealing with Unordered Events and Duplicates 428
Additional Topics for Element Groups. 429
Reactive Food Offer Summaries 430
Client-Side Projections of Change Events 432
Summary 434
References 434

Index 435

This page intentionally left blank

Series Editor Foreword

I don't recall exactly why I came across his self-published book, but it struck me as intriguing that someone outside the familiar voices common to Event Sourcing and CQRS had especially useful things to say about the topics. How refreshing! When I saw the author's name, I thought: "Huh, a Welshman living in Munich, Germany." That compounded my curiosity. So, I contacted Alex Lawrence, and we met on a video call, where I quickly learned from his elegant German accent that he was both German and Welsh. I'll leave the rest for you to learn when you get to know him. I am certain that you'll soon meet up with Alex at a conference or some other engagement. He has a lot to share, and here's one example.

Everybody chants "event-driven!" It's become a mantra of software architects and engineers. Still, I've found it odd that "events!" seem to be all that people think about. What about other kinds of messages? One renowned high-scale and high-throughput messaging expert told me that he considered events a detail among message types. It's as if Alan Kay knew what he was talking about when he described his thinking about object-oriented: "The big idea is messaging." In their book *Enterprise Integration Patterns*, Gregor Hohpe and Bobby Woolf point out that there are at least three kinds of messages: Command, Document, and Event. I'd even add one more to the list: Query Message. When I think back to just three years into my professional career in the mid-1980s, pipes and signals provided the foundation for messaging between concurrent processes by means of Unix interprocess communications (IPCs). That experience led to even more messaging on OS/2 and later the Windows Win32 API, COM, and DCOM.

With those thoughts circulating in my mind, Alex and I soon tracked on a subject beyond Event Sourcing and CQRS, and I am glad that we jumped to the bigger picture. We discussed the possibility of teaching the industry about the broader topic of messaging, and of Alex authoring a book on message-driven architecture. After introducing the purpose of my Signature Series, I explain further.

My Signature Series is designed and curated to guide readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

From here I am focusing now on only two words: organic refinement.

The first word, *organic*, stood out to me recently when a friend and colleague used it to describe software architecture. I have heard and used the word *organic* in connection with software development, but I didn't think about it as carefully as I did then when I personally consumed the two used together: *organic architecture*.

Think about the word *organic*, and even the word *organism*. For the most part these are used when referring to living things. However, they are also used to describe inanimate things that feature some characteristics that resemble life forms. *Organic* originates in Greek. Its etymology refers to a functioning organ of the body. If you read the etymology of *organ*, it has a broader use, and *organic* followed suit: body organs; to implement; describes a tool for making or doing; a musical instrument.

We can readily think of numerous organic objects—living organisms—from the very large to the microscopic single-celled life forms. With the second use of *organism*, though, examples may not as readily pop into our minds. One example is an *organization*, which includes the prefix of both *organic* and *organism*. In this use of *organism*, I'm describing something that is structured with bidirectional dependencies. An organization is an organism because it has organized parts. This kind of organism cannot survive without the parts, and the parts cannot survive without the organism.

Taking that perspective, we can continue applying this thinking to nonliving things because they exhibit characteristics of living organisms. Consider the atom. Every single atom is a system unto itself, and all living things are composed of atoms. Yet, atoms are inorganic and do not reproduce. Even so, it's not difficult to think of atoms as living things in the sense that they are endlessly moving, functioning. Atoms even bond with other atoms. When this occurs, each atom is not only a single system unto itself but also becomes a subsystem along with other atoms as subsystems, with their combined behaviors yielding a greater whole system.

So then, all kinds of concepts regarding software are quite organic in that nonliving things are still "characterized" by aspects of living organisms. When we discuss software model concepts using concrete scenarios, or draw an architecture diagram, or write a unit test and its corresponding domain model unit, software starts to come alive. It isn't static, because we continue to discuss how to make it better, subjecting it to refinement, where one scenario leads to another, and that has an impact on the architecture and the domain model. As we continue to iterate, the increasing value in refinements leads to incremental growth of the organism. As time progresses, so does the software. We wrangle with and tackle complexity through useful abstractions, and the software grows and changes shape, all with the explicit purpose of making work better for real living organisms at global scales.

Sadly, software organics tend to grow poorly more often than they grow well. Even if they start out life in good health, they tend to get diseases, become deformed, grow unnatural appendages, atrophy, and deteriorate. Worse still, these symptoms

are caused by efforts to refine the software that go wrong instead of making things better. The worst part is that with every failed refinement, everything that goes wrong with these complexly ill bodies doesn't cause their death. Oh, if they could just die! Instead, we have to kill them, and killing them requires nerves, skills, and the intestinal fortitude of a dragon slayer. No, not one, but dozens of vigorous dragon slayers. Actually, make that dozens of dragon slayers who have really big brains.

That's where this series comes into play. I am curating a series designed to help you mature and reach greater success with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs. And along with that, the series covers best uses of the associated underlying technologies. It's not accomplished in one fell swoop. It requires organic refinement with purpose and skill. I and the other authors are here to help. To that end, we've delivered our very best to achieve our goal.

Event-driven is a subset of message-driven, and message-driven deserves full consideration. It's the "big idea." Frankly, if events were the only tool—you know, that whole hammer and nail thing—we would have a huge, tangled mess of coupling. Oh, wait! That describes the vast majority of software systems globally. We can do much better, and Alex has given you the means: architecture, domain-driven modeling, Event Sourcing, CQRS, long-running processes (also known as sagas), message type life-cycle management, data aggregation and composition, message-driven user interfaces, and a bunch of pro-tips advice scattered across the chapters. Alex has made it possible for you to love messaging and message-driven architecture!

—Vaughn Vernon

This page intentionally left blank

Preface

Message-driven systems are omnipresent in software, yet their key concepts and inner workings are generally not well understood. Software products can benefit in many ways from a purposeful application of message-driven architecture, CQRS, and Event Sourcing. One problem is that these three concepts often get poorly implemented, leading to more damage than benefits. This circumstance, in turn, contributes to the false conclusion that message-driven systems are inherently complex and offer little value to software products. As a result, developers and architects tend to favor synchronous and temporally coupled communication and integration, even for distributed systems.

This book aims to provide a definitive resource for building software products as message-driven systems and applying CQRS and Event Sourcing wherever useful. To ensure its lessons have a purposeful application, the book draws the connection to business and product concerns. For an expert technical understanding, the explanations go into depth and provide detailed implementation guidance. In essence, this is the book I would have liked to have available when starting my journey with message-driven systems. The chapters represent a curated collection of both the fundamental and advanced concepts I picked up during my work and research.

The book is relevant for all developers and architects who build software products. This includes entry-level developers who want to learn about message-driven systems and seasoned experts who are interested in advanced patterns. The chapters serve both as a practical course for learning and as a contemporary reference for concepts, styles, and patterns.

Structure and Contents

The book's structure is intended to provide an optimal reading flow from start to finish while still allowing chapters to be read individually. Chapters 1 and 2 focus on high-level aspects and system design. Chapters 3 and 4 then go into more depth for individual system parts. Chapters 5 and 6 transition to advanced patterns while staying within the same architectural scope. From Chapter 7 onward, the viewpoint returns to the system level, covering more advanced concepts and techniques. Chapter 10 completes the journey by discussing frontends and user interfaces.

In summary, the individual chapters cover the following topics:

- **Chapter 1, Business Strategy Meets Message-Driven Systems:** This chapter starts by examining how businesses can benefit from message-driven systems and explains the most relevant high-level concepts. It also introduces the running example for the book and briefly discusses how to model message-driven systems.
- **Chapter 2, Anatomy of Message-Driven Systems:** This chapter provides an overview and in-depth discussion of all relevant parts and responsibilities of message-driven systems. For every topic, the respective benefits, implications, and implementation approaches are discussed.
- **Chapter 3, In-Context Architectures:** This chapter focuses on the software architecture of individual system parts and conceptual boundaries. It provides the reader with clear guidance on how to produce well-architected and well-designed autonomous components.
- **Chapter 4, Tactical Building Blocks:** This chapter explains and illustrates a set of tactical patterns for implementing domain models and their business logic. As part of that discussion, it also covers transactional consistency boundaries and domain events.
- **Chapter 5, CQRS:** This chapter introduces the Command Query Responsibility Segregation pattern, which can be applied to separate software into command and query sides. It explains why businesses should segregate responsibilities, how such a separation can be approached, and what technical options are available.
- **Chapter 6, Event Sourcing:** This chapter explains how the state of domain models can be represented as a series of immutable change events. The discussion includes the benefits, fundamental techniques, advanced mechanisms, and the relation to message-driven systems.
- **Chapter 7, Long-Running Processes:** This chapter teaches readers how to design and implement long-running software processes in message-driven systems. It provides in-depth explanations and comparisons of different orchestration patterns and shows how to convert technical implementations into domain-specific components.
- **Chapter 8, Evolution of Messages and Event Streams:** This chapter explores how messages and their structures can evolve during the life cycle of a software product. As part of this exploration, it explains which types of changes can occur and which adaptation techniques exist to handle them.

- **Chapter 9, Composing Data from Different Sources:** This chapter focuses on the topic of composing data from different sources to create derived data structures. The techniques described here include a gathering of events in a global journal as well as asynchronous and synchronous composition.
- **Chapter 10, Message-Driven User Interfaces:** This chapter transitions to building user interfaces for message-driven systems. It explains how a message-driven architecture can bridge the gap between backends and frontends, and how to enable reactive user interfaces.

Code Examples

The book explains both how to build relevant platform parts of message-driven systems and how to implement actual software products. While some code examples are deeply technical, others exemplify domain-specific scenarios. The goal is to provide readers with enough implementation guidance to put the covered theory into practice. The code included in the book is an excerpt of an extensive collection of standalone examples and a complete sample application. The full source code is publicly available at https://github.com/alexlawrence/message-and-event-driven-systems-with-cqrs-and-event-sourcing. This repository contains not only the code but also a setup to execute the examples and even run the sample application.

The primary programming language for the code examples is TypeScript, and the backend runtime is Node.js. For the exchange of messages, this book mainly uses RabbitMQ and additionally Apache Kafka for selected scenarios. The primary persistence technology is PostgreSQL. For the most part, the SQL code examples are independent of specific databases. When discussing Event Sourcing, KurrentDB is introduced as an alternative persistence technology. Finally, Debezium is used to capture changes from databases and forward them to messaging technologies. Readers should be able to understand TypeScript and SQL but do not need to be an expert in any of these languages or technologies.

Overall, the code aims to strike a balance between being short and being idiomatic and complete. There are deliberate shortcuts and compromises to avoid overly long code examples. As such, the code should not be seen as production-ready implementations. Specifically, security aspects may be missing, and some of the code may be vulnerable to exploits. Ultimately, its primary purpose is to illustrate the concepts being addressed in the book.

Diagramming Style

The diagrams in this book follow a consistent style that is a combination of different established techniques. The primary elements in the majority of diagrams are item types from EventStorming. The colors of these item types are visible only in the electronic editions of this book; the print book uses patterns to differentiate the elements. As an additional visual aid, the item types are extended with icons. For static diagrams, the book uses arrows to indicate relationships and dependencies. For dynamic diagrams, the book loosely follows the Domain Storytelling technique by connecting actors with work items through activities. In rare cases, the figures resemble sequence diagrams. Altogether, the book follows a style similar to that described by Vaughn Vernon in the following blog post: https://kalele.io/a-blended-diagramming-style/. Chapter 1 explains some parts of the utilized style in more detail.

Register your copy of *Message- and Event-Driven Systems with CQRS and Event Sourcing* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780137998616) and click Submit. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

This book would not exist were it not for Vaughn Vernon. Vaughn contacted me some time after I finished my first e-book, which I self-published via Leanpub. This first book project was a great learning and personal success for me. However, writing a book can be very energy-consuming, especially when you are obsessed with details. After completing my first book, I decided that it would also be my last—something I kept saying to my wife over the coming months. Yet, when Vaughn offered me the chance to write a book in his Signature Series, it was clear that the only possible answer was yes.

Vaughn is one of the industry experts who significantly influenced my career and work as a software developer and architect. His book *Implementing Domain-Driven Design* was a revelation: I read it multiple times in a row and absorbed so much valuable knowledge along the way. These learnings were critical for me as they happened at a time when I needed many of the things his books explain. Obviously, when Vaughn offered me the chance to become an author in his Signature Series, my previous decision not to write another book became irrelevant. And even though the new book was as energy-consuming as the first one, I am thankful for the opportunity he gave me.

The second person I want to thank is Haze Humbert, my executive editor at Pearson. Haze was incredibly helpful throughout the whole process and ensured that this book actually got finished. Many times, she had to deal with my vague estimations and plans, accompanied by ever-shifting deadlines. Haze kept me motivated and helped pull me out of difficult situations where I got blocked in details. As much as this book would not exist without Vaughn, it would not have been finished in a timely manner if not for Haze.

I also want to thank Florian Fanderl and Radoslaw Maziarka for reviewing my book. Just as when working as a software developer, collaboration and critical feedback are crucial for writing a book. The quality of the content would not be the same without the comments and suggestions I received. Irrespective of whether it was about little details or more transformative changes, every piece of input was valuable to me.

Finally, I want to thank my wife and my two kids for their support. All three of them had to continuously adjust to me working long evenings and weekends over an extended period of time. Also, my wife had to endure many deeply technical conversations, which were especially challenging to her given that she is a nontechnical person. Without the support from my family, I would not have had the environment, time, and energy to complete this project.

About the Author

Alex Lawrence is a software developer, architect, technical lead, author, and educator. He started his journey in software development at an early age, driven by a general fascination with computers. His most notable technical expertise lies in message-driven systems, Event-Driven Architecture, CQRS, and Event Sourcing. Over the years, he has contributed to multiple companies building large-scale message-driven systems and gained deep, hands-on experience. In his work, Alex focuses on understanding business and product challenges to design software for effective and lasting solutions. His approach goes beyond technical implementations, always ensuring that software serves real-world needs and drives meaningful outcomes. Sharing knowledge has always been a core part of his work. Over the years, Alex has lectured and given guest lectures at multiple universities.

Chapter 5

CQRS

The first four chapters covered essential concepts for building a message-driven system in combination with a domain-driven approach. In such a system, the individual parts are integrated in a loosely coupled way through asynchronous and reliable high-level message exchange. Within each conceptual boundary, the business logic is isolated from infrastructural concerns and external actors by applying a fitting architecture. Every distinct domain model can be implemented using tactical building blocks, and consistency boundaries are designed explicitly. Taken together, these concepts represent a strong technical foundation that can be applied to any software. This chapter focuses on an advanced architectural pattern that builds upon this foundation.

Command-Query Responsibility Segregation (CQRS) separates actions that cause state to change from those actions that yield data. The result is that there is one command side and one or more query sides. According to Greg Young, who originally coined the term, this basic concept can be applied universally [Young-CQRS].

In practice, CQRS is utilized as an architectural pattern to divide individual system parts, as illustrated in Figure 5.1. Depending on the exact design and implementation, the query sides can maintain their own purpose-specific data structures and evolve and scale independently. Such a strict separation can be especially beneficial for independent secondary query use cases.

CQRS as an architectural pattern is often misunderstood, and an exact definition is commonly debated. On one end of the spectrum, there is the viewpoint that almost any form of use case segregation counts as CQRS. On the other end, there are stricter definitions with exact criteria. These discrepancies can make the pattern appear ambiguous. On top of that, there are many severe misconceptions about CQRS. One of them is the assumption that separate databases are necessary when,

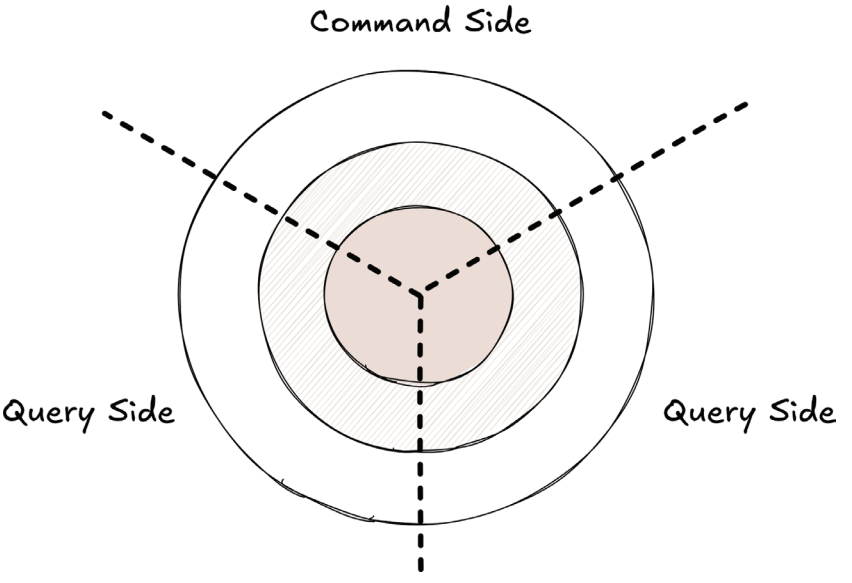


Figure 5.1 *Separating a system part into a command side and one or more query sides*

in reality, many scenarios work well with a single database. To worsen the situation, some explanations conflate CQRS with orthogonal concepts, most notably Event Sourcing. Such incorrect depictions associate this pattern with more complexity than truly exists.

This chapter aims to bring clarity in two ways. First, it isolates CQRS from other related concepts. Second, it does not attempt to formulate a detailed definition, but instead focuses on explaining the most typical components and their possible variations.

When Business Software Should Segregate Responsibilities

The segregation of command and query use cases should be driven by business and product needs. Naturally, many developers and architects are drawn toward new techniques and technologies. Applying CQRS can be an exciting activity that can produce well-designed software. Yet, the primary motivation for utilizing this pattern should be based on product requirements, necessary quality attributes, or organizational circumstances. CQRS can introduce additional complexity and non-trivial challenges, such as eventual consistency between separate model parts. These

costs should be justified with concrete needs or potential for future innovation. Promoting the segregation of responsibilities for purely technical reasons is not recommended.

Most of the business and product needs that may benefit from CQRS fall into the following categories:

- **Specialized query data:** Specific use cases must expose derived or denormalized data. This can range from computational results such as average values or percentiles to a complex composition of multiple components. While commands typically target one specific entity, query use cases may respond with data that incorporates information derived from multiple transactional boundaries.
- **Write/read ratio:** There is a significant difference between the number of executed commands and queries. In most such cases, the queries are executed more often. The ratio may be anything from 1:2 to 1:100,000,000. Moreover, the exact numbers can fluctuate based on usage patterns or the number of customers. The corresponding query sides should ideally be able to adjust to the varying load by scaling autonomously.
- **Criticality of commands:** Consistently successful execution of command use cases is critical to the product's success. The responsible components must provide the highest possible availability and resiliency. Any execution of query use cases must not affect the ability to process commands. This is especially relevant when there is a divergent write/read ratio.
- **Separate technologies:** Different technologies may be needed for the command side or one of the query sides. An exemplary scenario is when software based on legacy technology must be extended with additional querying capabilities. The existing components may be implemented in a programming language no longer supported within the organization. The new query use cases may need to be written in another language and operated as a separate runtime service.

These four categories incorporate functional requirements, quality attributes, and organizational challenges. When concrete needs arise in one of those areas, it can make sense to apply CQRS to selected system parts. Interestingly, in many projects, a combination of the first two categories drives the use of this pattern: Query use cases must expose denormalized or computed data, and there are substantially more read operations than write operations. The need for derived data may even be due to secondary purposes, such as internal reporting functionalities.

In contrast, the use of different technologies is less likely to be a necessity. Instead, it is typically a matter of available expertise or personal preference. Note that the categories identified here are indications for the usefulness of CQRS, but the right decision always depends on the specific scenario.

Food Offer Reviews

For the examples of this chapter, the focus transitions from the Food Offer Context to the Review Context of WasteWarriors. The primary motivation is that the review functionalities speak more clearly to the need to apply CQRS, which makes the examples more practical. The required capabilities of this system part are to let users write reviews and display them on store pages in different forms. This allows customers to give feedback on food offers and be more confident about choosing new stores. Within the Review Context, a specialized domain language is established that has no notion of stores, food offers, or reservations. This is because the underlying domain is generic, and its model should stay agnostic of other knowledge areas. Ideally, an implementation of this bounded context could be used in completely different software without requiring too much adaptation.

The primary component of the domain model in the Review Context is the review itself. Each review consists of an identity, an author, an item, a supplier, a date, a text, and a rating between 0 and 5. All of these attributes can be designed as value object types. The review component acts as the root element of the same-named aggregate type. The identities for authors, items, and suppliers reference entities from other system parts without adapting foreign knowledge. For this purpose, the corresponding value object types are built as a combination of a namespace and a UUID value.

Listing 5.1 shows a possible implementation of this design. The generic class `NamespacedIdentity` acts as a base for the more specialized types `AuthorId`, `ItemId`, and `SupplierId`. Within WasteWarriors, authors are users, items are food offers, and suppliers are stores. The identities are serialized as strings, where the namespace and the UUID are separated by a colon. For example, a command for writing a review may include a user identifier as an author identity in the form "user:95660c1b-5d5b-4765-8b46-630de609dcdc."

Listing 5.1 *Generic and specific classes for categorized identity value object types*

```
class NamespacedIdentity<TType extends string> extends TypedIdentity<TType> {
  readonly namespace: string;

  constructor(namespace: string, value: string) {
    super(value);
    this.namespace = namespace;
  }
```

```
  toString() {
    return `${this.namespace}:${this.value}`;
  }

  equals(other: NamespacedIdentity<TType>) {
    return other.namespace === this.namespace && other.value === this.value;
  }

  static fromString<TIdentity extends typeof NamespacedIdentity<string>>(
    this: TIdentity,
    input: string,
  ) {
    const [namespace, value] = input.split(':');
    return new NamespacedIdentity(namespace, value) as InstanceType<TIdentity>;
  }
}

// specialized categorized identity type
class ItemId extends NamespacedIdentity<'item'> {}
class AuthorId extends NamespacedIdentity<'author'> {}
class SupplierId extends NamespacedIdentity<'supplier'> {}
const itemId =
  ItemId.fromString('food-offer:ef78f0cc-6fb8-4605-a5b9-96a3cc62303e');
```

The use cases to implement for the generic Review Context are selected based on the specific needs of WasteWarriors. The state-changing actions consist of writing a review, editing a text, adjusting a rating, and removing a review. For the text editing and rating adjustment actions, it is necessary to query an individual record. Meanwhile, displaying ratings and reviews for stores requires advanced query functionalities. On every store page, three different components should be present: (1) the average rating of all reviews that belong to a store; (2) a distribution of ratings, showing the review count per rating value; and (3) a link that leads to a page where all reviews can be read. Altogether, these requirements translate into eight use cases, as summarized in Table 5.1.

The crucial architectural question is whether the product needs call for a segregation of the command side and query sides. The use cases for retrieving an average rating and a rating distribution demand specialized data that incorporates information from multiple entities. Based on the integration of reviews on store pages, there will inevitably be more read than write operations. Every review is written only once, but its data must be utilized for each visit to a corresponding store page. In view of these two circumstances, Unwasted decides to apply some form of CQRS.

As a next step, the company seeks to learn more about commands, queries, their associated models, and possibilities for synchronization.

Table 5.1 *Review use cases*

Use Case	Description
Write Review	Write a new review for an item from a supplier
Edit Review Text	Edit the text of an existing review
Adjust Review Rating	Adjust the rating of an existing review
Remove Review	Remove an existing review
View Review	View the text and rating of a review when aiming to edit it
Get Average Rating for Supplier	Determine the average rating among the reviews for a certain supplier
Get Rating Distribution for Supplier	Determine the number of reviews per rating value for a certain supplier
View Reviews by Supplier	View a list of all reviews for a certain supplier, potentially paginated

Commands and Queries

Commands and queries represent distinct concepts that can be found on multiple architectural levels of a system. Across those levels, the two concepts show the same fundamental characteristics. Commands are associated with an alteration of state, while queries yield data. Yet, their exact meaning and purpose vary depending on the context. When applying CQRS, three levels of commands and queries can be relevant. Unfortunately, these different interpretations are easily mixed and confused with each other. For a purposeful and practical application of CQRS, it is essential to understand the different levels separately before potentially combining them.

Command-Query-Separation

The Command-Query-Separation (CQS) principle states that each function must be either a command or a query [Meyer-CQS]. Commands modify state but do not return any value. Queries return data but must not change state, meaning they must execute without side effects. Although this differentiation happens on a low level, it fosters cleanly separated and predictable code designs. The type of each function should be expressed through an intention-revealing name. Commands should be phrased as instructions, such as `writeReview()`. Queries may use an imperative style like `getAverageRating()` but are often phrased as questions for boolean results, such as `isIdenticalTo()`. CQS on its own is entirely independent of DDD or CQRS and can be applied to any implementation.

Listing 5.2 shows the use of CQS with a first domain model implementation of the review entity type.

Listing 5.2 *Application of CQS in the domain model implementation of the Review Context*

```
class Review {
  /* .. */
  editText(newText: ReviewText) {
    verify('review is not removed', !this.#isRemoved);
    this.#text = newText;
  }

  adjustRating(newRating: Rating) {
    verify('review is not removed', !this.#isRemoved);
    this.#rating = newRating;
  }

  remove() {
    this.#isRemoved = true;
  }

  isIdenticalTo(other: Review) {
    return this.#rating.equals(other.rating) && this.#text.equals(other.text);
  }

  static averageRating(reviews: Review[]) {
    const sumOfReviews = reviews
      .reduce((sum, r) => sum + r.rating.value, 0);
    return sumOfReviews / reviews.length;
  }
}
```

The class `Review` exposes various operations, including the state-changing methods `editText()` and `adjustRating()`. Both of these methods expect an input and modify the state of an entity without returning a value. This behavior makes them commands. Two other operations of the review component are the function `isIdenticalTo()` and the static function `averageRating()`. Both of these functions return values but do not mutate any state. This characteristic qualifies them as queries. Creating and updating local data structures, such as for the average rating, is unproblematic. What matters for queries is that they do not mutate arguments or state of their enclosing scope. Applying CQS to a domain model implementation can help to make its classes, objects, and functions more expressive and deterministic.

Use Case Segregation

The use cases of an individual bounded context can also be segregated into commands and queries. Commands trigger behavior that leads to a persistent state change. In contrast, queries cause the retrieval of domain-specific data without

altering state. As explained in Chapter 3, the components that execute use cases are the application services. Depending on the exact implementation style, multiple approaches for segregating those services may be available. When using a categorized style, all command handlers can be put into one component and all query handlers into another. This approach works for a limited number of use cases but becomes harder to maintain for larger collections. With a dedicated style, all standalone command handlers can be co-located in one place, and the query handler components can be placed elsewhere. Such a split of use cases is independent of whether a domain model implementation applies CQS.

Listing 5.3 shows an implementation of two different review use cases for WasteWarriors.

Listing 5.3 *Example for segregating command and query use cases in the Review Context*

```
// command use case
class WriteReviewHandler implements MessageHandler<WriteReview> {
  /* .. */
  async handle(message: WriteReview) {
    const review = Review.write(
      new ReviewId(message.body.reviewId),
      AuthorId.fromString(message.body.authorId),
      ItemId.fromString(message.body.itemId),
      SupplierId.fromString(message.body.supplierId),
      this.#parseDate(message.body.date),
      new ReviewText(message.body.text),
      new Rating(message.body.rating),
    );
    await this.#reviewRepository.save(review.id, review, review.getEvents());
  }
  /* .. */
}

// query use case
class GetAverageRatingForSupplierHandler
implements MessageHandler<GetAverageRatingForSupplier> {
  /* .. */
  async handle(message: GetAverageRatingForSupplier) {
    const supplierId = SupplierId.fromString(message.body.supplierId);
    const results = await this.#reviewRepository.findBySupplier(supplierId);
    const reviews = results.map(result => result.entity);
    const averageRating = Review.averageRating(reviews);
    return new AverageRatingForSupplier({
      supplierId: supplierId.toString(),
      averageRating: averageRating,
    });
  }
}
```

The class `WriteReviewHandler` is responsible for processing commands to write a new review. It transforms incoming messages to value objects, creates an entity, and persists the resulting state via a repository. In contrast, the class `GetAverageRatingForSupplierHandler` executes the query use case of determining an average rating. As a first step, it loads all reviews for a given supplier from the corresponding repository. The resulting list of entities is then passed to the static function of the review component. Finally, the calculated rating is returned as a response message. The two handlers can be placed in separate locations together with others in the same category.

Listing 5.3 also illustrates the aforementioned rationale for using CQRS. While the write review use case affects a single transactional boundary, the query handler derives data from multiple distinct entities. Furthermore, every review is written only once (changes set aside) but is repeatedly read for rating calculations.

Message Categories

Within a message-driven system, commands and queries, alongside responses and events, are categories of exchanged messages. For this high-level exchange, the architecture and code design of individual system parts, including the use of CQRS or CQS, are irrelevant. The messages exclusively define the interfaces through which specific high-level components can be addressed and integrated. Their existence does not make any statement about implementation details. For example, within one system part, a primary messaging adapter could translate incoming messages into application service invocations with plain arguments. With such an approach, the Application Layer would stay entirely agnostic of the existence of system-level commands and queries.

In the case of WasteWarriors, a message-driven architecture facilitates an exchange of high-level messages between system parts. Listing 5.4 shows two such example messages for the Review Context.

Listing 5.4 *Example of high-level messages exchanged in WasteWarriors as JSON documents*

```
{
  "id": "4807fc29-3be3-45dc-8d0a-800c36904eec",
  "type": "WriteReview",
  "namespace": "review",
  "header": {"creationTime": "2023-10-27T11:44:43.492Z"},
  "body": {
    "reviewId": "307b3d98-5360-484b-a4ac-3e718fcd8cb9",
    "authorId": "user:9d003664-b72b-4942-a22d-8e122c16cc7f",
    "itemId": "food-offer:be647fb0-29b0-4766-9006-fe4ce417af2e",
    "supplierId": "store:e8dd1887-1c2d-4346-a4fd-6f442c8198f8",
```

```
    "date": "2023-01-01",
    "text": "I really enjoyed the food box. Staff was very friendly",
    "rating": 3
  }
}

{
  "id": "be4d256b-bf30-4275-8390-f80be97162dc",
  "type": "GetAverageRatingForSupplier",
  "namespace": "review",
  "header": {"creationTime": "2023-10-27T11:44:43.492Z"},
  "body": {
    "supplierId": "store:24fe755f-ce36-4cba-b835-5429eecdeb81"
  }
}
```

The message of type "WriteReview" is a command, while the type "GetAverageRatingForSupplier" is an example of a query. The existence of the query also implies that there is a corresponding response type "AverageRatingForSupplier." Despite this message exchange, there is no strict requirement for individual components to be implemented in a certain way. Nevertheless, it is common practice to make commands and queries first-class citizens within the Application Layer of specific system parts. Unwasted follows this approach by implementing all application services as command or query handlers.

The Focus of CQRS

CQRS as an architectural pattern is meant to be applied within individual conceptual boundaries or system parts. It primarily focuses on segregating commands and queries on the level of use cases. Command use cases are gathered into one command side, while the query use cases make up one or more query sides. In most situations, such a segregation also leads to a split of the domain model. While CQS fosters a well-designed implementation, it is not strictly required for this purpose. Similarly, employing a message-driven architecture with explicit message objects is not a prerequisite for applying CQRS. The only requirement is that commands and queries exist conceptually and somehow manifest as actions that can be executed. With that being said, the different described concepts do combine well. Within a message-driven system, there is an explicit notion of message categories, which supports the segregation of use cases. Also, applying CQS within a domain model implementation can help to make the different sides more explicit and deterministic.

Figure 5.2 visualizes the segregation of use cases for the Review Context in WasteWarriors.

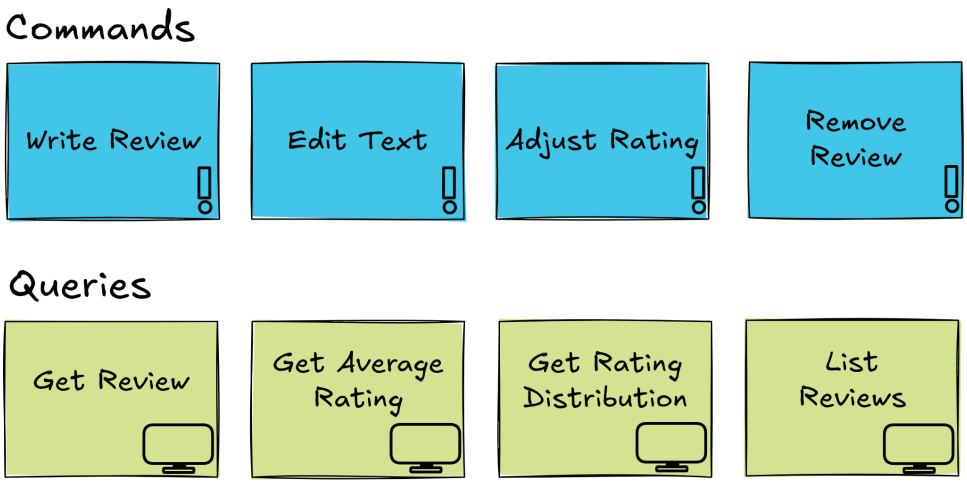


Figure 5.2 *Commands and queries for the Review Context in WasteWarriors*

In Figure 5.2, the commands include all of the state-modifying actions. Each of them is concerned with a single review entity. The query use cases are responsible for the retrieval of specialized and potentially derived data. Both the use case of getting an individual review and the use case of retrieving a review list align with transactional boundaries. In contrast, determining an average rating and a rating distribution requires a specialized model that incorporates data from multiple entities.

With the review use cases described and segregated, the next step for Unwasted is to design the distinct model parts.

Command Model and Query Models

The segregation of use cases is accompanied by a split of the respective domain model into a command model and one or more query models. Whenever use cases expose derived data, that implies the existence of multiple conceptual models. For example, suppose a result must be calculated based on a collection of entities. Along with implementing such a use case, a dedicated data structure for queries is introduced. This part should not be designed implicitly; instead, an explicit modeling effort should be made.

The different parts of a domain model do not just deal with different data; they are also associated with specific consistency guarantees. Therefore, a split should always be combined with a close examination of the transactional boundaries. However, note that a model separation does not imply any specific data segregation, such as using separate databases. There are various persistence options, of which the most important ones are explored in the next section.

Write Model and Read Model

An alternative name for the command and query model is the write and read model. These terms are commonly used when data structures are the primary focus. The main criticism of the *write/read* terminology is its inaccuracy, as both operations can be performed equally on command and query models [XOOM-Entities]. In contrast, the terms *command* and *query* are more aligned with the idea of CQRS.

Command Model

The command model is the behavioral part of a domain model component and provides the means to perform state changes. It is responsible for executing actions, maintaining transactional consistency, protecting business invariants, and supporting concurrent access. As such, it must always operate on the current state. For most software, the command side is the more critical part of a domain model. Consequently, it can make sense to invest more effort into this area instead of primarily focusing on the query functionalities. This is true even with complex query use cases, as the command part is the foundation for obtaining all necessary data.

With regard to the implementation, a command model is effectively an entity that has only command methods. Practically speaking, it may also include getters to retrieve data for persistence purposes, but they must not be accessed otherwise. The crucial design aspect is that the command side is exclusively concerned with state transitions but not specialized queries. The tactical building blocks covered in Chapter 4 are especially relevant for this behavioral part.

For each type of command model concerned with a specific aggregate type, there is usually one dedicated repository type. The required persistence capabilities are to load an aggregate via its identifier and save an aggregate's state. In addition, a repository should support concurrency control, typically through version-based optimistic concurrency.

Within the Review Context of WasteWarriors, the review command model is an entity type with operations for all previously identified state-changing use cases. The corresponding design is visualized in Figure 5.3.

An author can write a review, edit its text, adjust its rating, and remove a review. All of the command use cases target one aggregate on which the corresponding action is executed. Every method invocation causes state to change. For the implementation of this domain model aspect, most invariants are not defined directly in the root entity. Instead, the transactional constraints are enforced through value objects. For example, the value object type for a rating ensures the validity of the

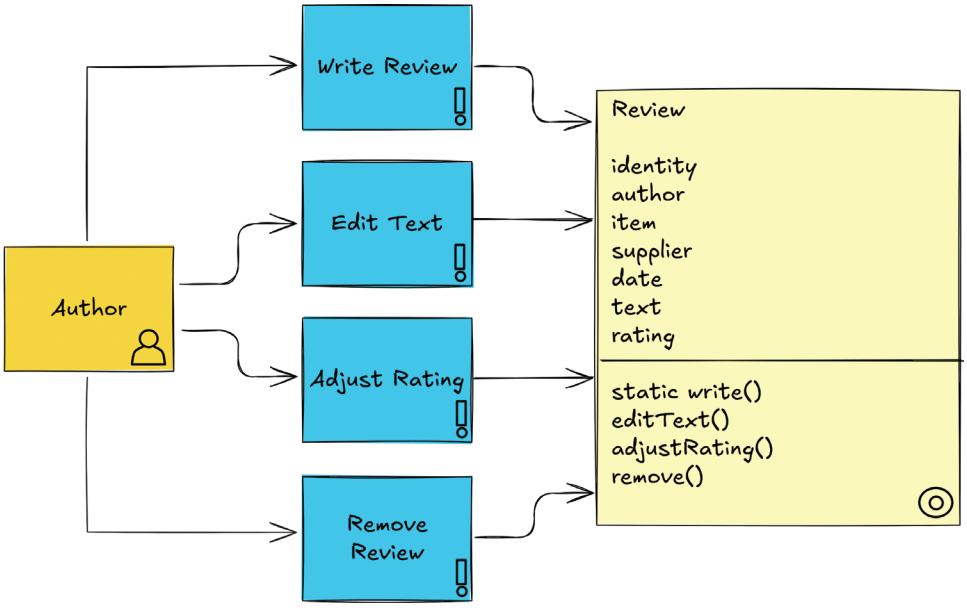


Figure 5.3 *Commands that trigger execution of behavior on a review instance*

given rating upon construction. Within the entity type, there is no notion of specialized query concerns. Listing 5.5 shows an excerpt of a first implementation for the review command model.

Listing 5.5 *Implementation of the review command model*

```
class Review {
  /* .. */
  static write(
    id: ReviewId,
    authorId: AuthorId,
    itemId: ItemId,
    supplierId: SupplierId,
    date: Date,
    text: ReviewText,
    rating: Rating,
  ) {
    return new Review(id, authorId, itemId, supplierId, date, text, rating);
  }

  editText(newText: ReviewText) {
    verify('review is not removed', !this.#isRemoved);
    this.#text = newText;
  }
```

```
  adjustRating(newRating: Rating) {
    verify('review is not removed', !this.#isRemoved);
    this.#rating = newRating;
  }

  remove() {
    this.#isRemoved = true;
  }
}
```

Query Model

A query model is a data structure for the purpose of presenting information. In most cases, it contains denormalized or derived data from one or more command models. The primary responsibility of this part is to serve specialized query use cases. While there is always just one command model per individual component, there can be an arbitrary number of query models. The ability to have multiple purpose-specific data structures is a key benefit of CQRS.

Query models are not concerned with behavior, transactional consistency, or business invariants. Depending on how their data is synchronized, these models may even be eventually consistent. This means that the information they enclose may temporarily not be synchronized with the current state of the originating command model. The topic of query model synchronization is discussed in the next section.

The exact manifestation of a query model in the implementation depends on various factors. This contrasts with the command model, which usually appears as a specific entity type. Query models can take the form of structural descriptions in the code. They can appear as table definitions, schemas, value object types, or anonymous objects. The data of a query model may be transient or stored persistently. When persistence is necessary, multiple options for the stored structures exist. The standard approach is to store fully denormalized data that is as close as possible to the final response output. This ensures that the corresponding query use cases need to perform just simple read operations. Another option is to store data in a normalized form, which can power multiple related derived models simultaneously. This approach can help to reduce duplication and maintenance overhead, though it risks making query use case executions more complex.

For the Review Context of WasteWarriors, there are three distinct query models, as illustrated in Figure 5.4. The first query model aligns one-to-one with the structure of the review entity type on the command side, except that it contains fewer fields. This part is used to render individual reviews for editing purposes and a list of

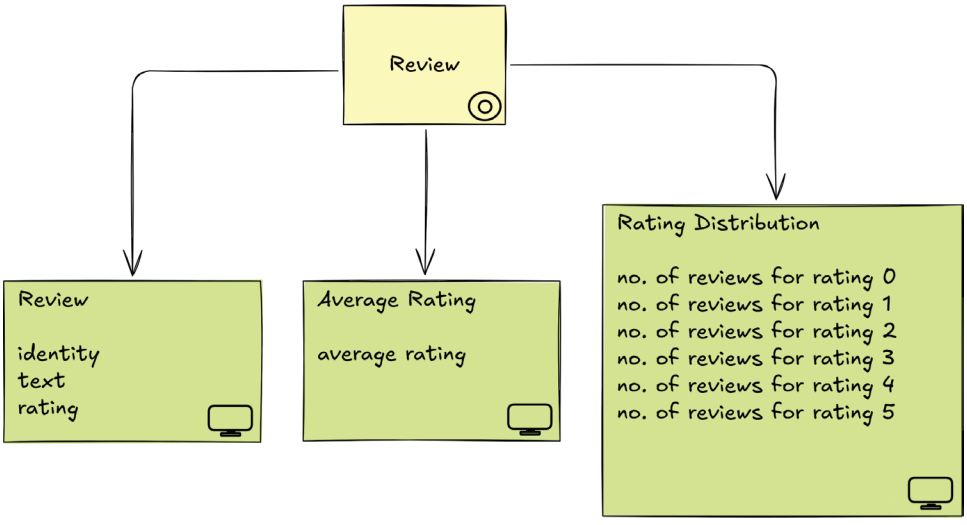


Figure 5.4 *Query models for reviews*

reviews on the store page. The second query model is the average rating, consisting of a calculated rating value. Finally, the rating distribution consists of six numbers, each representing the review count for a specific rating. Note that while the average ratings and distributions are initially required for suppliers, their data structures could be used for other scopes. For example, a future use case may be to calculate the average rating given per author.

Listing 5.6 shows a first implementation of the rating-related data structures as custom value command model.

Listing 5.6 *Value object types for rating-related query models*

```
type RatingDistribution = number[];

const createEmptyDistribution = () => [0, 0, 0, 0, 0, 0];

class AverageRating {
  #value: number;

  get value() { return this.#value; }

  constructor(value: number) {
    this.#value = value;
  }

  static fromReviews(reviews: Review[]) {
    const sumOfReviews = reviews
      .reduce((sum, r) => sum + r.rating.value, 0);
```

```
    return new AverageRating(sumOfReviews / reviews.length);
  }

  static fromDistribution(distribution: RatingDistribution) {
    const ratingCount = distribution.reduce((sum, value) => sum + value, 0);
    if (ratingCount === 0) return new AverageRating(0);
    const ratingsSum = distribution.reduce((sum, v, i) => sum + v * i, 0);
    return new AverageRating(ratingsSum / ratingCount);
  }
}
```

The rating distribution is represented as the type `RatingDistribution` together with a factory function for creating an empty object. In contrast, the average rating is implemented as the class `AverageRating`, which wraps a value and defines two static factories. The operation `fromReviews()` is the same functionality as previously implemented in the first version of the entity type. The function `fromDistribution()` provides the option to derive an average rating from a rating distribution.

With the command and query sides of the review domain model now implemented, Unwasted continues to look into how to perform data synchronization.

Model Synchronization

When a domain model is split into a command side and one or more query sides, synchronization between these sides becomes necessary. Every change to the command model must be reflected in all affected query models. Several different strategies may be used for performing such a synchronization. Naturally, each option has benefits and implications, and the most appropriate one should be chosen based on the specific scenario. Within a single conceptual boundary, a combination of different strategies may even be applied. For example, one query model might be best computed when requested, while another should be updated automatically with every executed command. Depending on the synchronization approach, query models are either transactionally consistent or eventually consistent. The different approaches and their variations can be grouped into three high-level strategies.

On-Demand Computation

The most straightforward approach for producing a query model is to compute it only when requested, as illustrated in Figure 5.5.

When a query use case is executed, the state from the command model is loaded and translated via a domain service. When this mechanism is used alone, the computed data is transient and not persistently stored. Due to this characteristic, schema

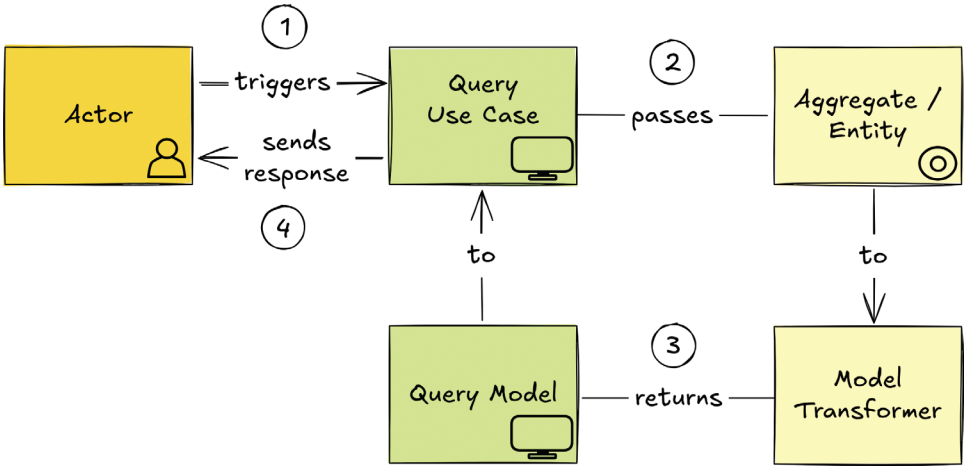


Figure 5.5 *On-demand computation of query model based on command model state*

evolutions can happen without needing data migrations. However, this approach has some critical downsides. Most importantly, the query side becomes strongly coupled to the command side. On top of that, the response time increases with the complexity of transformations and the amount of processed data. Finally, every query use case execution puts an additional load on the database where the command model is stored.

> **Does On-Demand Computation Qualify as Synchronization?**
>
> One might argue that an on-demand computation is not a synchronization mechanism and is the same as having one model. The validity of this argument largely depends on the architectural viewpoint. In the context of generating specialized query data, on-demand computation can be understood as the last possible point in time to do so.

One option to improve the response time and reduce the database load is to utilize caching. When a derived data record is first requested, it is computed, put into a cache, and returned. The next time the same record is requested, it can be retrieved from the cache. The cached data may exist only in memory or can be made persistent. Despite the clear advantages, the introduction of a cache has an important implication: It significantly increases the chance of a consistency delay for the

queried data. As long as a cache item is valid, state changes to the command model are not reflected. Only when a query is issued after an item expires is the latest data computed and cached again.

Listing 5.7 shows the implementation of an on-demand query model computation together with an in-memory cache.

Listing 5.7 *Query use case for average review rating with on-demand computation*

```
class GetAverageRatingForSupplierHandler
implements MessageHandler<GetAverageRatingForSupplier> {
  /* .. */
  @cached(
    ({body}: GetAverageRatingForSupplier) => body.supplierId.toString(),
    60_000,
  )
  async handle(message: GetAverageRatingForSupplier) {
    const supplierId = SupplierId.fromString(message.body.supplierId);
    const results = await this.#reviewRepository.findBySupplier(supplierId);
    const reviews = results.map(result => result.entity);
    const averageRating = AverageRating.fromReviews(reviews);
    return new AverageRatingForSupplier({
      supplierId: supplierId.toString(),
      averageRating: averageRating.value,
    });
  }
}

type KeyExtractor = (...args: any[]) => string;

type CacheEntry = {time: number, value: unknown};

const cached = (keyExtractor: KeyExtractor, cacheDurationMs = 1000) =>
  (_: unknown, __: unknown, descriptor: PropertyDescriptor) => {
    const originalFn = descriptor.value as (...args: unknown[]) => unknown;
    const cache = new InMemoryObjectStore<CacheEntry>();
    descriptor.value = async function cachedExecution(...args: unknown[]) {
      const time = Date.now(), key = keyExtractor(...args);
      const entry = await cache.load(key);
      if (entry && time - entry.time < cacheDurationMs) return entry.value;
      const newEntry = {time, value: await originalFn.apply(this, args)};
      cache.save(key, newEntry);
      return newEntry.value;
    };
    return descriptor;
  };
```

In this code, the class `GetAverageRatingForSupplierHandler` is responsible for the use case of retrieving an average rating. For this purpose, it accesses the command model state and computes the requested data. The method `handle()` is annotated with a caching decorator that is created via the factory function `cached`. The factory expects a key extractor and a duration as arguments. The extractor defines how to derive cache keys from the parameters of the decorated function. For the average rating use case, the key is the supplier's identity. With this mechanism, subsequent requests for the same supplier return the same cached data.

Such an on-demand computation can be a feasible option, at least as a first iteration. Even the caching duration of one minute is unproblematic for this particular use case. The main issue, aside from tight coupling, is that rating computations with many reviews may take too long and produce timeouts.

Synchronous Update

Another approach to model synchronization is to maintain a query model that is updated together with the command model. Whenever a command use case is executed, the resulting change is applied to both models within one persistence transaction. Figure 5.6 illustrates this procedure.

When a query use case is executed, the persisted derived data is accessed and returned, potentially without transformation. With this approach, the response time of query use cases can be near-constant, and no significant additional database load is produced. The main disadvantage is that command use cases become tightly coupled to parts of the corresponding query side. As the two updates must be atomic, the operations must be able to share a persistence transaction. Especially when multiple query models must be maintained, a synchronous update is not a practical choice for all of them.

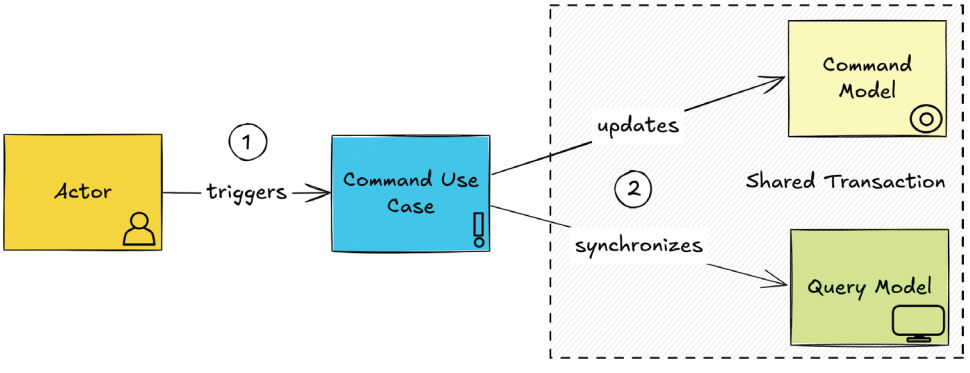


Figure 5.6 *Synchronous update of a command model and a query model in one transaction*

For the Review Context of WasteWarriors, a synchronous update can be attempted for both average ratings and rating distributions. To handle the persistence of the query model data, Listing 5.8 introduces the generic interface `ObjectStore`.

Listing 5.8 *Interface for query model storage component*

```
interface ObjectStore<TValue extends Record<string, unknown>> {
  save(key: string, value: TValue): Promise<void>;
  load(key: string): Promise<TValue | null>;
  findBy(field: keyof TValue, fieldValue: unknown): Promise<TValue[]>;
  delete(key: string): Promise<void>;
}

class PostgresObjectStore<TValue extends Record<string, unknown>>
implements ObjectStore<TValue> {
  /* .. */
}

type SupplierRatingsStore = ObjectStore<{
  distribution: RatingDistribution,
}>;
```

This component represents a key-value-based store and defines basic operations to save, load, and delete individual items. In addition, it provides a function to search for items that match a specified field value. The interface is backed by the PostgreSQL-based implementation `PostgresObjectStore`. For brevity reasons, the details of the store implementation are omitted here. The last part of Listing 5.8 defines the specific type `SupplierRatingsStore` for storing rating distributions. As explained previously, derived data may be stored in a format that powers multiple query models simultaneously. Storing the rating distributions enables the implementation to both return the data as is and compute average ratings.

Repositories Versus Stores

The persistence component for a query model can also be seen as a repository. The required functionalities are to store new or updated state and to retrieve items. However, the term *repository* is strongly associated with entities and aggregates. Therefore, it can make sense to use another word in the context of query models. In the examples in this book, only those components that exercise the behaviors typically associated with DDD repositories are labeled as such. Most of the other persistence components are referred to as *stores*.

Listing 5.9 shows a synchronous update using the command handlers to write a new review and adjust a review rating. Both classes access a review entity, execute behavior, and persist changes. Every change to the command model is saved via the review repository. The supplier ratings data is updated via an implementation of the `SupplierRatingsStore` interface. Both operations are executed within one persistence transaction using a transaction manager component.

Listing 5.9 *Example of synchronous update for the supplier ratings query data*

```
class WriteReviewHandler implements MessageHandler<WriteReview> {
  /* .. */
  async handle(message: WriteReview) {
    const review = Review.write(/* .. */);
    await this.#transactionManager.executeAsTransaction(async () => {
      await this.#reviewRepository.save(review.id, review, review.getEvents());
      const supplierIdValue = review.supplierId.toString();
      const ratings = await this.#supplierRatingsStore.load(supplierIdValue)
        ?? {distribution: createEmptyDistribution()};
      ratings.distribution[review.rating.value] += 1;
      await this.#supplierRatingsStore.save(supplierIdValue, ratings);
    });
  }
  /* .. */
}

class AdjustReviewRatingHandler implements MessageHandler<AdjustReviewRating> {
  /* .. */
  async handle(message: AdjustReviewRating) {
    const id = new ReviewId(message.body.reviewId);
    const newRating = new Rating(message.body.newRating);
    const result = await this.#reviewRepository.load(id);
    verifyExistenceOf('review', result);
    const {entity: review, version} = result;
    const oldRating = review.rating;
    review.adjustRating(newRating);
    await this.#transactionManager.executeAsTransaction(async () => {
      const events = review.getEvents();
      await this.#reviewRepository.save(review.id, review, events, version);
      const supplierIdValue = review.supplierId.toString();
      const ratings = await this.#supplierRatingsStore.load(supplierIdValue);
      verifyExistenceOf('ratings', ratings);
      ratings.distribution[oldRating.value] -= 1;
      ratings.distribution[newRating.value] += 1;
      await this.#supplierRatingsStore.save(supplierIdValue, ratings);
    });
  }
}
```

The exact details of the transaction manager are irrelevant here. With this synchronous approach, the query model data is always updated together with the command model. This has the advantage of ensuring that there is practically no consistency delay. The main downside is that the command side is tightly coupled to query concerns.

Listing 5.10 shows the query handlers for the two review use cases that yield specialized derived data.

Listing 5.10 *Query handlers to retrieve query models based on supplier ratings data*

```
class GetRatingDistributionForSupplierHandler
implements MessageHandler<GetRatingDistributionForSupplier> {
  /* .. */
  async handle(message: GetRatingDistributionForSupplier) {
    const {supplierId} = message.body;
    const rating = await this.#supplierRatingsStore.load(supplierId);
    return new RatingDistributionForSupplier({
      supplierId: supplierId.toString(),
      ratingDistribution: rating?.distribution ?? [],
    });
  }
}

class GetAverageRatingForSupplierHandler
implements MessageHandler<GetAverageRatingForSupplier> {
  /* .. */
  async handle(message: GetAverageRatingForSupplier) {
    const {supplierId} = message.body;
    const ratings = await this.#supplierRatingsStore.load(supplierId);
    verifyExistenceOf('supplier ratings', ratings);
    return new AverageRatingForSupplier({
      supplierId,
      averageRating: AverageRating.fromDistribution(ratings.distribution).value,
    });
  }
}
```

The message handler for a rating distribution loads data from the supplier ratings store and directly returns a response. The use case execution for an average rating also starts by loading the query model data. Next, the returned rating distribution is transformed into an average rating via the function `AverageRating.fromDistribution()`. As a final step, the response message is assembled and returned.

These code examples for the query handlers illustrate the main benefit of having persistent derived data. In the best case, only one record must be read, and it can be returned without performing complex transformations.

Event-Based Synchronization

The most powerful model synchronization approach is an event-based update mechanism, as visualized in Figure 5.7.

Whenever the state of a command model changes, a corresponding event is produced and published. Every interested query side consumes the event via a dedicated subscription channel. For each incoming message, the derived data structures are updated. This procedure is similar to the event-based state synchronization of aggregates, as described in Chapter 4. The key benefit is that the different model sides communicate indirectly and are temporally decoupled. Also, the response time of query use cases can be near-constant, as with a synchronous update.

The main downside of this approach is an increased delay in consistency. Compared to an on-demand computation or a synchronous update, query models are more likely to be outdated. However, in most situations, such a delay is only a matter of milliseconds to seconds.

> **At-Least-Once Updates**
>
> With an at-least-once delivery guarantee, events from a command side can arrive multiple times at a query side. Ideally, an update to a query model is performed in an idempotent way so that duplicated messages do not produce inconsistencies.

To perform event-based query model synchronization, multiple options are available depending on the event category. The database in which the command model is stored may perform Change Data Capture (CDC), publishing changes as low-level database-specific events. This approach can certainly work, but the query sides become strongly coupled to the persistence model of the command side. Another

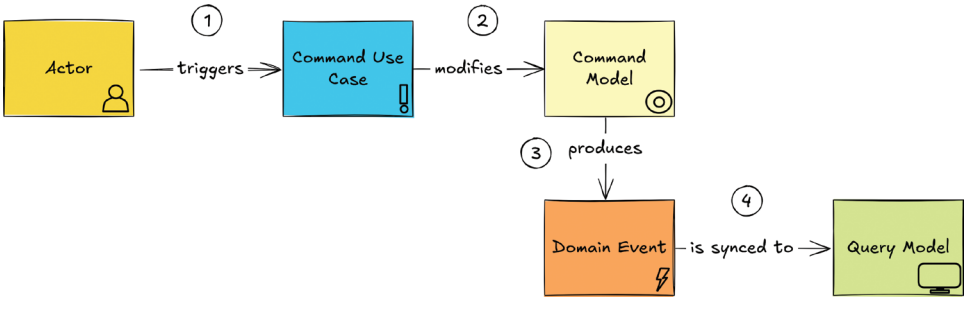


Figure 5.7 *Event-based synchronization of a query model*

option is to publish fat events containing the whole state of an entity. With this event category, a lot of unnecessary data may be transmitted. Also, a query side may be unable to figure out what changed. For well-designed domain model implementations, the most useful variant is to utilize domain events. The corresponding command model can be implemented in a way that it produces events for all relevant state transitions. Domain events express changes in the language of the model, clearly communicate intent, and can be designed with consumers in mind.

The review entity type in WasteWarriors can be easily extended to produce domain events, as shown in Listing 5.11.

Listing 5.11 *Review entity type that produces domain events*

```
class Review {
  /* .. */
  static write(/* .. */) {
    const review = new Review(
      id, authorId, itemId, supplierId, date, text, rating,
    );
    review.events.push(new ReviewWritten({
      id: review.id.toString(),
      authorId: review.authorId.toString(),
      itemId: review.itemId.toString(),
      supplierId: review.supplierId.toString(),
      date: review.date.toString(),
      text: review.text.toString(),
      rating: review.rating.value,
    }));
    return review;
  }

  adjustRating(newRating: Rating) {
    const oldRating = this.#rating;
    this.#rating = newRating;
    this.events.push(new ReviewRatingAdjusted({
      id: this.#id.toString(),
      supplierId: this.#supplierId.toString(),
      oldRating: oldRating.value,
      newRating: this.#rating.value,
    }));
  }

  remove() {
    this.#isRemoved = true;
    this.events.push(new ReviewRemoved({
      id: this.#id.toString(),
      supplierId: this.#supplierId.toString(),
      rating: this.#rating.value,
    }));
  }
}
```

In this code, the static factory function `write()` creates an event of type "ReviewWritten," which contains the full initial state of an entity. The method `adjustRating()` produces an event of type "ReviewRatingAdjusted." In addition to the new rating value, it contains the supplier identity and the old rating value. This structure is a good example of designing the event content with the consumer in mind, as described in Chapter 2. From the viewpoint of the entity type, only the new rating value would be necessary. The additional fields are placed into the event so that the query side can easily update its derived data structure. Finally, the operation `remove()` creates an event of the type "ReviewRemoved." Similarly, the event contains additional fields for updating the query model. Note that the operation that adjusts a review's text may also be changed to produce a domain event. However, for the purposes of maintaining rating-related query models, such an extension is not necessary.

The three events produced on the command side can be processed by infrastructure components on the query side, as shown in Listing 5.12.

Listing 5.12 *Event handlers to update supplier ratings query data*

```
class ReviewWrittenHandler implements MessageHandler<ReviewWritten> {
  /* .. */
  async handle(message: ReviewWritten) {
    const {supplierId, rating} = message.body;
    const ratings = await this.#supplierRatingsStore.load(supplierId)
      ?? {distribution: createEmptyDistribution()};
    ratings.distribution[rating] += 1;
    await this.#supplierRatingsStore.save(supplierId, ratings);
  }
}

class ReviewRatingAdjustedHandler
implements MessageHandler<ReviewRatingAdjusted> {
  /* .. */
  async handle(message: ReviewRatingAdjusted) {
    const {supplierId, oldRating, newRating} = message.body;
    const ratings = await this.#supplierRatingsStore.load(supplierId);
    verifyExistenceOf('ratings', ratings);
    ratings.distribution[oldRating] -= 1;
    ratings.distribution[newRating] += 1;
    await this.#supplierRatingsStore.save(supplierId, ratings);
  }
}

class ReviewRemovedHandler
implements MessageHandler<ReviewRemoved> {
  /* .. */
  async handle(message: ReviewRemoved) {
    const {supplierId, rating} = message.body;
```

```
    const ratings = await this.#supplierRatingsStore.load(supplierId);
    verifyExistenceOf('ratings', ratings);
    ratings.distribution[rating] -= 1;
    await this.#supplierRatingsStore.save(supplierId, ratings);
  }
}
```

The class `ReviewWrittenHandler` handles newly written reviews, while the classes `ReviewRatingAdjustedHandler` and `ReviewRemovedHandler` synchronize the changes to existing reviews. For performing the necessary updates, the event handlers utilize a previously introduced component, `SupplierRatingsStore`. The query use cases to expose the denormalized data are identical to when using a synchronous update. With this reactive synchronization approach, three different responsibilities can be clearly separated: executing command use cases, updating a query model, and executing query use cases. If one of those parts experiences a failure, the other two can continue to operate autonomously.

For WasteWarriors, both an on-demand computation and a synchronous update seem viable choices for experimenting with new query models. Yet, Unwasted decides that the only feasible option for a production scenario is to perform an event-based synchronization using domain events. At no point should the command side be tightly coupled to a query side. Such a design would go against the idea of building the software as a collection of autonomous and loosely coupled parts.

Technical Segregation

When applying CQRS, different levels of technical segregation are possible for the command and query sides of an individual bounded context. As explained earlier, the focus of CQRS is the categorization of use cases, which also implies a split of the domain model. This conceptual division can be mapped to various technical designs and layouts. The source code can be sliced into multiple areas, which may even be hosted in different locations. The data for the individual model parts can be maintained as distinct data structures in separate databases. Furthermore, each command and query side can be operated as one or more stand-alone runtime services. For specific scenarios, it can even make sense to utilize different programming languages or backend runtimes. When and in which forms these levels of segregation should be applied depends on the software product being built.

Implementation Parts

The implementation of a command and its query sides can either apply a single architecture or be split into separate architectural areas. Figure 5.8 visualizes both options.

For standard scenarios, it is better to apply a single in-context architecture, as this is typically easier to implement and maintain. The second option is to establish distinct areas with their own architectural layers, often accompanied by an area for shared functionality. Generally, such a strict separation can make a command and its query sides more independent. Also, it can be a necessary part of the preparation for operating multiple runtime services. The different parts may be hosted in a single code repository or be split across multiple ones. While employing separate in-context architectural areas can be a powerful approach, it can also introduce unnecessary complexity without significant benefits.

Unwasted chooses a hybrid approach for the implementation of the individual system parts in WasteWarriors. Within every context, a single in-context architecture is applied, but the command and query sides may be placed in separate directories. The implementation of the Review Context is split into four directories: one for the command side, one for review queries, one for rating-related queries, and one for shared functionalities. The shared part hosts the review entity type, associated event definitions, and the corresponding repository interface and implementation.

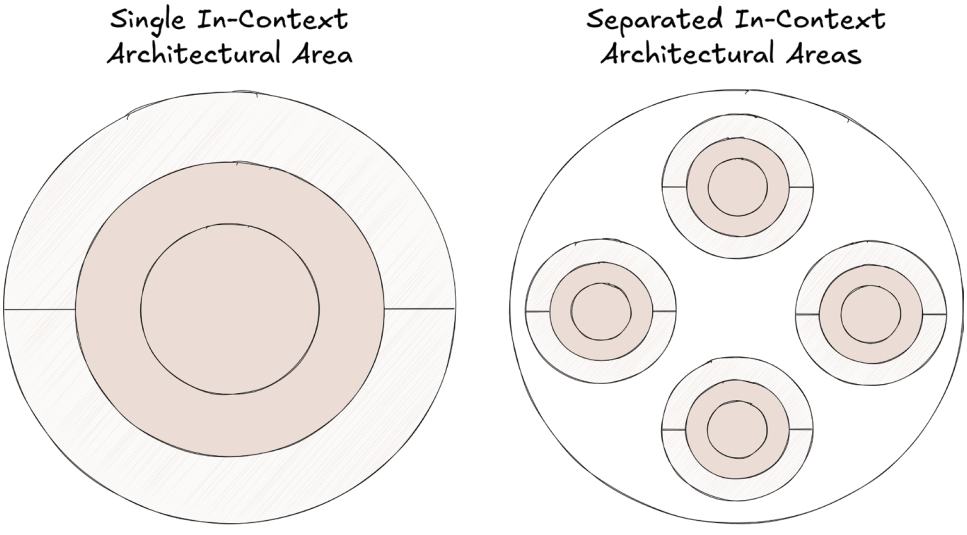


Figure 5.8 *Single in-context architecture versus separated architectural areas*

In addition to such a local part, a system-wide area for shared functionalities may exist independently of the subdomains. The resulting directory layout is shown in Listing 5.13.

Listing 5.13 *Review Context directory layout with separated command and query sides*

```
/review
  /command-side
    /application
    /domain
    /infrastructure
  /common
    /application
    /domain
    /infrastructure
  /query-sides
    /default
      /application
      /domain
      /infrastructure
    /ratings
      /application
      /domain
      /infrastructure
```

Both the command and default query sides utilize the repository and the review entity type. The event definitions are shared between the command side and the rating-related query side. The whole implementation is maintained in a single code repository.

Model Persistence

The data structures for command and query models can be maintained in a shared database or in separate ones. When possible, shared model persistence can be a good option. When aiming for a synchronous model update, it can even be necessary to achieve atomicity.

Separate databases can provide various benefits but should be chosen only when there are actual product or system needs for this approach. One such scenario is when query use cases slow down the execution of commands. With separate databases, the different sides remain independent of each other and can scale their persistence performance autonomously. Another scenario that calls for separated storage is when product requirements necessitate the use of specialized technology. This aspect

is especially relevant when applying Event Sourcing, where the command model is maintained in a particular form.

The initial approach for the Review Context of WasteWarriors is to use a single PostgreSQL database. Due to their structural similarities, the query model for individual reviews is implemented as an on-demand computation of the command model. The derived data structure for rating distributions and average ratings is maintained in a separate table. If, at some point, the rating-related queries cause a significant database load, various strategies can be used to optimize performance. For example, PostgreSQL supports replication with one writer and a configurable number of replicated reader instances. While this mechanism does not make the command and query sides entirely independent, it decouples write performance from read performance. If separate databases are still needed at some point, the split can be introduced later without a lot of additional effort.

Runtime Services

The command and query sides can be operated as any combination of runtime services. The simplest option is to execute everything within a single unit. In the case of monolithic software, this may even be the only choice. For performance and throughput purposes, a single runtime service is often sufficient. For many systems, the majority of time is not spent running actual code, but rather waiting for I/O operations. Consequently, a segregation of model persistence is the more relevant aspect for increasing throughput. Nevertheless, for evolvability and availability reasons, runtime separation can be beneficial. Each separated command and query side can change and redeploy autonomously. Also, if one side experiences a failure, it does not cause the other software parts to crash.

When applying CQRS, a standard runtime layout is to operate each side as an individual service. This design ensures that the command and query use cases are entirely independent. For query sides with an event-based synchronization, the model updates and use cases can be split again into two services. This kind of subdivision is helpful when the updates significantly outnumber the read operations, and vice versa. Figure 5.9 shows the two different runtime layouts.

Going even further, each individual use case type could be operated as an autonomous service. Such a fine-grained runtime layout can make sense if different use cases have distinct characteristics. However, it can also introduce unnecessary complexity and maintenance overhead.

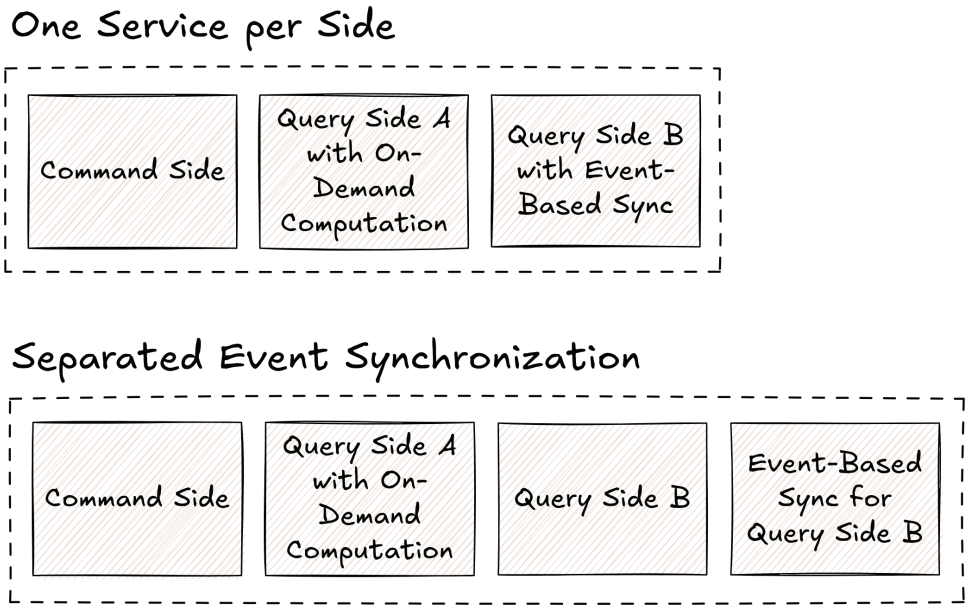


Figure 5.9 *Typical runtime layouts for a context with command and query sides*

Concurrent Query Model Updates

When a query side with an event-based synchronization scales beyond a few instances, concurrency issues may arise with model updates. For example, if two domain events for the same entity are processed concurrently, the corresponding update operations may be executed in the wrong order. This topic is further explored in the context of query model projections in Chapter 6.

For WasteWarriors, Unwasted decides to operate the command and query sides of the Review Context individually. The primary motivation is that command use cases should not influence query use cases, and vice versa. For example, reading reviews in high quantities should be independent of the ability to write or modify reviews. The event-based synchronization mechanism of the rating query model is combined with the corresponding use cases into one service. At this point, it is not foreseen that the rating query side needs to scale to many instances. If such scaling is required at some later point, the model synchronization can be broken out into a separate runtime service.

Use of Technologies

In some cases, it can be beneficial, and even necessary, to utilize distinct technologies for different sides of a bounded context. For example, certain query model computations may be expressed adequately only with a specific programming language. One issue with separate technologies is the inability to share code between the sides. On top of that, when different backend runtimes are part of the system, the respective sides cannot be operated as a single runtime unit. Given these challenges, the need to use different technologies for the command and query sides should be judged with a critical view. Sometimes, it is more a matter of preference than a necessity. Whenever possible, the command and its query sides should use the same programming language, and at least the same backend runtime.

As for WasteWarriors, there is no need to support different technologies for the command and query sides of individual bounded contexts. Unwasted prescribes a mandatory use of Node.js as the backend runtime, independent of the respective product capabilities. Regarding the programming language, it is possible to choose between TypeScript and JavaScript for each part individually. Especially for prototypical query sides, the use of plain JavaScript can avoid unnecessary overhead compared to the use of TypeScript.

Initializing and Evolving Query Models

The structure and contents of the data for a domain model must evolve over time. There are two primary reasons why such ongoing changes can be necessary. First, when the capabilities of a software product change, the corresponding data structures may also need to be adjusted. Second, a correction can be required when stored information is corrupted or faulty. The changes to perform can be of different types, ranging from simple field additions to merging multiple data structures. Also, different procedures may be needed to carry out such adjustments. In some situations, data changes can be applied while a system is running. In other scenarios, planned downtime may be necessary.

As noted earlier, query models are data structures that are derived from one or more command models. They should not be treated as the source of truth, but rather as use-case–specific views. When it comes to the initialization of new query models and the evolution of existing query models, some special considerations apply. When a new derived data structure is introduced to a running system, the existing command model data must be reprocessed. When a command model changes, it is likely that the associated query models also need adjustment. For such evolutions, it can be beneficial to recompute the derived data, making this process similar to introducing new query models. In the case of an event-based synchronization, a rebuild may need

to be based on domain events. This section discusses different techniques for initializing and evolving query model data.

Data Migrations

The default approach to data evolution is to execute a migration. This usually takes the form of a script or code that performs the necessary data changes in place. When using a SQL database, a migration script is a collection of statements, which may potentially be executed within one persistence transaction. In some cases, another set of statements can be provided to undo an evolution. If such a reversal is impossible, the executed change is considered a destructive operation. Query models can be evolved by performing migrations in the same way as with any other data structure. If both a command model and a query model need to change, the order in which the adjustments are executed can be relevant.

As an example, consider a change in the functionality of the Review Context for WasteWarriors. The initial design established that the rating in a review can be between 0 and 5. Over time, Unwasted learns from customer feedback that a 0 rating is ambiguous. Some customers understand it as the lowest value, while others think it represents the absence of a rating. Due to this confusion, the company decides to make the number 1 the lowest possible rating value. This product change requires that the existing model data also evolves. All reviews with a 0 rating must be changed to have a rating of 1, both on the command and query sides.

Listing 5.14 shows the statements for adjusting the data of the command and query models in the Review Context of WasteWarriors.

Listing 5.14 *Migration script to update command and query model data*

```
BEGIN;

-- command model
UPDATE reviews
SET data = jsonb_set(data, '{rating}', '1')
WHERE data->>'rating' = '0';

-- query model
UPDATE supplier_ratings SET value = jsonb_set(
  value,
  '{distribution,0}',
  to_jsonb(
    (value#>'{distribution,0}')::numeric +
    (value#>'{distribution,1}')::numeric
  )
);
UPDATE supplier_ratings SET value = jsonb_set(
  value,
  '{distribution,1}',
```

```
  to_jsonb(value#>'{distribution,2}')
);
UPDATE supplier_ratings SET value = jsonb_set(
  value,
  '{distribution,2}',
  to_jsonb(value#>'{distribution,3}')
);
UPDATE supplier_ratings SET value = jsonb_set(
  value,
  '{distribution,3}',
  to_jsonb(value#>'{distribution,4}')
);
UPDATE supplier_ratings SET value = jsonb_set(
  value,
  '{distribution,4}',
  to_jsonb(value#>'{distribution,5}')
);
UPDATE supplier_ratings SET value = jsonb_set(
  value,
  '{distribution,5}',
  'null'::jsonb
);

COMMIT;
```

The SQL code first updates the table `reviews` so that each record has at least a rating value of 1. Next, all entries in the table `supplier_ratings` are adjusted. For this purpose, the ratings of 0 and 1 in a distribution are summed up and assigned to the first index. Then, each of the remaining ratings is shifted one position to the left. Because the command and query models are maintained in the same database, the two migrations can be executed as one transaction. With this approach, the data structures can be evolved while the software is running, meaning no downtime is necessary.

Derive from Command Model State

In many cases, a query model can be initialized or rebuilt from the current state of the corresponding command models. For certain evolution scenarios, this approach can be a useful alternative to classical data migration. Especially when there are significant changes to a command model, a rebuild is likely more straightforward than in-place modifications. This procedure can be implemented in several different ways. When the affected models are maintained in a shared database, it can be written as a plain migration script. With separated data storage, a rebuild should be implemented as dedicated code. Also, when dealing with many entities, the data should be processed in batches.

As an example, suppose that Unwasted realizes there is an error in the WasteWarriors system: Users can write reviews for arbitrary food offers. At some point, a store

employee discovers this possibility and abuses it by writing hundreds of reviews. The employee does not use their own account but instead creates several fake accounts. As soon as Unwasted detects this issue, it adjusts the implementation to eliminate the error. The incorrect command model data can be fixed by deleting all reviews linked to a fake user account. In contrast, the data structure for the rating-related query models cannot be adjusted quite so easily, because the table `supplier_ratings` does not store any author-related information. For this scenario, a complete rebuild seems to be the only feasible approach.

Listing 5.15 shows a possible implementation for a rebuild of the supplier ratings data structure.

Listing 5.15 *Component to rebuild the supplier rating query data from the command model*

```
class PostgresSupplierRatingsRebuild {
  #chunkSize = 100;
  #dbClientPool: Pool;
  #supplierRatingsStore: PostgresSupplierRatingsStore;

  constructor(dbClientPool: Pool) {
    this.#dbClientPool = dbClientPool;
    this.#supplierRatingsStore = new PostgresSupplierRatingsStore(dbClientPool);
  }

  async execute() {
    const dbClient = await this.#dbClientPool.connect();
    await dbClient.query('DELETE FROM supplier_ratings');
    type Row = {data: {supplierId: string, rating: number}};
    const cursor = dbClient.query(new Cursor<Row>('SELECT data FROM reviews'));

    return new Promise((resolve, reject) => {
      const processChunkRepeatedly = () => {
        cursor.read(this.#chunkSize, async (error, rows) => {
          if (error) return reject(error);
          if (rows.length === 0) return resolve(null);
          for (const {data: {supplierId, rating}} of rows) {
            const ratings = await this.#supplierRatingsStore.load(supplierId)
              ?? {distribution: createEmptyDistribution()};
            ratings.distribution[rating] += 1;
            await this.#supplierRatingsStore.save(supplierId, ratings);
          }
          return processChunkRepeatedly();
        });
      };
      processChunkRepeatedly();
    });
  }
}
```

In Listing 5.15, the class `PostgresSupplierRatingsRebuild` first deletes the data from the table `supplier_ratings` and then repopulates it. For this purpose, it retrieves all rows of the `reviews` table and initializes and updates the corresponding rating distribution entries. This procedure is done in chunks with the help of a database cursor to avoid memory issues when there are many reviews. After all the entities are processed, the query model data is fully synchronized again with the command model. With this particular implementation, the rebuild may be performed only during a planned downtime of the affected system part. If it is executed while that part is running, the data of reviews that are submitted in the meantime may be corrupted or lost.

Rebuild from Event Journal

The domain events in a message journal can also be used to initialize or rebuild query model data. This approach especially makes sense when the query model is updated via event-based synchronization. Also, a rebuild from domain events can be required when the command model does not contain all the necessary data. This can be the case when a query model incorporates historical information that gets lost during command model updates. Naturally, utilizing a message journal works properly only when the persisted messages are retained indefinitely. The underlying idea of this kind of rebuild is closely related to query model projections with Event Sourcing, a topic covered in Chapter 6.

When a query model must be evolved through an event-based rebuild without downtime, one option is to work with a second data structure. For example, this can be achieved with three queues, two separate event handlers, and two data structures, as visualized in Figure 5.10.

The first queue, as shown in Figure 5.10, is the regular one consumed by the old version of the handlers. The second queue accumulates all newly occurring events from the start of the evolution. The third queue is populated with all messages from the message journal.

The new version of the event handler first consumes the journal items. When this processing is done, a switch to the queue with the newly occurring events is performed. The query data produced by the new handler is stored in a separate location. When the new handler is caught up, the old queue, handler, and persisted data can be removed.

For performing an event-based rebuild, the class `PostgresMessageJournalRePublisher` in Listing 5.16 can be used for reprocessing domain events from a message journal. Its method `publishToTopic()` sends all persisted messages of a journal table to a specified topic on a RabbitMQ broker. This republishing component is meant to be executed as a single-instance, one-time job. How the different queues must be set up and how a switch can be performed depend on the specific scenario.

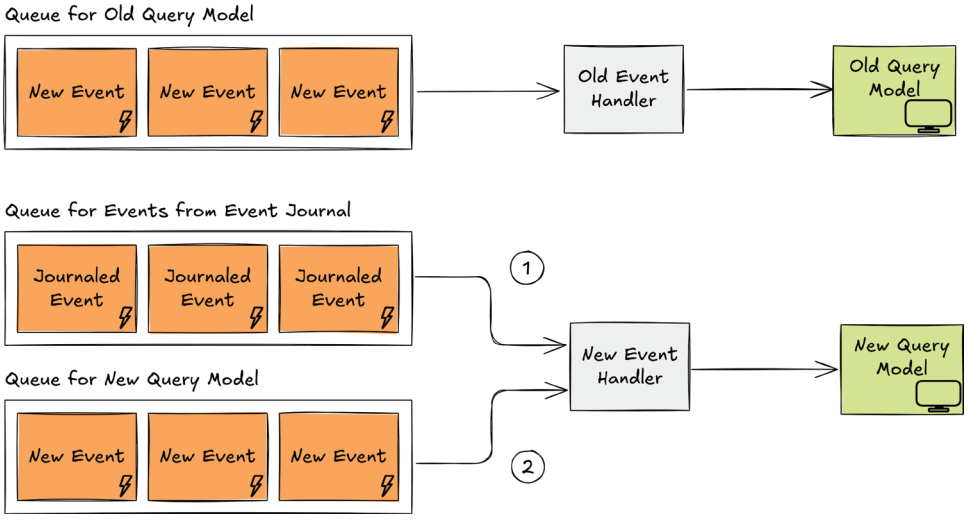


Figure 5.10 *Rebuilding a new version of the query model from an event journal*

Listing 5.16 *Component to republish journaled events to a specified topic*

```
class PostgresMessageJournalRePublisher {
  #chunkSize = 100;
  #dbClientPool: Pool;
  #messageSender: RabbitMQMessageSender;
  #table: string;

  constructor(
    dbClientPool: Pool,
    messageSender: RabbitMQMessageSender,
    table: string = 'message_journal',
  ) {
    this.#dbClientPool = dbClientPool;
    this.#messageSender = messageSender;
    this.#table = table;
  }

  async publishToTopic(topic: string) {
    await this.#forEachMessage(m => this.#messageSender.send(topic, m));
  }

  async #forEachMessage(action: (message: Message) => Promise<any>) {
    const dbClient = await this.#dbClientPool.connect();
    const cursor = dbClient.query(
      new Cursor<{message: Message}>(
        `SELECT message FROM ${this.#table} ORDER BY position ASC`,
      ),
    );
```

```
    return new Promise((resolve, reject) => {
      const processChunkRepeatedly = () => {
        cursor.read(this.#chunkSize, async (error, rows) => {
          if (error) return reject(error);
          if (rows.length === 0) return resolve(null);
          for (const {message} of rows) await action(message);
          return processChunkRepeatedly();
        });
      };
      processChunkRepeatedly();
    });
  }
}
```

For WasteWarriors and the rebuilding of rating distribution data, a planned downtime and simplified processing setup are entirely sufficient. The more advanced approach for query model evolution without downtime may be applied in the later stages of the product's life cycle.

Eventual Consistency in the User Interface

The user interface of a software product must account for the eventual consistency of its query models. Independent of the use of CQRS or the split of a domain model, any information may potentially be outdated when it is displayed. However, when there is an expected consistency delay in query data structures, there is a higher likelihood of rendering stale information. Whenever a user issues a command, there is no guarantee about when the corresponding changes will reach the query models. The possible delay depends very much on the synchronization approach and its implementation. With a synchronous model update, there can be a small or even no consistency delay. In contrast, with event-based synchronization, a derived data structure is likely to be outdated for a noticeable time. Other than with a single model, the UI components that render query data must be designed with eventual consistency in mind.

Progress Indication

One way to deal with a consistency delay in the UI is to indicate the progress of the corresponding update. Ideally, this is done by communicating that activity is occurring without stating a remaining time. Good examples of progress indicators are spinning wheels or pulsating objects, as illustrated in Figure 5.11.

The exact duration for showing a progress indicator can be determined in different ways. There can be a fixed time after which the query model is actively accessed and re-rendered. Naturally, this strategy is error-prone when synchronization frequently takes longer than expected. Another possibility is to repeatedly access the query model until the expected data is returned. However, this procedure puts an additional load on the system and requires knowledge of which data is expected. The third option is to inform the UI when an update is done, as illustrated in Figure 5.12. How such an event-based approach can be implemented is further explored in Chapter 10 in the context of message-driven user interfaces.

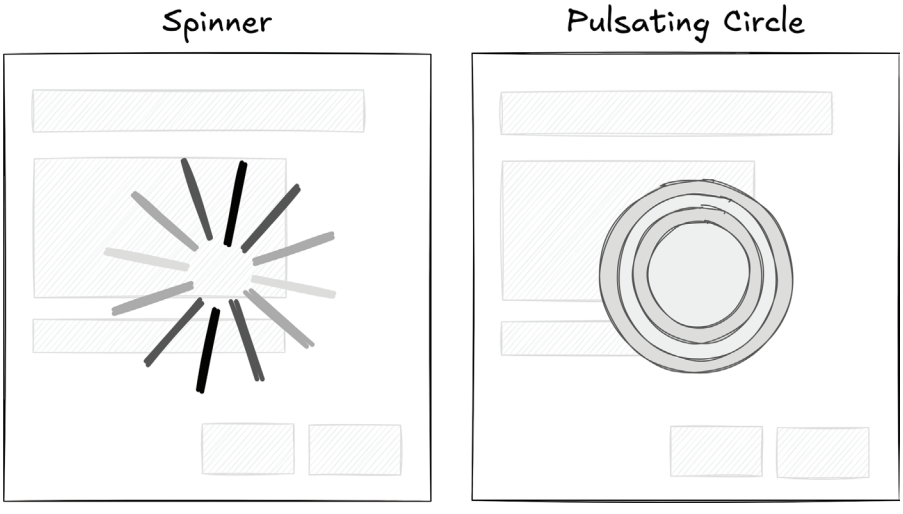


Figure 5.11 *UI progress indicators when waiting for query model update*

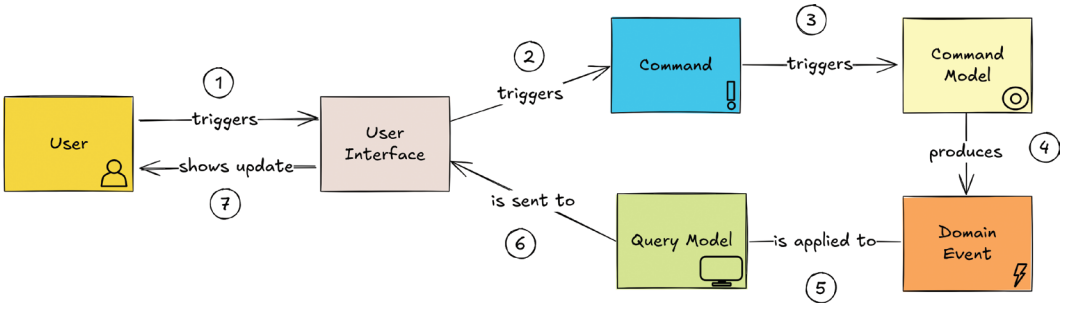


Figure 5.12 *Event-based update of UI in response to command execution*

Optimistic UI

Another solution to deal with the eventual consistency of query models is the Optimistic UI pattern. The idea here is to mimic an update of data independently of the actual use case progress or success. The user interface suggests the success of an action and updates all affected components, even though a use case execution is still ongoing. The rendered data must be based on the information that is locally available in the UI. This can also include information from the original command message. In the background, the synchronization of the query model should be verified. This verification can happen in the same ways as described in the previous subsection. If an update does not happen eventually, corrective action must be taken. The simplest approach is to inform the user that there is a problem and reload the user interface.

For WasteWarriors, there are many parts of the user interface where an optimistic approach can help with eventual consistency. One example is the placement of a food offer in the Food Offer Context. When this action is performed, the UI should report immediate success, potentially by displaying the submitted item in a list of available offers. The actual use case execution and the query model update may complete only a few hundred milliseconds later. However, given the low chance of failure, it is safe to suggest that the offer is already available. Through this mechanism, a customer experiences a seamless offer placement process with instant feedback. In the rare case of an issue with the use case or the data synchronization, the customer can be informed accordingly.

Dos and Don'ts

CQRS is a simple yet powerful approach for segregating responsibilities. When it is applied as an architectural pattern, different levels of segregation are possible, each with numerous variations. Despite these variations and nuances, certain practices are universally recommendable, and some pitfalls must be avoided whenever possible.

The following is a list of best practices when applying CQRS:

- **Be precise in communication:** Message-driven architecture, tactical DDD, CQRS, and Event Sourcing are orthogonal concepts with overlapping aspects and terminologies. Precise communication is vital to avoid confusion and conflation of the different topics. This includes not only verbal and written exchanges but also diagrams, documentation, and implementations. For example, consider the many possible meanings of commands and queries. Whenever the context is unclear, more explicit terms such as "command methods" or "query use cases" help eliminate ambiguity.

- **Transition existing components iteratively:** When CQRS is applied to an existing system part, the transition should be done in multiple steps. It is often best to start by categorizing use cases on a conceptual level, potentially followed by refactoring the code. As a next step, the domain model is split, and a synchronization mechanism is employed. Introducing different implementation parts, separating model persistence, and operating multiple runtime services are all optional steps.
- **Choose a fitting model synchronization:** The different possibilities for model synchronization are all valid approaches. While event-based synchronization is the most powerful variant, it is not a universal solution. On-demand computation is also a good option, especially when experimenting with or introducing new query models. In combination with a caching mechanism, and potentially with persistent data, it can also be a viable long-term strategy.
- **Query models on the command side:** The command side may maintain and utilize its own eventually consistent derived data structures. This approach can be useful when command use cases must look up data outside the affected transactional boundary. The topic of query models on the command side is discussed in more detail in Chapter 6.

In addition to these best practices, the following list describes some anti-patterns to avoid:

- **Enforcing CQRS as a system architecture:** When software consists of multiple parts with dedicated domain models, CQRS should not be arbitrarily mandated at the system level. The individual software parts define boundaries within which they should choose whether to segregate their use cases and domain models. This is also true for monolithic software with a single database, where the whole implementation is hosted in one code repository. Even so, building a system-wide query model layer that incorporates data from multiple domain models may be a valid choice in some scenarios.
- **Always segregating model persistence:** Many explanations of CQRS suggest that the command and query sides must use individual databases for their models. Similar to event-based synchronization, segregated model persistence can be very powerful. However, unless there are specific reasons for it, a shared database should be preferred initially. As explained earlier, there are ways to optimize persistence performance without employing separate databases.

- **Exposing implementation details:** The use of CQRS as part of an in-context architecture is an internal detail that must not leak to the outside. Other components of the same system should not rely on the fact that a bounded context segregates its responsibilities. Otherwise, this can lead to tight coupling between system parts. More specifically, a dedicated database for query models must not be directly accessed outside its enclosing boundary.

Summary

CQRS can improve the modularity, evolvability, resiliency, and scalability of selected parts of a software system. Nevertheless, it should not be viewed as a standard practice, but rather as a powerful tool for specific scenarios. Generally, the pattern should be considered only when needs exist in the areas described at the beginning of this chapter. When a system part is primarily concerned with Create–Read–Update–Delete (CRUD) operations, CQRS can introduce unnecessary complexity.

The most important takeaways from this chapter are as follows:

- The focus of CQRS is the segregation of command and query use cases within a system part.
- A domain model is split into a command model and one or more purpose-specific query models.
- There are different approaches for synchronizing derived data structures, such as event-based updates using domain events.
- The segregation of responsibilities can be mapped to various levels of technical separation, such as model persistence and runtime services.
- New query models can be initialized and existing ones can evolve through different approaches, such as reprocessing of journaled domain events.
- User interface components that render query model data must be designed with eventual consistency in mind.

Chapter 6 introduces another advanced architectural pattern—one in which the command model is represented as a series of change events.

References

[Meyer-CQS] Bertrand Mayer. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1988.

[XOOM-Entities] https://docs.vlingo.io/xoom-lattice/entity-cqrs

[Young-CQRS] https://web.archive.org/web/20211124134459/http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/

Index

A

abstract base classes, 129–130, 134
abstractions, 15, 84–85, 88, 107, 126
active event publishing, 377–381
adapters
  in custom architectures, 94, 95
  at Infrastructure Layer, 101–105
  in Ports and Adapters architecture, 89–92
advisory locks, 142, 143, 236, 238, 256
Aedes, 415–416
aggregates
  atomic transitions for, 154
  behavioral focus of, 222
  business rules across, 157–162
  concurrency for, 153–154
  event-sourced, 213, 310–315
  and eventual consistency, 147–162
  explicit definition of, 147
  and invariant protection, 151–153
  for order placement, 310–315
  and persistence, 150, 153
  root entities in, 147, 148, 215
  size of, 147, 222
  structure and content of, 147–150
  term defined, 19
  for transactional consistency, 124, 150–151
AMQP (Advanced Message Queuing Protocol), 28, 30, 60, 416–418
Anti-Corruption Layer, 145, 286
anti-patterns, software, 38, 87, 131, 139, 204, 227
Apache Kafka, 29, 33, 245, 344, 356, 361, 372, 374, 380, 383, 386, 388
Apicurio Registry, 356
APIs, 35, 44, 60, 90, 91, 97, 111, 306, 330, 356, 391, 408, 409, 412
append-only message journals, 26, 28, 61, 235, 361
Application Core, 94
Application Layer, 93, 95, 96, 97–101, 106, 126, 139, 143, 146, 174, 215–216, 246, 286
application services, 97, 98, 100, 125, 126, 127, 172, 173, 174, 214
architecture
  Clean Architecture, 95
  CQRS, 165–166, 191–192, 204
  defining externality in, 91–92
  dependency injection (DI), 86–88
  dependency inversion, 82–85
  and desired capability, 25
  event store, 224–225
  generic vs. custom, 92–95
  global message journal, 363–364
  isolating business logic via, 165
  layers in, 80–82
  for message-driven processes, 286, 310
  node clusters, 58
  Onion Architecture, 94, 95
  Ports and Adapters, 89–95
  separating process versions in, 317
  system and runtime, 79
  term defined, 79
  user interface, 401, 402–406
  See also message-driven systems
artifacts, shared, 355–356
asynchronicity, 10, 26, 31, 32, 35, 36, 60, 98, 117, 124, 136, 175, 228, 232, 249, 253, 257, 261, 293, 360, 399
at-least-once message delivery, 58, 60, 120, 155, 187, 254, 262, 376, 414, 415
at-most-once guarantee, 57
atomic transitions, 100, 147, 150, 154, 156, 183, 270–273
attributes
  of entities, 119, 121
  of identities, 116
  of immutable domain events, 138
  invariant effects on, 152
  of value objects, 111, 113
auditable business history, 208–210

authentication, 100–101
Avro, 356
AWS Simple Notification Service (SNS), 28

B

backend runtimes, 11, 14, 190, 195, 391
Backends for Frontends (BFFs), 391, 420
backfills, 377, 378, 381, 384, 389
behavior
  at the Application Layer, 98
  behavior changes by entities, 121–123
  business logic as intrinsic, 3
  of command model, 176
  of DDD repositories, 184
  decoupling retrieval from, 40, 41
  entity/service as providing, 19
  and Event Sourcing use, 210
  executed in Application layer, 97
  functional implementation of, 219–222
  object-oriented implementation of, 216–219
  of order aggregates, 313
  persistent data and product-specific, 82, 83
  in Ports and Adapters architecture, 90
  of processes, 269, 270, 276, 278
  side-effect-free, 115, 219, 249, 250
  standardizing messaging, 58, 59, 60
Big Ball of Mud, 11–12
big-picture modeling, 19, 211
binary serialization, 48, 49–51, 62, 128, 223, 243, 329, 414
body, message, 36, 39, 245
bounded contexts
  for conceptual boundaries, 4, 137
  in-context architecture of, 80
  coordination between, 6, 308
  domain events in, 137
  event store in, 223
  evolving logic in single, 9
  for feedback/reviews, 168
  for financial transactions, 275–277
  in frontend architecture, 402
  with individual domain models, 14
  mapping, 21–22
  in monolithic software, 11–12
  processes as spanning, 315
  term defined, 19
  unique identities in, 116
  use case segregation for, 171
Broadcasting/Multicasting. *See* Multicasting/Broadcasting
browser-based clients, 82, 117, 406–408, 411, 412, 414, 418, 419, 424, 432
business
  auditable history of, 208–210
  competitive advantage in, 2, 5, 321, 401
  domain event data for, 136, 141
  domain modeling for, 15
  processes in, 270
  term defined, 1
  vision for, 13
  when to use CQRS in, 167
business logic
  aggregates to protect, 147
  data isolation vs., 272
  in Domain Model Layer, 93, 96
  events as history of, 208–210, 214
  externalities isolated from, 91, 92
  handling duplicates in, 68, 69
  headers as separated from, 37
  and infrastructure functionality, 81
  intrinsic software behavior as, 3
  isolating, 82, 95, 165
  layers to serve, 81, 82
  message ordering by, 66
business rules across aggregates, 152, 155, 157–162

C

caching data, 181–183, 397–398
Cap'n Proto, 50
categorized style, 98
causation IDs, 305–308
Change Data Capture (CDC), 42, 187, 371, 381–385, 389
change events, 207, 208, 210–213, 246, 249, 262, 263, 305, 313, 315, 364, 375, 378, 387, 402, 424, 432–434
channels
  to confirm delivery, 59
  consumption on single, 38, 187, 363, 364
  content-/topic-based routing in, 27
  to organize messages, 7
  RabbitMQ, 32
  recipient-exclusive, 27
child/subordinate elements, 147, 222, 248, 274, 317, 326, 423
Choreography of processes, 284–290, 293, 316
circular dependencies, 21, 80, 285

classes
attributes of, 111
dynamically created, 54–57
OOP implementation of, 217
classical message brokers, 27, 28, 29, 33, 71, 245
Clean Architecture, 95
client/server communication, 406–418
coarse-grained stories, 16–17
Cockburn, Alistair, 89
cognitive load, lowering, 81
collection-oriented repositories, 126
collisions, identifier, 117
command model, event-sourced, 215–222
Command-Query-Separation (CQS) principle, 170–171, 174
commands
atomic, in aggregates, 150
binary format for, 51
command handling, 44–45, 48, 143, 280, 286
conceptual ordering of, 65
CQS principle for, 170
critical, 167
defined, 19, 40, 170
vs. events, 42–43
evolution of, 326–327
JSON format for, 49
in long-running process, 278–280, 281, 284, 289, 290, 305, 309
in message-driven systems, 173–174
for one-to-one communication, 321
passive-aggressive, 43
queries and, 41
as source of process state, 296
stateful orchestration of, 295–297, 301–304
state transition via, 60
UI execution of, 418–423
command side in CQRS
architecture and function of, 165–166
changes to, 195
command model overview, 175–178
on-demand computation on, 180
event-based data synchronization on, 187–190
event-sourced command model, 215–222
event store on, 223
query model projections on, 257–259
query models on, 204
rebuilding query model from, 197
synchronous command/query updates, 183–186
technical segregation for, 190–195
technologies specific to, 167
and use case segregation, 171–173
communication patterns
and message scope, 324
Multicasting/Broadcasting, 8, 9, 27, 30, 42, 43, 406–418
one-to-one, 321, 322
for Orchestration, 291
Publish–Subscribe, 8, 27–28, 432
Request–Response, 8, 9, 27, 30, 34, 43, 104, 280, 322, 325, 360, 389
for server/client messaging, 406, 409
compensating transactions, 273, 274, 278, 280
compilation tools, 49–50
components
boundaries between, 2
broker-less communication for, 30
clear purpose for, 110
communication between, 3
for data composition, 359
data points in, 111
decorators to configure, 87
for domain event processing, 145
domain models as, 96, 109
error handling within, 71
errors in health/availability of, 71
idempotent updates for, 155
identities for persistent, 117
loose coupling between, 9
message distribution among, 26
processes as domain model, 308–309
processes to span, 269, 284–285
query model projections as, 247
RabbitMQ send/receive components, 31–33, 34
in Reactive Architecture, 10
in synchronous exchanges, 35
composing data
choice of method for, 399
dedicated components for, 359
in global message journals, 360–369
via query model replication, 370–389, 399
synchronous on-demand composition, 389–398, 399, 420
system location for, 359
for the UI, 402, 419–420
use cases for, 359

conceptual boundaries
of bounded contexts, 4
business logic isolation in, 165
in-context architecture and, 79–80
contexts as, 19
CQRS in, 174
domain models within, 109
layers as, 81
concurrency control
for aggregates, 147, 153–154
via command model, 176
and event stream ordering, 236
handing conflicts, 231
in long-running process, 269, 271, 273, 277, 283, 298, 299, 302
optimistic concurrency, 133–135, 176, 230–231, 234, 237, 243, 245, 299, 302
and query model runtime layout, 256–257
in query model updates, 194
in repositories, 128, 133–136
row locks for, 238
for sagas, 273
for scalable services, 64
and stateful process orchestration, 298–299
conditional command execution, 279
consumers
domain event wording for, 138
errors and stoppage of, 71
and message scope, 324
producers and, 8, 328
tolerant message consumption, 338–340
containers, dependency injection (DI), 85, 86–87, 88, 101, 106
Context Mapping, 21–22, 23
coordination of processes
Choreography for, 284–290
correlation IDs for, 306
Orchestration for, 290–304
subprocesses for, 317
copy-transform of streams, 345–347
core functions
in Domain Model Layer, 96
in generic architectures, 4, 92
layers to separate, 3, 14, 81
Ports and Adapters to isolate, 89
as strategically critical, 109
correlation IDs, 305–308, 317
coupling
via bidirectional identifier references, 123
of command/query sides, 181, 187, 190
in data composition, 359
lowering cross-layer, 80, 81
temporal, 292, 294, 390
CQRS (Command-Query Responsibility Segregation)
best practices for, 203–205
command and query models in, 175–180
component separation via, 2, 171–173, 190
and CQS, 170
data synchronization in, 180–190, 201, 204
events in, 207
Event Sourcing and, 208
eventual consistency in, 201–203
for existing systems, 204
for feedback/reviews, 169
overview of architecture, 165–166, 174–175, 191–192
technical segregation in, 190–195
when to apply, 166–168, 174
Create–Read–Update–Delete (CRUD) operations, 205, 211
Crypto Shredding, 266, 369
custom architectures, 92–95
customers
analyzing past actions of, 209
customer-facing processes, 270
customer support, 270
data privacy for, 265, 351, 369, 370–371
eventual consistency in UI for, 201–203
high expectations of, 5, 14
improving experience of, 315
solving needs/problems of, 1, 2, 13–14
See also user interface (UI)
custom queries, 131–133, 299, 300, 302, 305

D

"The Dark Side of Event Sourcing: Managing Data Conversion", 322
data, message
and message evolution, 327–328
in message structure, 36
serialization of, 48
as string values, 37
databases
column types, 128
connection issues, 71, 280–281
CQRS in single, 165–166, 175, 204
data migrations in, 196
defining tables in PostgreSQL, 44

document vs. relational, 44, 127–129
domain model data storage in, 190
effect of processes on, 271
event store technologies, 234–246
locking for concurrency in, 64
processes for single/multiple, 271–272
streaming changes to, 64
transactional outbox in, 62–63
data caching, 181–183, 397–398
data format errors, 71
data loss, 281
data migrations, 196–197
data synchronization, 180–190
DDD (Domain-Driven Design), 1, 3, 4, 6, 7, 11, 14, 15, 21, 35, 38, 79, 89, 93, 96, 97, 109, 145, 170, 184, 203, 211
dead-letter queue, 73, 75, 259
deadlocks, 271
Debezium, 64, 240, 381–385, 389, 430
decorators, 87, 101, 183
decoupling, temporal, 9, 26, 30, 39, 187, 359, 360, 367, 399
deleting data, 129, 214, 265–266, 369, 370–371, 383
delivery guarantees, 1, 57–65, 67, 120, 187, 254, 262, 376, 414
dependencies
in/across layers, 80
circular, 21, 80, 285
core layer as free from, 96
dependency inversion, 82–85, 88–89, 106
eventual consistency to maintain, 123
Orchestration to break, 290, 292, 294
in process Choreography, 284–285, 289
dependency injection (DI), 85–87, 101, 106, 124, 126, 146
Dependency Inversion Principle (DIP), 84
derived entities, 348–350, 354
deserialized messages, 48, 50, 71, 299, 383
design-level storming, 19, 211
diagrams, Eventstorming, 19–21
diagrams, storytelling, 16–17
direct publishing of messages, 292
distributed message logs, 245
distributed systems, 11, 12, 21, 79, 399
distributed transactions, 271–272, 277–278
distribution guarantees, 29, 45, 58–66
distribution of messages, 25, 65–66
Docker, 15
Document Object Model (DOM), 404
document stores, 127–129, 153
document trees, 404
domain logic, 3, 9, 139, 317
Domain Model Layer, 84, 93, 96–97, 126, 139, 146, 162, 286
domain models
applying CQS to, 170–171, 174
behavior and state in, 110
bounded contexts in, 4, 6, 14
command model in, 176
concurrency issues in, 136
as core of system, 109
CQRS and splits in, 174, 175–180, 190, 204
descriptive aspects of, 116
designing well-defined, 310
domain events in, 136, 137
domain modeling, 15–22
domain services in, 125
entities in, 119
events in, 207
event-sourced, 211, 308, 326
evolution of, 195, 351–354
for feedback/reviews, 168
identities in, 116
intermediate states in, 157
invariants in, 151
isolating implementation of, 89
layers in, 81
mapping, 21–22
and message evolution, 346–347
multiple timelines in, 210
Orchestration in, 291
processes in, 273, 308–315
repositories in, 126
storage for components of, 127
synchronization of split, 180
term defined, 3
terminal states in, 214
value objects in, 110–115
domain services, 109, 124–126, 139, 140, 143, 180, 262–265, 287
Domain Services layer, 94
domain-specific data
and command model, 215, 217
decoupling, 84
delivery guarantees for, 57–58
in domain events, 42
in Domain Model Layer, 97

in entities, 119
in events, 42, 136, 145, 211
explicit typing for, 54
idempotence and, 69, 218
isolated from external concerns, 81
in message body, 39
and message ordering, 66
in processes, 273, 280, 291, 295, 296, 308, 310, 316
in query model, 145
in query model projections, 246, 250
in query responses, 41, 171
stateful orchestration of, 296
upcasting, 344
in value objects, 115, 119
Domain Storytelling, 16–17, 18, 23
dropped messages, 72
duplicate identities, 117
duplicate messages
exactly-once processing for, 58, 68–69
idempotent consumption for, 69–70, 187
identifiers for de-duplication, 37, 62, 149
infrastructure/business logic to handle, 58
in publishing state events, 376
in reactive user interfaces, 428–429

E

ECMAScript Programming Language Standard, 49
EDA (Event-Driven Architecture), 6, 7
encapsulating external communications, 35–36, 313
enrichment of domain events, 140
entities
behavior/state changes for, 121–123
command model, 176–177
creating new, 69–70, 83–84, 97
dedicated entity types, 137
deleting, 129, 214
event-sourced entity implementation, 216
identities for, 116, 348–350
merging/splitting types of, 347
process entities, 308–309
querying, 41
relationships between, 123–124
repository functionalities for, 129
responsibilities of, 121
root entity with child elements, 147, 154, 222, 248
terminal states for, 214
as value containers, 121
value object use by, 121, 123
equality by values, 113–114
equals() method, 114, 118
errors
atomic transitions to prevent, 150
and automatic retries, 135–136
Big Ball of Mud, 11
component health/availability, 71
concurrency conflicts, 135–136, 231
confusing commands and events, 43
data format errors, 71
deduplication, 69
and delivery retries, 72–73, 75
DI container, 86
dropped messages, 72
duplicate messages, 58
event naming to reduce, 138
Event Sourcing, 210
global message journal, 367, 368
innovation and, 5
in long-running process, 280–281
loose coupling for, 9
message handler for, 71–76, 99
in microservices, 12
in Reactive Architecture, 10
runtime errors, 88
validation, 71
Evans, Eric, 110, 147
event-based data replication, 245
event-based data synchronization, 195
Event-Carried State Transfer, 42, 324, 348, 371, 372, 389, 428
Event-Driven Architecture (EDA), 6
event logs, 2, 141–145, 207
events
for aggregates, 148
categories of, 42, 424
changes described by, 42
commands and, 42–43
conceptual ordering of, 65, 67
consumption/processing of, 145–146, 361, 364
creating, 139–140
defined, 19, 42
designed for consumers, 188
enriching, 140
event-based data synchronization, 187–190, 194, 201

Event-Carried State Transfer with, 371, 374
Event Sourcing records and, 211, 215
EventStorming to identify, 19
explicit, 137, 374
as facts, 136, 207, 280, 321, 344
frontend publication of, 401–402
identities for, 116, 118
implementation for handling, 45–48
journals of, 61, 141–145, 199–201, 207
local event processing, 259–261
in message-driven processes, 278–280, 281, 284, 290, 305, 309
modeling states via, 207–208, 210–211
naming and structure of, 42, 138–139
processing heterogenous, 213
prototypical exchanges of, 136, 137
publishing, 60–61, 141–145, 300, 424–428
stateful orchestration of, 295–304
state synchronization via, 155–156, 158
transactional outbox for, 100
event-sourced command model, 215–222
Event Sourcing
aggregate IDs in, 213
auditable history via, 2, 208
backfills with, 378
conceptual ordering in, 67
concurrency control for, 230–231
and CQRS, 166, 208
data deletion and, 265–266
events as facts in, 136, 207, 280, 321, 344
vs. EventSource, 411
event-sourced command model, 215–222
Event Sourcing records, 212, 215
event stores, 223–224
immutability in, 211, 216, 217, 344
in message-driven processes, 279, 280
message streams in, 324
modeling states as events in, 207–208, 211–215, 310
naming event types in, 213, 214
for order aggregates, 310–315
persistent message streams in, 347
processing/publishing events in, 62
query model projections, 246–265, 368
for stateful process coordination, 270
store technology options, 234–246
terminal states in, 214
terminology, 211
when to use, 209–210
EventStoreDB, 243
event stores
basic operations of, 227–230
event streams in, 141, 225–227
evolution of, 344–345
frontend access to, 432–433
and global message journals, 362
reactive consumption in, 232–233
serialization for, 223–224
store technology options, 234–246
topography of, 224–225
EventStorming, 17–21, 23, 137, 211
eventual consistency
in CQRS, 166, 178, 180
in entity relationships, 123
as tactical building block, 154–162
in the UI, 201–203
evolution of global message journals, 367
evolution of messages. *See* message evolution
exactly-once processing, 58, 60, 68
exchanges, 30
explicit classes, 56
explicit events, 375
explicit schemas, 51, 54, 127, 329
explicit versioning, 340–342
external actors, 91, 98
external communication, encapsulating, 35–36
external communication, message evolution in, 325
externality, defining functional, 91–92

F

factories, 55–56
failures
broker replication for, 58
and delivery retries, 35, 72–73, 75
isolating, 2, 5
in message-driven processes, 280–281
See also errors
fat events, 42, 188
feedback system, 168–170
Fetch API, 408
financial transactions, 67, 209, 275–277, 280, 281, 283, 288
find() method, 131
fixed state, 207, 257, 261, 296, 300, 308, 364, 365, 378
Functional Core with Imperative Shell, 94
Functional Programming (FP), 94, 110, 122, 219–222

G

General Data Protection Regulation (GDPR), 265
generic type Message, 55
generic vs. custom architecture, 92–95
GitLab, 355, 356
global event journal, 361, 362, 363, 364, 367
global event streams, 226–227, 235–242
globally unique identities, 116
global message journals, 360–369
global order locks, 238
granular scaling, 2
graphical user interface, 105
GraphQL, 391
grouped event streams, 226–227, 374

H

headers, message
  for commands, 41
  dynamic classes for, 56
  in message structure, 36
  original query in response header, 46
  overview of, 37–38
  in server/client exchanges, 411, 414
Hexagonal Architecture, 89
high-level architectural elements, 79, 84, 85, 88, 89, 92
high-level messages, 2, 3, 6, 12, 23, 165, 167, 173, 175, 180, 215, 269, 271, 279, 281, 381, 386
Hohpe, Gregor, 7
HTTP
  for command/query execution, 419, 420
  HTTP-based communication, 104, 105
  for server/client messaging, 407–409
  and WebSockets, 412

I

idempotent behavior, 69–70, 120, 122, 155, 187, 218, 221, 254, 424, 425, 427
identifiers
  bidirectional references for, 123
  correlation IDs as, 305–306
  deduplication with, 68
  dynamic classes for, 56
  in message-driven processes, 286
  in message structure, 36–37
  and new entity creation, 69–70
  for server/client messaging, 410
  for stateful process coordination, 295, 299–300, 303
  in transactional outboxes, 62
identities
  absence of conceptual identity, 113
  and cache keys, 183
  for derived entities, 348–350
  in Event Sourcing, 213
  for feedback/reviews, 168–169
  identity formats, 37, 116, 117–118
  for individual processes, 295–297
  overview of, 116–119
  in payment processes, 275, 286–287
  of process entities, 309
  storage for, 129
  as tactical building block, 109
  for user reviews, 168–169
Identity & Access Management (IAM) Context, 33, 138, 139
immediate consistency, 150
immutability, 28, 29, 110, 114, 115, 116, 119, 138, 210, 211, 216, 219, 221, 249, 344, 345
implicit schemas, 51, 328
imports, message journal, 364–366
independent message throughput, 43
information retrieval, 98
Infrastructure Layer, 96, 101–105, 106, 124, 125, 146, 227, 286, 291, 374
initialization of query models, 195–201
in-memory caches, 181–182
in-memory maps, 69
innovation, business, 5
input queues, 43
instance retrieval, 87
interfaces
  classes and, 54–57
  dependency inversion, 84–85
  domain services as, 125
  event journal, 141
  event store, 227, 228, 230
  generic message handler for, 98–99
  for high-level exchanges, 173
  query model storage interface, 184
  reactive, 246
  repository, 126, 129
  separating domain- and technology-related, 131
  TypeScript repository, 87

intermediate states, 67, 154, 157, 170, 271, 272, 273, 278, 280, 281
internal processes, 270
invalid state prevention, 150, 152, 230, 283
invariants, protecting, 147, 150, 151–153, 155, 157, 162, 176, 215, 218, 219, 228, 231, 269
inversion, dependency, 82–85, 88–89, 106

J

Jaskula, Tomasz, 93
Java, 11, 15, 29, 381
JavaScript
  binary format for, 51
  compilation tools, 50
  as CQRS runtime service, 195
  dependency inversion via, 88–89
  enforcing immutability in, 114
  JSON functionalities in, 91
  for JSON Schema, 52
  message types and, 54, 57
  representing identities in, 117–118
  syntax for, 49
  vs. TypeScript, 55, 88
  without explicit structure, 328
Java Virtual Machine (JVM), 11
journals, event, 62, 141–145, 199–201, 207
journals, message, 360–369, 378
JSON data queries, 132
JSON Schema, 52–53, 57, 329–331, 339, 340, 356, 357
JSON serialization
  for client communication, 407, 410
  for document storage, 128
  for event-related persistence, 223–224
  KurrentDB with, 243
  and message evolution, 327–328
  overview of, 49
  for RabbitMQ message exchanges, 31, 33
JSX, 404

K

Kafka. *See* Apache Kafka
Kafka Connect, 245, 356, 381
Kafka Streams, 245
key–value pairs, 41, 55
K-Sortable Unique IDentifier (KSUID), 117
KurrentDB, 243–245, 246, 344, 361, 362

L

Layered Architecture with Dependency Inversion, 95
layers
  four standard, 95–106
  function of architectural, 80–82
  generic vs. custom designs, 92–95
  segregated event layers, 263–264
legacy systems, conceptual boundaries in, 4
life cycle of messages, 322–327
loading entities, 129
local event processing, 259, 262, 266
local transactions, 271–272
locking, 142, 143, 236, 238, 256
logs
  database transaction, 64, 240
  distributed message logs, 245
  of domain events, 141
  message logs, 26, 28–29, 33, 324, 361, 388
  of message processing errors, 71
  for reactive stream consumption, 240–241
long-running processes
  atomic transitions in, 270, 271
  Choreography of, 284–290
  coordination of, 270, 284, 290
  correlation of messages in, 305–308
  defined, 269, 270, 274
  domain concepts in, 310
  in domain models, 308–315
  embedding entities in, 37
  errors in, 280–281
  evolution of, 306, 315–318
  intermediate results in, 272, 273
  local vs. distributed transactions, 271, 277
  in message-driven systems, 278–284
  Orchestration of, 290–304, 310
  reversal in, 273, 275–277, 280, 281, 283, 288–289, 302, 316
  simple vs. complex, 271
  terminal state of, 316
  timeouts for, 302
loose coupling, 9, 10, 26, 34, 95, 145, 284–285

M

maintenance, 81, 178, 193, 234, 245, 290, 316, 317, 329
managers, process, 271, 290
mapping contexts, 21–22

Martin, Robert C., 84, 94
merging entity types, 347
merging message types, 334–336
message brokers
  autonomous operation of, 48
  classical, 27, 28, 29, 33, 71, 245
  as cluster of nodes, 58
  delivery guarantees by, 58–60
  domain event consumption via, 145–146
  Kafka products, 245
  for local event processing, 259, 260
  logs for, 28–29
  in message-driven processes, 281
  overview of, 26–35
  as primary and secondary actor, 90, 91
  to publish domain events, 261–262
  with PubSub and message queues, 27–28
  RabbitMQ as, 28, 29, 30–33
  relaying messages via, 26–27
  for server/client messaging, 413, 416–418
  technologies for, 28
  term defined, 26
message bus, 26
message-driven systems
  autonomy of parts in, 2, 5
  bounded context interaction in, 6
  business benefits of, 1
  category/topic mapping in, 43
  component communication in, 3, 4, 23
  data composition in, 359, 399
  distribution guarantees in, 58–66
  domain events in, 140, 141, 162
  error handling in, 71–76
  events in, 207
  evolution of, 336, 357
  long-running processes in, 269–270, 278–284, 318
  loose coupling in, 9, 34
  message categories in, 39–48, 173–174, 321
  for Monoliths/Microservices, 10–12
  ordering of messages in, 65–67
  for reactive systems, 10, 22
  reliable delivery in, 57
  for runtime services, 10
  senders and receivers in, 8
  specialized message types in, 54
  technical knowledge for, 25
  term defined, 6–7
  three complementary models for, 2
  traces in, 306
  transactional consistency in, 162
  UI in, 105–106, 401
  validation in, 51–53
message evolution
  adaptation techniques, 338–347
  backward-/forward-compatible, 332–336, 339
  and data/serialization/schema/type, 327–332
  evolution of persistent streams, 347–354
  and message purpose/scope, 322–327
  types of changes, 332–338
message handlers
  adapters and, 103, 104
  and application services, 97, 101, 106
  for correlation of messages, 305
  data extraction by, 340
  dependency inversion and, 83, 84, 85, 87, 88
  errors as transformed by, 280
  for message-driven processes, 286–289
  MessageHandler interface, 98
  and order aggregates, 314
  and query model replication, 384, 385, 387, 389
  query model retrieval by, 186
  for server/client messaging, 419
  for stateful process coordination, 298, 299
  in stateless Orchestration, 292, 294
  validation of, 331
message journals, global, 360–369, 378
message logs, 26, 28–29, 33, 324, 361, 388
message relays, 64, 143–145, 240, 292, 302, 376, 378
message streams
  copy-transform of, 345–347
  derived, 348–351
  embedding source references in, 351
  for event-based state transfers, 374
  evolution of, 321, 324, 338, 347–354, 389
  message logs for, 28, 141
  message purpose in, 323
  optimizing, 388–389
  ordered stream consumption, 67
  RabbitMQ for, 29
  streaming terminology, 29
  in the UI, 432
messageType() operation, 56
message types
  data/serialization and, 327–328

and field changes, 332–334
interfaces and classes for, 54–56
merging/splitting, 334–336
message body and, 39
in message structure, 36
overview of, 38, 321
and purpose, 322
renaming, 336
for server/client messaging, 410
messaging
adapters for, 103–105
architectural overview, 25
broker-less, 30
categories of messages, 25, 39–48, 321
channels to organize. *see* channels
content- and topic-based routing, 27
distribution guarantees, 58–66
for domain events, 138–139
duplicate messages. *see* duplicate messages
Event Sourcing records, 212
idempotent processing of, 121
life cycle of messages, 322–327
message correlation in processes, 305–308
message structure, 25, 36, 338–340
messaging patterns, 7
Orchestration of, 291–292
ordering of messages, 66–68
order of messages. *see* order of messages
to publish state events, 376
reliable message publishing, 60–65
specialized message types, 54
structure/terminology in, 36–39
synchronous, 34–36
term defined, 6–7
two basic forms of, 8
UI protocols/technologies for, 406–418
metadata
domain event, 136
Kafka for storing, 356
KurrentDB, 245
message, 36
message header as, 37
upcasting, 343–344
Microservices, 10–12, 14, 21
Microsoft Teams, 5
migrating data, 196–197
mini-choreographies for processes, 285–286
modeling state as events, 211–215
modifying persistent messages, 344–345
modular monoliths, 11–12
monolithic frontend codebases, 106, 402
Monoliths, 10–12, 14, 21, 106, 204
MQ Telemetry Transport (MQTT), 28, 414–418
Multicasting/Broadcasting
events in, 42, 43
loose coupling in, 9
mapping categories in, 43
mapping messages to topics in, 43
message purpose in, 323, 324
routing in, 27, 30
for server/client messaging, 406–418
term defined, 8

N

nack() function, 75
namespaced identities, 116, 306, 318
naming
of commands, 41
of derived entities/streams, 350
of domain events, 138–139
of event types, 42
general recommendations for, 138
of headers, 37
of queries, 41
renaming fields, 333
renaming message types, 336
of response types, 41
type names, 38
NanoID, 117
nested message type, 49, 50
nested objects, 36
network protocols, 414
node clusters, 58
Node.js
Aedes for, 415
as backend runtime, 14, 195
and DI containers, 86
Express library, 407
integration with RabbitMQ, 30
for KurrentDB, 243
spawning two processes in, 34
Temporal API for, 111
UUID for, 117
npm package pg, 129
npm package amqplib, 59
npm package jsonschema, 53, 329, 332, 356
npm package KafkaJS, 374
npm package protobufjs, 50, 51, 52
npm package ws, 413

O

object-oriented programming (OOP), 121, 216–219
Object- Relational Mapping (ORM), 127
objects
  attributes of, 111
  data exchange as, 48
  persistence of aggregates as, 153
  syntax for JavaScript, 49
  value objects, 110–115
on-demand computation, 180–183, 187, 190
on-demand data composition, 389–398, 399
on-demand query model projections, 250–252
Onion Architecture, 94, 95
open-source schema registries, 356
optimistic concurrency, 133–135, 176, 230–231, 234, 237, 243, 245, 299, 302
Optimistic UI pattern, 203
Orchestration of processes, 290–304, 307–308, 316; *See also* stateful process coordination
orchestrators, 290, 291, 308, 402
order-independent processing, 67
order of messages
  causation IDs for, 306
  delivery retries and, 72
  in distributed message logs, 245
  in event streams, 225, 226–227, 235–236, 259
  in global message journals, 361
  and message consumption, 28
  overview of, 65–67
  in PostgreSQL event stores, 235–242
  in publishing state events, 376, 379
  in reactive user interfaces, 424, 428–429
  relaxed global ordering, 238, 241–242
  topic partitioning and, 29
outbox, transactional, 61–65, 141, 143

P

Palermo, Jeffrey, 94
parallel command execution, 279, 285
parallelism vs. concurrency, 133
partitions, 29, 67, 245, 257, 261, 372–375, 379, 388
passive-aggressive commands, 43
Payment Context, 275–277
pending transactions, 157–159, 162, 217–220, 250
performance
  batch processing and, 145
  binary format for, 49
  consistency delays, 359–360
  and data caching, 181–183
  data composition and, 390
  delivery retries and, 72
  in event-based data synchronization, 187
  global ordering and, 240, 242
  and human interaction, 272
  persistence and, 153
  of processes, 271–272, 277, 280, 317
  publishing latency, 64
  of query model projections, 256
  scoping message order and, 67
persistence
  aggregates and, 150, 153
  for atomic transactions, 271–272
  of command/query model data, 184–186, 192–193, 204, 246
  and CQRS use, 175
  dependency inversion for, 84–85
  of event-sourced state, 223
  of invalid event sequence, 230
  and message delivery guarantees, 58, 60
  in message journals, 142–144, 367
  modifying persistent messages, 344–345
  product-specific behavior and, 82, 83
  via relational data storage, 127
  reliable message publishing and, 61
  repositories for, 126, 184
  and stateful process orchestration, 298–299
pessimistic concurrency, 133, 230
pivot transactions, 273, 274
plain HTTP, 407–409, 411, 416, 418
polling, 63, 64–65, 144, 145, 232, 240, 241, 407–409, 411, 418
ports, 89–92, 94
Ports and Adapters architecture, 80, 89–95
postal addresses, 338, 342, 345
PostgreSQL
  Apicurio Registry for, 356
  data interpretation in, 128
  event journals, 144, 362
  for event stores, 234–242, 246, 344–345
  KurrentDB vs., 245
  message relays in, 64

repositories, 129, 134–135
primary actors, 89–92
primary adapters, 89–92, 95, 101, 145
primary ports, 89–92
privacy, data deletion and, 265, 351, 369, 370–371
privacy, query model replication and, 370
private events, 262–263, 265
processes. *See* long-running processes
producers, 8, 13, 28, 321, 322, 323, 328, 329, 340, 346, 354, 355, 356, 375, 389
product(s)
behavior and data in software, 208
change/innovation of, 5
context maps for, 21–22
core functionality of, 14
defined, 1
domain models for, 3, 15, 310
need for data composition in, 359
problem solving via, 13–14
processes in, 269, 273, 274, 279
pros/cons of CQRS for, 167
role of UI in, 401
value generated by, 1
product-specific behavior, 3, 72, 82, 83, 213, 359, 361, 390
projections, query model. *See* query model projections
property injection, 87
Protobuf (Protocol Buffers)
field names in, 333
JSON Schema vs., 52, 329
KurrentDB with, 243
limited validation with, 52
schemas, 52, 356
serialization via, 49–51
protoc, 50, 52
publishing
of events, 60–61, 136, 141–145, 223, 261–265
filter mechanisms for, 262–263
of four message categories, 48
message ordering and, 66
publisher confirmations, 58
publishers/subscribers, 8
publishing latency, 64
in reactive user interfaces, 401, 424–428
reliable message publishing, 60–65
republishing event journal, 200
of state events, 376–385
transactional outbox for, 61–65
transactional publishing for processes, 300–302
when applying Orchestration, 292
Publish–Subscribe communication, 8, 27–28, 30, 432
pure domain storytelling, 16–17
purpose of messages, 322–324, 329, 332, 338

Q

quality of service (QoS) levels, 414, 415
queries
commands and, 41
CQS principle for, 170
defined, 19, 40, 41, 170
evolution of, 326–327
for feedback/reviews, 169
implementation for handling, 45–48
in message-driven systems, 173–174
for one-to-one communication, 321
Orchestration of, 290, 291
in reaction to messages, 39
repositories for custom, 131–133
responses to, 41
specialized query data, 167
storage and performance of, 128
UI execution of, 418–423
use-case-specific, 2
query model projections, 246–265, 346, 361, 432–433
query model replication, 370–389
query side in CQRS
architecture/function of, 165–166
on-demand computation on, 180–183
event-based synchronization on, 187–190, 226
Event Sourcing use for, 209
event store on, 223
initialization/evolution of, 195–201, 246
multiple data sources for, 248
query model overview, 175, 178–180
reactive stream consumption on, 232–233
rebuilding query model, 256
synchronous command/query updates, 183–186
technical segregation for, 190–195

technologies specific to, 167
and use case segregation, 171–173
queues for messages
as bound to exchanges, 30
and brokers with PubSub, 27–28
dead-letter queue, 73, 75
identifiers for de-duplication in, 37
for incoming messages, 34
input and output, 43
post-data migration, 316
problematic message queues, 72
in server/client exchanges, 411
quorum queues, 74

R

RabbitMQ
channels vs. TCP connections for, 32
in client messaging, 416, 418
error handling in, 74–75
Kafka receivers vs., 386
mapping messages to topics in, 43
as message broker, 28, 29, 30–33
message categories in, 43
message logs with, 361, 362
message relay to, 144
publisher confirmations with, 59
for reactive UIs, 429, 430
record updates in, 344
to transfer query model state, 374
RabbitMQ Streams, 29, 33, 344, 361, 374
React, 404, 405
Reactive Architecture (Reactive Systems), 10
reactive query model projections, 253–255, 259, 346, 368
reactive query models in the UI, 428–432
reactive stream consumption, 232–233, 240–241, 246
reactive synchronization, 190, 207
reactive user interfaces, 401, 423
read model. *See* query side in CQRS
readonly modifier, 114
read/write models, 176
read/write ratio, 167
rebuilding query model, 195–201
receive() function, 31, 75
registration, duplicate, 68
registries, schema, 354–357
relational database management system (RDBMS), 234
relational data storage, 127–129, 153
relaxed global ordering, 238, 241–242
relaying messages, 64, 143–145, 240, 292, 302, 376, 378
reliable message publishing, 57–65
Remote Procedure Calls (RPCs), 5
replication of query models, 370–389
repositories
aggregate persistence in, 150
aggregate/repository relationship, 153
command model, 176
for long-running processes, 298–304
overview of, 126–136
vs. stores, 184
version control repositories, 354–355
Request–Response communication
for data composition, 360, 389
domain-specific errors in, 280
at Infrastructure Layer, 104
loose coupling in, 9
mapping categories in, 43
mapping messages to topics in, 43
as message purpose, 322, 324
and message scope, 325
routing in, 27, 30
for server/client messaging, 406–418
synchronous, 34
term defined, 8
timeouts in, 393
responses
to commands, 40, 41
commands/queries to produce, 40
domain-specific data in, 41
evolution of, 326–327
implementation for handling, 45–48
at Infrastructure Layer, 103–104
naming, 41
for one-to-one communication, 321
to queries, 41
term defined, 41
responsibilities
for choreography, 284
of each architectural layer, 95–106
of entities, 121
of event-sourced command model, 215–216

layers to delineate, 80, 81
separation of. *see* CQRS (Command-Query Responsibility Segregation)
in state event publishing, 380
retries, automatic, 35, 72–75, 135–136, 229, 231, 259, 273, 274, 281, 284, 292, 298, 315, 378, 396–397, 398
retroactive query model state transfers, 370
reversals, process, 273–274, 275–277, 280, 281, 283, 288–289, 302, 316
RNG (random number generation), 116
root entities, 147, 148, 154, 176, 215, 222, 248
routing of messages
in the Application Layer, 99
content- and topic-based, 27, 28
in process Orchestration, 291–292, 308, 309
route() method, 99, 332
routing keys, 30, 43
technologies for complex, 28
type-based, 99
in the UI, 402
runtime services
architecture for, 81–82
backend runtime, 11, 14, 190, 195
coarse- and fine-grained, 11
of command/query sides, 192–193
and CQRS, 190
infrastructure separation from, 94
JSON Schema as, 52
loose coupling of, 34
message-driven architecture for, 10, 12
and message evolution, 327–328
modular monoliths vs. separate, 12
schema registries as, 356
and system architecture, 79

S

sagas. *See* long-running processes
saving entities, 129
SBE (Simple Binary Encoding), 50
scalability
and concurrency control, 64
via CQRS, 2, 194, 205
and eventual consistency, 156
of Kafka products, 245
loose coupling for independent, 10
persistence and, 153
of processes, 317
Reactive Architecture for, 10
for remote work, 5
scoping message order for, 67
of separate parts, 9
separating runtime services for, 194
scalar type, 49, 50
Schema Registry from Confluent, 356
schemas
event logs and evolution of, 216
explicit, 127
global message journal, 367, 368
and message evolution, 327–328
overview of, 51–53, 57
PostgreSQL event store schema, 235
Protobuf for, 49
schema registries, 354–357
secondary actors, 89–92
secondary adapters, 89–92, 95, 101
secondary ports, 89–92, 96, 97
security, 37, 100–101, 265, 414
segregated event layers, 140, 263–264, 301
send() method, 31
serialization
for client communication, 407
for event stores, 223–224
overview of, 48–52
text-based and binary formats for, 48–51
server/client communication, 406–418
Server-Sent Events (SSE), 409–411, 413, 414, 432, 433
Service Layer, 94
Service Layer with Domain Model, 93, 95
session tokens, 37
side-effect-free behavior, 115, 219, 249, 250, 404
Simple Text Oriented Messaging Protocol (STOMP), 414
single logical transactions, 271, 277, 279
Single Message Transformations (SMTs), 381, 382, 384
Single Responsibility Principle (SRP), 102
software
anti-patterns in, 38, 87, 131, 139, 204, 227
behavior and state in, 110
continuous evolution of, 321
demand for high-level, 2
domain models in, 15, 109, 310
essential aspects of, 147
layers for complex, 80
loose coupling for, 9

maintaining quality of, 5, 23
monolithic, 11
processes in, 269, 273, 274, 279
when to apply CQRS to, 166–168
software development
bidirectional identifiers in, 123
building blocks for, 109
business mindset for, 1, 2, 22
domain modeling in, 15–22
evolution of bounded contexts in, 22
lowering cognitive load in, 81
naming tips for, 37
Reactive Architecture for, 10
solving customer issues via, 1, 2
technical capability for, 25
use-case-specificity in, 25
specialized query data, 167, 177, 178, 181, 183, 266, 270, 385, 389
splitting entity types, 347
splitting message types, 334–336
SQL, 15, 62, 127, 196, 197, 361
state (fat) events, 42, 188
state-changing actions
aggregating, 147
atomic transitions, 150
command model for, 170
commands to execute, 170, 175, 176
CQRS for, 2
for entities, 121–123
and Event Sourcing, 210–211
and order-independent processing, 67
processes as, 270
by process orchestrator, 297
reliable publishing of, 60–61
and value object immutability, 115
state event publishing, 376–385
stateful process coordination, 270, 285, 290, 291, 295–304, 306, 310, 314, 315, 316, 319
stateless computation, 98, 345
stateless Orchestration, 292–295, 310
state modeled as events, 211–215
state synchronization via events, 155–156, 158
STOMP (Simple Text Oriented Messaging Protocol), 28
storage
binary format for, 49
of current states only, 209
data caching, 181–183
in documents vs. relational data, 127–129
message persistence in, 58, 60
post-delivery deletion from, 60
for query model data, 246–247
of sensitive information, 266
stores, repositories vs., 184
stories, domain, 16–17
strong consistency, 150
structure, message, 36–39
subordinate elements. *See* child/subordinate elements
synchronization of data in CQRS, 180–190, 201, 204, 246, 257
synchronous communication, 34–36, 76, 293, 294, 360
synchronous data updates, 183–186, 190, 192, 201, 232, 252
synchronous on-demand data composition, 389–398, 399, 420
synchronous queries, 39
synchronous query model projections, 252

T

tables
creating PostgreSQL, 44
custom event store tables, 234–235
in document storage, 128
for event journals, 141
persistence of aggregates in, 153
transactional outbox, 62
tactical building blocks
aggregates, 147–162
domain events, 136–146
domain services, 124–126
entities, 119–123
identities, 37, 116–119
repositories, 126–136
value objects, 110–115
temporal coupling, 292, 294, 390
temporal decoupling, 9, 26, 30, 39, 187, 359, 360, 367, 399
testing, 81, 88, 95
text-based serialization formats, 48–49
threading, 11, 133, 240, 257
tight coupling
of command/query sides, 187, 190

and on-demand computation, 183, 190
reactive queries by receivers and, 39
in reactive UIs, 424
in synchronous exchanges, 35
timeouts, 75, 183, 302, 393–395, 396, 398
time-related information, 37, 65, 111–112, 115, 116
tolerant structure and consumption, 338–340
tombstone messages, 383, 389
traces, 306, 307, 319
tracking in message journals, 141, 143
transactional boundaries, 65, 136, 147, 148, 150, 153, 154, 155, 157, 167, 173, 175, 204, 207, 211, 222, 227, 257, 266, 308, 310, 326
transactional consistency, 124, 147–162, 176, 180, 222, 274, 359
transactional outbox, 61–65, 141, 143
transient messages, 322, 338
translation, message, 291–292, 308, 309
TSyringe, 86, 87, 106
2PC (two-phase commits), 271
TypeScript
compilation tools, 50
as CQRS runtime service, 195
dependency inversion with, 106
dynamic classes in, 55
generating JSON Schema from, 329–331, 340
immutability in, 114
vs. JavaScript, 54, 88
limits and TSyringe, 87
message type factory for, 56, 57
representing identities in, 117–118
structural variance in, 339
for tactical building blocks, 110
updating field data types with, 333

U

UI composition, 402
union types, 213, 229, 329, 330, 339, 340
unique identifiers, 116
upcasting, 342–344
updates
concurrency and competing, 133, 135
continuous, local, 217
event-based, 187–189
idempotent, 155, 187
synchronous command/query side, 183–186
UI progress indicators, 201–202
use case segregation, 165, 166, 169, 171, 174, 175, 190, 203
Use Cases layer, 95, 97
use-case-specificity
architectural layers for, 94
for bounded contexts, 4
and CQRS use, 2, 178, 190
for custom queries, 131–132
and idempotent behavior, 69
and publishing state events, 383
in query model, 178, 195, 361
in repositories, 126, 131–132
user experience (UX), 2, 14, 19, 153, 274, 401
user interface (UI)
and architectural layers, 82
building reactive query models, 428–432
client-side query model projections, 432–433
concurrency error reporting in, 135
dedicated layer for, 95
events in, 207
eventual consistency in, 201–203
executing commands/queries in, 418–423
frontend architecture, 402–406
importance of, 401
infrastructure separation from, 94
message-driven, 105, 401
messaging protocols/technologies for, 406–418
reactive UIs, 423, 424–438
User Interface Layer, 96, 105–106
UUID (Universally Unique Identifier), 37, 56, 116–117, 118, 275, 296, 303, 350
UUID version 4, 37, 116, 349

V

validation
errors, 71
implementation design for, 329–332
invariants vs., 152
for message structure/type, 48

schema registry for runtime, 354
schemas and, 52–54, 57
value objects
in command model, 176
entities as utilizing, 121, 123
identities as, 117–118
overview of, 110–115
for user reviews, 168–169
Vernon, Vaughn, 93, 119
versioning
for concurrency control, 133–136
to establish message order, 65
explicit, 340–342
for processes, 316–317
version control repositories, 354–355
version number formats, 341

W

WebSockets, 412–414, 415, 416, 418
WHERE clauses, 132, 135
Woolf, Bobby, 7
write model, 176; *See also* command side in CQRS
write/read ratio, 167

Y

Young, Greg, 165

Z

ZeroMQ, 30