

Courtesy of



redis

Redis Microservices

for
dummies[®]
A Wiley Brand



Learn about
messaging with Redis

Discover microservices
architectural concepts

Build high-performance
services with Redis

2nd Limited Edition


Kyle Davis
with Loris Cro

About Redis

Data is the lifeline of every business, and Redis helps organizations reimagine how quickly they can process, analyze, make predictions with, and take action on the data they generate. As the home of Redis, the most popular open source database, we provide a competitive edge to global businesses with Redis Enterprise, which delivers superior performance, unmatched reliability, and the best total cost of ownership. Redis Enterprise allows teams to build performance, scalability, security, and growth into their applications. Designed for the cloud-native world, Redis Enterprise uniquely unifies data across hybrid, multi-cloud, and global applications, to maximize your business potential.

Learn how Redis can give you this edge at redis.com.

With more than 48,000 GitHub stars, 19,000 forks, and 430+ contributors, Redis is an incredibly popular open source project supported by a vibrant community. Redis has been voted the most loved database, rated the most popular database container, the #1 cloud database and the #1 NoSQL in software stacks.

 redis.com

 [@redisinc](https://twitter.com/redisinc)

 <https://www.linkedin.com/company/redisinc/>

 <http://www.youtube.com/c/Redisinc>

 <https://developer.redis.com/>

Learning Redis? Check out Redis University

Redis University was established with the goal to create a destination for all things Redis. Created and delivered by core Redis developers, practitioners and experts, Redis University provides a variety of courses for developers & administrators. Online courses provide an immersive experience, with guided labs and experiments for practitioners to sharpen their skills and deepen their knowledge and insights.

 university.redis.com/

Redis Microservices

**for
dummies[®]**
A Wiley Brand



Redis Microservices

2nd Limited Edition

by **Kyle Davis with Loris Cro**

for
dummies[®]
A Wiley Brand

Redis Microservices For Dummies®, 2nd Limited Edition

Published by

John Wiley & Sons, Inc.

111 River St.

Hoboken, NJ 07030-5774

www.wiley.com

Copyright © 2022 by John Wiley & Sons, Inc., Hoboken, New Jersey

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Trademarks: Wiley, For Dummies, the Dummies Man logo, The Dummies Way, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

ISBN 978-1-119-82429-9 (pbk); ISBN 978-1-119-82430-5 (ebk)

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact info@dummies.biz, or visit www.wiley.com/go/custompub. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Editor: Elizabeth Kuball

Acquisitions Editor: Ashley Coffey

Editorial Manager: Rev Mengle

Business Development

Representative: Matt Cox

Production Editor:

Tamilmani Varadharaj

Special Help: Steve Suehring

Table of Contents

INTRODUCTION	1
About This Book	1
Foolish Assumptions	1
Icons Used in This Book.....	2
Beyond This Book.....	2
CHAPTER 1: What Is a Microservices Architecture?.....	3
Defining Microservices Architecture	3
Knowing Why You Would Use Microservices	4
Starting from Zero or Breaking the Monolith	5
Understanding Where Microservices Should Be Used	6
Exploring How Redis Fits with Microservices	7
Using Redis for Messaging	8
Pub/sub	9
Redis Streams.....	9
Redis Lists	9
Using Redis for Storage	10
Hashes.....	10
Sorted sets	10
Search.....	11
Graph.....	11
JSON	11
Using Redis for Caching.....	11
Describing a Redis-Powered Microservices Architecture	12
CHAPTER 2: Microservices Communication Patterns.....	15
Defining a Stateless Service	15
Knowing Where to Break the Monolith	16
Getting Services Talking.....	18
Having a Conversation with a Service.....	19
Normal order.....	19
Insufficient funds	20
Unshippable product.....	21

CHAPTER 3:	Distributed State with Microservices	23
	Defining Distributed State	23
	Discovering the Needs of Distributed State	24
	Exploring Data Types	25
	Pub/sub	25
	Lists	26
	Streams	26
	Publish/Subscribe or Logged Events	27
	Getting Data across Multiple Services	27
	Clusters, Multi-Tenancy, and Redis Enterprise	30
	Nodes and shards	31
	The cluster and databases	32
CHAPTER 4:	Active/Active and Microservices	35
	What Are CRDTs, and What Is Active/Active?	35
	Synchronizing Data across Clusters	37
	Understanding How Data Changes	38
	Fitting Active/Active into a Microservices Architecture	40
	Primary data storage with Active/Active	41
	Caching with Active/Active	42
	Scaling writes with Active/Active	43
	Knowing Where to Use (and Not Use) Active/Active	45
CHAPTER 5:	Building a Service	47
	Getting Clear on What This Service Does	47
	Understanding the Language and Setup	48
	Processing Events	50
	Lending books	50
	Returning books	52
	Invoking the Service	53
CHAPTER 6:	Ten Key Microservices Takeaways	55
APPENDIX:	Advanced Microservices Design Patterns	57
	Defining Microservices	57
	Reviewing Design Patterns	57

Introduction

The microservices architecture is a modern way of developing applications. The goal is to reduce development complexity, ease deployment headaches, reinforce resiliency, and increase release velocity.

Redis is a highly popular, open-source, widely used NoSQL in-memory database that focuses on high-performance use cases. Redis is a core component of many architectures, including microservices. Redis Enterprise extends the capabilities of Redis by providing enterprise-grade operational functionality to Redis without affecting compatibility with open-source Redis.

About This Book

This book is a starting point for anyone considering implementing the microservices architecture and who already has a rough idea of what Redis does in a modern application.

The book covers what an application developer or architect may need to understand about implementing a service in the microservices architecture. If you're a manager, you'll get a high-level overview of the concepts the architecture provides. If you're in Ops or DevOps, you'll get a basic understanding of how Redis and Redis Enterprise operate in the context of scale and performance.

You can read the book from cover to cover, but each chapter is self-contained, so you can dip in and out based on your interests and needs.

Foolish Assumptions

While writing this book, we assumed you have a general understanding of databases, performance, and what Redis is from a basic level. We mention the moving parts of building an application, but we assume you know how things connect over Transmission Control Protocol/Internet Protocol (TCP/IP), have a basic understanding of cloud technology, and also have a fundamental

understanding of networking time scale (nanoseconds, milliseconds, seconds) and data sizes (gigabytes, megabytes, kilobytes, bytes).

If you're a developer, you should have a local development environment available to run the code in Chapter 5. We also use GitHub to host the example code, so you'll need to be familiar with downloading repositories and following basic instructions in the installation README file.

Icons Used in This Book

Throughout the book, we occasionally use special icons to call attention to important information. Here's what to expect:



REMEMBER

The Remember icon points out where we're providing a friendly reminder or giving information you may want to commit to memory.



TECHNICAL
STUFF

Material marked with the Technical Stuff icon explains any jargon or technical processes in plain language.



TIP

Look for the Tip icon to find useful nuggets of information and helpful advice.



WARNING

Anything marked with the Warning icon will help you avoid frustrating mistakes.

Beyond This Book

Microservices Architecture, Redis, and Redis Enterprise are approachable topics but have a lot of depth. This book serves as an introduction to the concepts and the software, but there's a lot to learn. For more information, please visit <https://redis.com/microservices>.

IN THIS CHAPTER

- » Diving into what microservices architecture is
- » Discovering why you'd use services and where to use them
- » Starting from zero or breaking the monolith
- » Using Redis for messaging, storage, and caching
- » Exploring a Redis-powered microservices architecture

Chapter 1

What Is a Microservices Architecture?

Say the word *microservice* in a crowded office, and you're likely to see developers rip earbuds out of their ears and look in your direction. The problem is that, although it's not an uncommon term, the commonly held definitions are often loose, and the concept is poorly understood.

Defining Microservices Architecture

Reading the words *microservices architecture*, you may have an intuitive idea of what those words mean: small services in a computing architecture. And you're not *wrong*, but you're also not completely correct.

In your reading on the subject, you may have seen the associated phrase *breaking the monolith*. If you're anything like me, you think of *2001: A Space Odyssey* and that great scene with the apes. Sadly, it has nothing to do with some mysterious black obelisk,

but rather the concept of breaking (or *decomposing*) a single, large program into pieces.

So, thinking about these meanings, you know you'll make small programs (*microservices*) from a whole. The program still needs to behave like a single entity at times; at other times, it needs to have the properties of smaller programs.

A good metaphor might be a heating, ventilation, and air conditioning (HVAC) system in your home. This system may consist of a furnace, air conditioner, humidifier, thermostat, and fresh-air exchanger. When you're in your house, you never go to the humidifier or furnace and flip a switch to turn it on; instead, you make adjustments on the thermostat. Each individual system comes in a box from a manufacturer, and they're all connected, yet each acts as a single unit. You can swap the gas furnace out for an electric model, and the operation stays the same for you and your family. When you adjust the thermostat in your house, you don't need to know what kind of furnace or air conditioner you have; you just know your thermostat. In this metaphor, each HVAC component is a service, and the whole design of your HVAC system is your microservices architecture.



TECHNICAL
STUFF

We use the term *service* instead of *microservice* to describe individual programs. Your service (not microservice) is part of a microservices architecture.

An HVAC system in a house is something that is changed maybe once a decade, but software architectures are far more dynamic. Cloud instances go up and down and new versions are deployed sometimes many times per day. Plus, HVAC systems have very simplistic inputs and a small number of connections. A microservices architecture may have thousands (millions even!) of inputs that may route to do many complex activities, not just regulate the climate of your house.

Knowing Why You Would Use Microservices

With many moving parts (also known as *services*), there's extra complexity to building this type of architecture. Complexity is generally considered to be the primary foe of building good software, so why introduce more?

The problem lies in the alternative. In a monolithic architecture, everything is contained in one large unit. When something breaks, as even the best software sometimes does, it breaks as a whole. Even if you started out with a lean project, over time, software tends to grow in complexity, as you add more and more features and work-arounds until you have a hulking, unwieldy monster. Updates and releases become slow and painful. A service in a microservices architecture is smaller and concerns itself only with minimal responsibilities, bounding the overall complexity. If a single service becomes overly complex, rewriting that single service is easier than having to rewrite and merge in something to a monolith.

When you try to scale a monolith, problems tend to arise. As an example, often one part of the monolith is particularly resource intensive. You have no options to scale an individual part of the monolith; your only option is to create replicas of the whole system, which is wasteful of resources. In the worst cases, monoliths aren't built for replication, leaving hard limits on scale.



REMEMBER

Services are simply easier to build and manage than monoliths. The microservices architecture isolates complexity, allowing for smaller, more agile teams to create a service.



TECHNICAL
STUFF

You may have heard about “two-pizza teams” before, where the entire team — leaders and all — can be fed with only two pizza pies. The microservices architecture lends itself to being built by many, smaller teams, so keep the pizza guy on speed dial.

Finally, a microservices architecture is flexible. Individual services allow for a variety of platforms, languages, and tools to be used because these choices affect only a small team at one time. You can rapidly make changes and build and release updates, and as long as the inputs and outputs aren't affected, developers can move fast *without* breaking things.

Starting from Zero or Breaking the Monolith

In the previous sections, we discuss “breaking the monolith,” but the microservices architecture is not just reparative. It's equally as useful, if not more so, as an option for greenfield projects.

When breaking down (decomposing) a monolithic architecture, you have to worry about many factors:

- » **Gradualizing into services:** Moving from a single deliverable to many small services is a radical shift and requires careful planning.
- » **Splitting at the right places:** When breaking a monolith, you have to take a step back and really think about where the most logical places are to divide responsibilities into services. At the same time, you want to avoid getting stuck in how your current application is built; instead, rethink without the technical debt of the past.
- » **Transitioning a team:** Making changes to monoliths is really easy — perhaps too easy — in code. While you may need to touch dozens of function calls to deploy a single feature, your team can do this easily (testing and deploying is another issue). Isolating teams to build services also isolates teams from making changes in places for which they aren't responsible. A double-edged sword!

If you're starting a truly *greenfield project* (a new project with no legacy code or architecture), you sidestep many of these transition issues. So, if you're moving to a microservices architecture, or if you're starting with a clean sheet, you gain tons of advantages. However, you have to think more carefully about your strategy for a transition from a monolithic architecture to a microservices architecture.

Understanding Where Microservices Should Be Used

Using a microservices architecture is a viable strategy, but it isn't always right for every situation. Let's examine a situation where microservices would work:

- » **The code base is (or will be) large.** A small code base will probably not benefit from splitting up into logical services.
- » **You have an adequate staff to split into teams devoted to particular services.** Many advantages are lost if the whole team works on one service at a time.

- » **The operational team is ready and willing to support the many services in the architecture.** Although the long-term operational benefits of a microservices architecture are well understood, at the start it can seem like a lot more work to run many servers or instances than a single large one.
- » **The underlying business processes are well defined.** One of the cool things about a microservices architecture is that anyone with a knowledge of the business can look at an architecture diagram and see the mapping. If, however, the business processes are ad hoc or poorly delineated, then your architecture will reflect this messiness.

If all the preceding statements are true, you may be a good candidate for a microservices architecture. That is not definite — there are innumerable other reasons why you may not be ready for a microservices architecture. On the flip side, you could be “false” on a few points, but it’s the right time to press forward. Keep your critical thinking hat on and truly consider if it’s right for your organization and project.

Exploring How Redis Fits with Microservices

Redis evolved differently than many other popular database systems. Many of the most popular database systems were developed in an era when a company adopted a single database across the entire enterprise. The single database system would run all the functions of the enterprise, storing and running it all in one place. You can probably picture this — a room full of refrigerator-size machines, many having reel-to-reel tape drives. Of course, this is an image best left to nostalgia and has no relationship to modern computing infrastructure or hardware.

If that mental image is so far removed from today’s reality, why are databases from that era still in use? The answer to that question is probably more philosophical than this book should be, but, in short, databases are important and not often changed. These considerations, paired with the comfort of familiar software, result in a powerful resistance to alter the status quo.

Redis, on the other hand, was not built with this notion in mind. It was originally designed to solve a particular problem and to be as small and fast as possible. Redis was built in the NoSQL era where the “shape” and use of the data was first considered; a database was then matched with these factors.

Configurability is at the heart of Redis: In this way, databases can be configured in opposed ways. Redis can serve as a completely ephemeral layer that automatically manages eviction of data as limits are reached. This configuration is often used as a caching layer. Alternatively, Redis can be configured to be durable and to persist keys until they’re explicitly removed. Use cases such as messaging/stream storage or fast primary databases rely on this second configuration.



REMEMBER

A microservices architecture has the same type of goals in mind: Your service is designed to fit a particular use — you’re not running everything in the business.

Redis is a very flexible and versatile database designed not to store massive amounts of data that will be mostly idle. Redis is designed to store active data that will change and move often with an indefinite structure with no concept of relations. A Redis database has a small footprint and can serve massive throughput even with minimal resources. An individual service in a microservices architecture concerns itself only with input and output and data private to that service. This means that Redis databases can back a wide range of different microservices, each with its own individual data store.

The very nature of having many services means each service must perform as fast as possible to make up for the connection and latency overhead introduced by interservice communication.

Using Redis for Messaging

Metaphorically, databases — and, by extension, Redis — are thought of as a box in which you put things, but this is not the only case for Redis. Redis is quite at home passing messages between services, either with storing them for later use or without.

Pub/sub

The first (and oldest) messaging capability of Redis is pub/sub. This pattern allows for *publication* of messages to channels where other services can *subscribe* to these messages. Both publishing and subscribing in Redis are very lightweight and can be executed usually in a fraction of a millisecond. The pub/sub system is fire and forget, so it's not right for every messaging situation, but it's extremely useful to *notify* a service to check for something else.



TECHNICAL
STUFF

Fire and forget is a way of communicating that doesn't retain the sent message. So, if a channel has no subscribers, the message is instantly lost. Likewise, if a subscriber goes down for a few moments, the service can't look back and see what it missed when it comes back online.

Redis Streams

Redis added a new data type in version 5 that allows for a *stream* of timestamp-ordered key/value pairs in a single key. Inspired in part by the Kafka message system, this data type was designed specifically to address situations where there are producers and consumers (or groups of consumers) that may or may not always be available.

Part of the power of this data type is that a consumer can wait for messages ("block") to come in and pick up where it left off if it goes offline (unlike pub/sub). Given the structure and facilities packed into the commands, it's no surprise this plays a large role in connecting services.



WARNING

People have thought of Redis Streams as a replacement for pub/sub. This is, however, not the case. Pub/sub and streams are just different — each having a unique set of uses and advantages over the other.

Redis Lists

Redis Lists are doubly linked lists of elements (strings) stored at a single key. Like Streams, Lists can wait for new elements. Lists are a great way of representing a first-in, first-out (FIFO) or queue. Lists can also be rotated or atomically transferred from one key to another, easily creating a queue with an additional pending queue.



Atomicity is a database concept meaning that two or more actions occur in a way that are incapable of being interrupted. As a consequence, you can guarantee the data changes predictably inside an atomic action.

Using Redis for Storage

Redis was initially based on the concept of a key/value store, a type of database that can address each piece of data by a unique string. Redis works great in a key/value capacity, but the additional data types both built into Redis and provided by modules give it a much richer ability to consolidate and find data than a strict key/value store does. In the following sections, we describe a few (but not all) of the storage data types in Redis.

Hashes

Redis Hashes are much like a key/value store inside a key/value store. A single hash is referred to by a key that contains any number of fields and values. As an example, you may have a user with a key of `user:1234` and the fields of `username`, `location`, and `age` with each field having its own value.

Some have compared it to a single row in a table, although this is probably not a very useful comparison. Redis enforces no connection between different hashes (unlike rows in a table), and there's no schema (unlike the columns in a table). Hashes are often used to represent single-depth objects or structs.

Sorted sets

Sorted sets in Redis are sets with scores, or intrinsic numeric sorting values, for each member (represented by a string). This allows for Redis to easily retrieve members between given scores or at the top or bottom of the score range. *Note:* This data type still has the properties of a set, chiefly that members cannot be repeated. Trying to add a repeated member will update the score, rather than insert a second one.

As an example, you may have items in an e-commerce store, with the score being the price and the member being a unique product ID; listing the items by ascending or descending price could use the same structure.

REDIS MODULES

Search, Graph, and JSON are all implemented as Redis Modules. Redis Modules are extensions to the core data types and commands of Redis that operate from within Redis and allow for new and extended functionality. Modules have direct access to memory and CPU resources, so they're on par with the in-built commands. The software development kit for modules is available in a variety of systems programming languages, so you can even write your own.

Search

RediSearch, a Redis module, accommodates full-text search. This allows Redis to store documents and search across multiple fields and types. Like hashes, each document is a series of fields and values. Unlike hashes, all the documents are bound together into an index, and individual fields can be described in a schema that makes them queryable via a special query language.

Graph

RedisGraph, also a module, is designed to store nodes and ad hoc relationships. Each node contains a single-depth series of attributes; those nodes can be connected to each other through relationships that can also have attributes. Collectively, the nodes and relationships are referred to as a graph. The graph can be queried with the Cypher query language.

JSON

JSON documents can be stored in Redis using the RedisJSON module. This module allows for the storage of complex, deeply nested JSON documents each at its own key. By specifying a path, you can retrieve parts of a JSON document very precisely, allowing for data buried deeply within the structure to be retrieved — without reading the whole thing — and it will be updated atomically.

Using Redis for Caching

Redis is a very fast database that runs entirely in-memory. As a consequence, it can be placed in front of your existing disk-based database or to prevent an expensive or time-consuming

application programming interface (API) call. Redis has built-in timers that remove data at keys after a specified time period (*time to live*, or TTL) as well as very efficient key presence checking. These two features allow your application to make a quick check in Redis before going out and making a slower or more expensive call. You can use TTL to ensure you're not returning stale data. Additionally, you can specify database-wide eviction policies that ensure that you're optimally using your memory.

Redis Enterprise also enables you to extend your random access memory (RAM) into flash memory with Redis on Flash. Doing so places the most-used pieces of data in the fastest storage and least-used into flash (solid-state drive, or SSD) storage. By tiering the storage between RAM and flash, you optimize the use of the more-expensive RAM and the less-expensive flash memory without having to evict for resource constraints.

Describing a Redis-Powered Microservices Architecture

A key component of the microservices architecture is that each individual service stands on its own. The service is not tightly coupled with another service. Going down one more level, this means that services must maintain their own states, and to maintain a state you need a database. Because services can be numerous in an architecture, overhead is the enemy of scale — services that rely on infrastructure that itself needs lots of resources simply to run don't make much sense.

In an ideal situation, the service data is completely isolated from other data layers, which allows for uncoupled scaling and cross-service slow resource contention. Services are specifically designed to fill one role (business-process-wise), so the state they store is inherently nonrelational and well suited to NoSQL data models. Although saying that Redis is a blanket solution for all data storage in a microservices architecture is unfair, it certainly fits well with many of the requirements.

After you build your service, the service needs to talk to other services. In a traditional microservice environment, this occurs over private HTTP endpoints using REST or similar conventions. After

a request is received, the service begins processing the request. During the processing, occasionally a tricky problem occurs: What happens when you want to transactionally do processing across multiple services? You can't solely rely on layers of HTTP requests, because failures along the way result in partially applied transactions. The saga pattern can be used to solve the problem of transactions across services, and Redis Streams can be used to implement sagas.



TECHNICAL
STUFF

The saga pattern is not a new idea. It was defined in 1987 for long-lived transactions when something was prone to latency because a human was involved in the transaction. Sagas take their name from the Old Norse word for a narrative — it's a story of what your data does in the transaction that can be written and retold.

The HTTP approach works and is widely used; however, an alternate method of communicating is available in which services write to and read from loglike structures (in this case, Redis Streams). This allows for a completely asynchronous pattern where every service announces events on its own stream and listens only to streams belonging to services it's interested in. Bidirectional communication at that point is achieved by two services observing each other's streams.



WARNING

Isolating the data from one service to another is an important part of the microservices architecture. If you're using Redis Streams as a communication method or sagas as well as storing your data in Redis, they should be used in different instances.

Even in services that don't use Redis for storage, communication, or sagas, Redis plays a vital role. To deliver a low-latency final response, it's critical that each individual service responds as fast as possible to its own requests, often outside the performance threshold of traditional databases. Redis, in this case, plays the role of the cache, where the teams that developed the microservice decide where data is not always required to be retrieved directly from the primary database, but instead can be pulled from Redis at a much faster rate. Additionally, any external data services that need to be accessed through an API will likely be far too slow for a reasonable response time — again Redis is used in this case to prevent unneeded and lengthy (or potentially costly) calls from impacting the overall performance.

- » Defining the stateless service
- » Breaking up the monolith
- » Eavesdropping on services' conversations

Chapter 2

Microservices Communication Patterns

A service in the microservices architecture can't really do anything useful on its own. Indeed, in the microservices architecture, the communication between services actually creates the utility. This chapter introduces the patterns that are available to services using Redis, which provides the plane in which services can communicate and create utility.

Defining a Stateless Service

A few years ago, when you joined a company, you may have been introduced to the servers you would be working with — this was almost like a personal introduction. You knew and understood their names, their jobs, where they lived, and sometimes even how old they were and which of them had quirks. Some servers

even had personalities and temperaments. We recall an exchange that went something like this:

Coworker #1: Production is down! Does anyone know anything?

Coworker #2: Let me check. Yep. It's Titan having a fit again. Titan always does this at the worst time. Let me talk nice to Titan and fix it.

Hearing Coworker #2's response alone, you could think that Titan was a dog or some other living being. Servers were pets. We cared for them and made sure they had a long life — even bragging about years of uptime. When organizations started moving to the cloud, this attitude persisted for a time before people began to realize that it was madness. The server was just software running on some machine you'd never see and could disappear at any moment. What if you wrote your software where it just didn't care about the server itself? And it worked anywhere and didn't require any lengthy process to initialize or shut down?

Because of these requirements, the concept of the *stateless service* was born. In this case, your service itself doesn't contain anything special: On startup, it's configured to connect to an external data store and sits behind a gateway that can be configured to mask any changes from the world.

On the whole, the service itself becomes greatly simplified and boiled down to logic only. This does, however, present some challenges. Because you never know if the next request will be served from the current service, you can't make any assumptions nor pass the data along directly. Either the data has to come in from the call itself, or it has to come from an external data store.

Knowing Where to Break the Monolith

Looking at a large, complex software architecture and trying to determine where to section it into services can seem intimidating. What's interesting about the microservices architecture is that this is less about the technical requirements for each individual service and more about what is logical for your organization.

In monolithic architectures, you may have very ad hoc patterns for accessing data — cross-joining tables, updates across functional lines, and so forth. In a microservices architecture, however, each service has access to only the data legitimately relevant to the function of that service.

Take, for example, a shopping cart in an e-commerce application. A shopping cart service would track only the items user #123 has in his/her shopping cart. In a monolith, you may have a single row in a table that represents the demographic data about the user (name, location, preferences, and so on). Another table represents the shopping cart that's joined to another table with the product IDs. When breaking out this cart service, you won't, for example, be able to join tables because in a microservices architecture, each service has a private/internal data store only. In exchange, you gain flexibility in terms of composition and scalability. Indeed, in a monolithic implementation of our example, when trying to scale up the number of products, users, or shopping cart sizes, each function (users, products, and carts) is so tightly coupled with the others that they all have to scale together. Whereas having users, products, or carts as a service, you're free to scale, or even modify, each service independently as long as the interface stays the same.

Now, suppose you break out the service and store the data in, say, a Redis Sorted Set — with the quantity represented by the Sorted Set score and the item by the member, as shown in Table 2-1.

TABLE 2-1 A Redis Sorted Set

Score (Quantity)	Member (Product ID)	Redis Command
2	abc	ZADD usr:123 2 abc
1	xyz	ZADD usr:123 1 xyz
1	efg	ZADD usr:123 1 efg

What's absent is any description of what product ID “abc” actually is and who `usr:123` represents. This is okay. The service is doing one thing: keeping track of the cart. The *products service* would tell you what “abc” is, and the *users service* shows who has the “123” ID.

What's cool about this is that each service can be independently scaled. Imagine Black Friday: The shopping cart service will be under extreme write load, whereas you may have just a little more load on the user service or the product service — the product service is probably just reading more than normal in this case. In a microservices architecture, you can scale up only the shopping cart services — essentially adding resources only where they're needed.

The next challenge is to understand what needs to happen when you have an operation that needs to transactionally operate across multiple services. Traditional transactional integrity cannot be guaranteed in this case. Each service has its own data, and these services are using application programming interfaces (APIs) to communicate with each other. The other option employed in similar scenarios is two-phase commit — where the data layer(s) prepare for the data, then block until everyone is prepared, and finally commit all at once. This is impractical because two-phase commit can block further operations and requires a centralized coordinator to make sure everyone is prepared and committed.



Two-phase commit is a coordinated distributed transaction mechanism. In the first phase, all the participating services vote on whether the transaction is valid. In the second phase, if all the services saw it as valid, the data is written, every participating service acknowledges the writes, and the transaction is over. If no agreement is reached during the first phase, any actions are undone, acknowledging the undoing of the transaction, and finally the transaction ends. During all the phases, the services can't do anything else because it might disrupt the validity of the transaction.

Getting Services Talking

After determining where to break a monolith, you next face these challenges:

- » Translating the data layer into a reasonable communication pattern for the specific domain.
- » Determining the correct semantics required by this pattern.

- » Considering not what the other service needs (this would be tight coupling), but rather where it needs to communicate with other services. This makes the overall architecture more accommodating for the future.

Generally, it's unimportant to issue messages that have no feasible external utility. For example, if you're writing a single entity to several different structures in a Redis database, but the entity is written atomically, this should be considered a single message.

Additionally, the way the internal storage is handled should not be visible to the messaging; instead, the messaging should follow the underlying business rules. An HTTP endpoint that synchronously creates a user, for example, would issue a message that indicates that the user is created, not that a new user welcome message was sent.

Finally, while messaging may cover a vast amount of the communication need in a microservices architecture, often a synchronous response is still needed, so it's not an antipattern to, for example, have both an endpoint to check if a package is delivered and to issue a message that a package was delivered.

Having a Conversation with a Service

In this section, we imagine that we have actors playing the roles of our services in an e-commerce application. Let's take a look at a few scenarios and how the services are communicating.

Normal order

In this interaction, we'll follow the personified services when everything works *as normal* — with no hiccups or failures. Take note that, in this example, you never know *how* a given service does anything, and it doesn't matter: You could replace a service with one that does the same operation entirely differently, and as long as they communicate the same way, it's just fine.

Order Service: (*shouting*) Hey, I've got an order #123 for \$1,234 on user ABCD, which is currently `AWAITING_PAYMENT`.

User Service: (*looking at Order Service's event stream*) An order `AWAITING_PAYMENT`, eh? Let's see how much ABCD has in

their account. Yup, that works. I'm going to reserve \$1,234 out of that account for that order. (*shouting*) Order #123 was PAID in full!

Order Service: (*mumbling to itself*) Cool cool cool. Let me update that order. (*shouts*) Order #123 for user ABCD is now AWAITING_SHIPMENT!

Fulfillment Service: (*eavesdropping*) An order AWAITING_SHIPMENT you say? Just a second, let me check stock. . . .

Order Service is now preoccupied with another order, ignores fulfillment service.

Fulfillment Service: (*to Order Service*) Order #123 for user ABCD? I SHIPPED it.

Order Service. Cool. I'll update my records to change this order to COMPLETED_SUCCESSFULLY. Oh, User Service? Could you go ahead and change those reserved funds for ABCD to a full-on deduction?

User Service (*waking from a nap*) What? Where am I? How long has it been? (*Shakes off tiredness*) Right. ABCD, we reserved \$1,234. Let me deduct that fully.

Insufficient funds

The interaction in this scenario outlines the situation where a problem occurs with the payment. Notice that the Fulfillment Service is never even involved — because the domain of fulfillment is never activated, this part of the architecture is idle or doing something else.

Order Service: (*shouting*) Hey, I've got an order #123 for \$1,234 on user ABCD, which is currently AWAITING_PAYMENT.

User Service: (*to Order service*) An order AWAITING_PAYMENT, eh? No can do. Not enough money in User ABCD's account. (*shouting*) Order #123 has FAILED_PAYMENT_NO_FUNDS.

Order Service: Oh, wow, then I guess (*shouting*) Order #123 is now BLOCKED_NO_FUNDS.

Unshippable product

The personified services here have to deal with a shipping problem rather than the previous ordering problem. In this case, the funds have been reserved already, and the User Service has to compensate for the problematic order.

Fulfillment Service: (*eavesdropping*) An order AWAITING_SHIPMENT you say? Just a second, let me check stock. . . . Okay, product “haggis” was found. Shipping address is in the United States. Let me look at my list. . . . Nope, can't ship haggis to the United States, so I guess (*shouting*) Order #123 for user ABCD SHIPMENT_FAILED_BANNED_PRODUCT.

Order Service: Oh, Order #123 for User ABCD was not fulfilled. . . . What a shame. So now (*shouting*) Order #123 is AWAITING_REFUND.

User Service: (*waking from a nap*) What? Where am I? How long has it been? (*Shakes off tiredness*). Right. ABCD, we reserved \$1,234, giving those back now. (*shouting*) Money for Order #123 for User ABCD was REFUNDED.

Order Service: Great, so now I guess (*shouting*) Order #123 for User ABCD is COMPLETED_WITH_FAILURE.

OrderFailureMonitoring Service: Oh, really? Interesting. . . .

IN THIS CHAPTER

- » Understanding why distributed state is tricky
- » Digging into the needs of distributed state
- » Examining data types
- » Weighing pub/sub against logged events
- » Moving data across services
- » Clustering for multi-tenancy

Chapter 3

Distributed State with Microservices

Distributed is often a feared word in software development. Usually it conjures up worries about simple things becoming very complicated, as well as poorly understood terminology and edge cases. In this chapter, we dive into what distributed state is and how you can manage it using the commands and data types built into Redis.

Defining Distributed State

Non-distributed state is straightforward: It has one centralized, canonical representation of state. This state is accessible — you can read (and manipulate) the state fearlessly. Distributed state has many murky corners, making it tricky. By that we mean, with distributed state, no one picture of the data is complete — only messages being passed back and forth create the whole picture.

These messages might arrive out of order, partially, delayed, or even not at all. You must consider failure when you're dealing

with distributed computing. Simply the nature of distribution outlines that, by introducing multiple machines (virtual or physical), you increase the complexity and latency as compared to the same operations being executed in a single environment.

Off-hand examples sometimes include massively oversimplified (and frankly unusual) scenarios of having completely stateless services. These services do things like input a few numbers and calculate a far bigger or smaller number. These examples are presented as a way to scale this very mathematically intensive operation. That's nice, but when dealing with more common problems, we have situations that require pieces of context from all over the architecture, and services can't just accept inputs and give outputs. This real-life class of services needs access to its own data, as well as to the data managed by other services.

Discovering the Needs of Distributed State

In a large, complex production system, state change may be the result of many different inputs. Inputs might be user activity that affects only their own experience or that has a greater effect. Inputs can come from other services, however, at a machine-scale speed affecting a minor or even more global state. Either way, by a volume of users making relatively slow state changes or by a small number of machine processes making high-frequency changes, the state will be constantly moving.

The speed and scale at which the state is changing makes managing this with some sort of synchronous application programming interface (API) call seem overwhelming. Additionally, *changes* to the state may be relevant not just to the end result. Request/response API endpoints are not well suited to this type of change observation.

To cope with this type of rapid state alteration coming from different services, the architecture must be able to keep track of changes in an ordered way, without having to lock or pause any other given process — everything must be asynchronous. Given these constraints, a service may need to retrieve the state as a whole (say, on startup), or it may just need to retrieve changes since a given point in time.

Finally, because distributed state is vital to the operation of many services, anything backing the state should not be ephemeral — in the case of a power failure or other disaster, the state should be able to be restored from last-known good. Additionally, it's unwise to assume all services and connections will always be up and/or high quality. This consideration means anything that will manage distributed state must be able to handle disruptions, preferably without having to re-retrieve needlessly. (For example, re-retrieving a required state only to find out nothing has changed is a waste.)

Exploring Data Types

Redis descends from the lineage of key/value stores but extends the key/value concept to include a data type that holds the value. All data is, at some level, accessible by a key in Redis, but each key also has an associated data type. The data type dictates which commands can be used on the key and how data will be stored at the key. Many of these data types are primarily used for storing information; others are designed to transport information; and still others can be used for both. If the data type can be used to transport data, then it could be useful for distributing state.

In this section, we fill you in on how the following three data types can be used specifically in distributing state:

- » Pub/sub
- » Lists
- » Streams

Pub/sub

Redis has built-in pub/sub, or more formally, *publish and subscribe*. This is not strictly a data type, but in Redis it stands alone and often is categorized as a data type. This mechanism allows for one attached client to publish a message to any number of other subscribing clients. Subscribing clients are placed into a mode where they only receive published messages. As such, these messages are instantly, without any form of polling, received.

Pub/sub in Redis has very little overhead because it's *fire and forget* — a way of communicating that doesn't retain the sent (published) message. The lightweight, fire-and-forget publishing is great for notifying services of something that needs immediate, but not retroactive, attention.

Lists

Lists, effectively doubly linked lists in Redis, are accessed primarily by pushing in and popping values off either to the left or the right end of the list. It's also possible to retrieve, insert, or remove items from the middle of the list, but this comes at a computational complexity penalty.

Like pub/sub, lists do not need to ask for new items; they have the ability to “block” until items arrive. The blocking functionality — paired with low-complexity operations — makes lists ideal for maintaining queues.



TECHNICAL
STUFF

Blocking in Redis can mean one of two things:

- » Blocking the server from doing any other operation, which is usually intentionally done only in very specific situations because it has serious performance implications.
- » A blocking command that blocks a particular database/client connection. This has only a localized effect to a single connection and provides the ability for a command to intentionally not respond until a specific condition is met.

Streams

Recently added to Redis, Streams are a way of keeping time sequenced, field/value pairs. When you add an item to a stream, it's stored by a millisecond precision timestamp and a sequential identifier in each millisecond time frame. Streams can be read between two timestamp/identifier bounds, or, similar to lists, Stream commands can wait for new items.

Additionally, Streams have a mechanism that allows for distributing load across a group of readers to manage a producer and consumer model. Streams can be used to store sequences of data that may be consumed by other services.

Publish/Subscribe or Logged Events

It's tempting to look at pub/sub as an option for maintaining distributed state because it's very lightweight and easy to implement: Services subscribe to what other services are publishing. However, the fire-and-forget property of publishing limits its uses.

Think of a scenario in which something causes a network disruption between services. During the disruption, important events take place that alter the state for some, but not all, services. With pub/sub, any non-connected service just misses out on these events, and you end up with data that eventually isn't consistent; instead, it's inconsistent with no way to recover.

Lists (used as queues) or Streams provide messaging with a form of persistence. With the previous example, disconnected services could still pop items off a queue when they reconnect. Perhaps more compellingly, with Streams, a service can find the point where it left off and process events that alter the state.

Does this mean that pub/sub has no place in a microservices architecture? Pub/sub can provide a very useful notification in situations where processing is relevant only to some sort of ephemeral, point-in-time event. This, however, by nature, is not a state-altering event.

Lists can be used as logged event stores, but they have some limitations. Namely, the list only implies order in a static sense. It is possible to get a range of values by an index offset, but this is dependent on the position relative to either side (left/right) of the list; this operation may be computationally intensive, depending on how far it is from each side. Indeed, unless the list never grows or grows indefinitely, lists are better used to represent a queue rather than a list of logged events.

Getting Data across Multiple Services

Getting data across different services is a challenge in a microservices architecture. You have to keep data isolated yet allow for both synchronous and asynchronous communication. Additionally, you need to make sure that each service is responsible for

data related to its domain. Let's take an example of three services used to send an e-book to a user:

- »» The User service maintains biographical information about users.
- »» The Mailing service takes care of sending the book to the user.
- »» The Order service accepts orders.

The Order service has the user's email address and PDF name, and it synchronously triggers the Mailing service. The Mailing service needs the first and last name from the User service to proceed, though. Although it would be possible to synchronously contact the User service for every mailing, this starts to resemble a waterfall with rising latency.

If the Mailing service needs only a small bit of information from the User service, is it worth it to make a synchronous call for every mailing? How can this be resolved without making an extra API call? The answer lies in using messaging with a Stream:

- »» When the User service updates a user's name, it adds an entry in a Redis Stream containing the user ID and the name, as well as updating the normal data held by the service.
- »» The Mailing service doesn't care about biographical information for users in this case, so it ignores any messages about this.
- »» The Order service does care about the name change, so it consumes messages relating to biographical information.

When the Mailing or Order services consume the information from the User service, they record this in their own private stores. Indeed, this is critical: If all the instances of either of these services is down, they must be able to find out the current state by consuming messages past a given known recorded point. This makes the services tolerant to underlying infrastructure failure. Additionally, because each service has its own copy of the data, these services can still be up and running even though the User service may be in an outage scenario.

One of the hard and fast rules of a microservices architecture is to make sure each service is only loosely coupled to other services;

a key point to this is that services should not be able to share the same database. This may seem arbitrary, but if you think about it, if you have two services that write to the same database entry, then both services have to be updated when the format of the entry changes. Considering this scenario, how does this hold true when multiple services are reading from the stream, as shown in Figure 3-1?

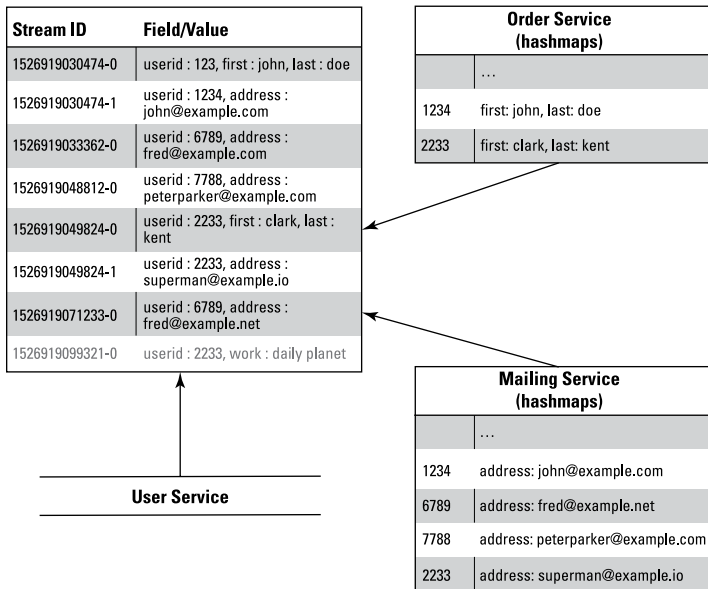


FIGURE 3-1: When multiple services read from the stream, the stream transports information rather than stores data.

Here's what's happening with each of the services:

- » *User service* is writing to the Stream regarding userid 2233. This has not completed yet.
- » *Order service* is reading only demographic-related information from the Stream and is somewhat behind. The information from the Stream updates the internal store for service.
- » *Mailing service* is reading only address-related information from the Stream and is up to date. The information from the Stream is stored in its own internal store.

In this case, the stream is acting as an interface or *transport* of information rather than storage of the data. Also note that in this example, we're only ever reading from the stream. The stream key is analogous to the endpoint, and the results are similar to how an API endpoint outputs the data. As long as the User service keeps this information the same, how the data is processed coming in would also stay the same.



WARNING

This model is fine if you have one instance of each service. What happens if you have, for load distribution reasons, four instances of the Mailing service? Having each copy of the Mailing service write an update to the internal mailing service database is, at best, wasteful and, at worst, could yield incorrect data.



TIP

Redis solves this with the Streams consumer group commands. This enables a group of consumers (those reading from the stream to get the state, in this case) to manage how the items are delivered to groups of clients (also known as single instances of the same service). In this way, you can have unread, pending, and acknowledged messages.

Clusters, Multi-Tenancy, and Redis Enterprise

If you have only worked with a single instance of Redis, the idea of Redis being central to a microservices architecture may seem a little odd. A single instance limits you to only the random access memory (RAM) available on a virtual machine — outgrowing these confines seems easy. It gets more complex when you consider creating a stream for connecting services and keeping all the data isolated. Redis Enterprise has the ability to cluster together many instances of Redis across many machines and to provide a management plane to isolate data from cluster and database operations.

A Redis *cluster* is a collection of nodes. There is no central point to the cluster, just a series of peer nodes. A single cluster can contain a number of databases. These databases are completely isolated from one another — from the standpoint of data and keys and central processing unit (CPU) utilization. A database is made up of a number of shards or a division of the Redis keyspace. Each *shard* is responsible for a number of “hash slots,” a further logical division of the keyspace.

Nodes and shards

A node of Redis Enterprise represents a machine (a container, virtual or bare metal). Each node, as shown in Figure 3-2, has two primary layers, the *Enterprise Layer* and the *Open Source Layer*.

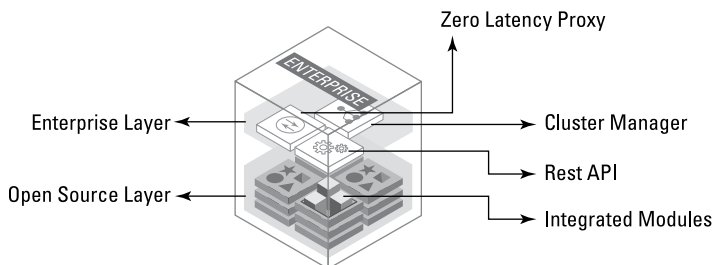


FIGURE 3-2: A node of Redis Enterprise has two primary layers: the Enterprise Layer and the Open Source Layer.

The Enterprise Layer provides management and access control to the underlying data path located in the Open Source Layer. Redis Enterprise bifurcates responsibilities of management and data manipulation so that management is done through an entirely separate API and authentication mechanism as compared to the data layer. Each node in the cluster is a peer when it comes to the Enterprise Layer.

Your service connects to Redis Enterprise via a Zero Latency Proxy. This proxy is compatible with the Redis open-source protocol (Redis Serialization Protocol, or RESP) and simplifies developing software as it abstracts away clustering complexities. To your code, a Redis Enterprise database appears as a single, very large instance of Redis. If you enable High Availability on Redis Enterprise, the database endpoint will stay the same even if the underlying node goes down.

The Open Source Layer contains a series of shards, each pinned to a CPU core. Each shard is equivalent to a single instance of open-source, non-clustered Redis, inheriting the single threaded, event loop architecture of the Redis core.

Individual shards are solely responsible for a number of hash slots — no key will ever reside in multiple hash slots (and, thus, neither in shards nor nodes) in a given cluster. This allows Redis Enterprise to maintain true multi-tenancy: Both CPU and

memory resources are isolated entirely from both other databases and other shards. Shards may also integrate a module as needed to extend the functionality of the database.

The cluster and databases

Aside from being made up of a number of nodes, a cluster provides a common management, configuration, and monitoring pathway to each database contained in the cluster. A single database may span multiple nodes in the cluster, allowing the database to scale horizontally, as shown in Figure 3-3. A database can also be configured to extend outside of RAM by utilizing flash memory as a “cooler” storage tier.

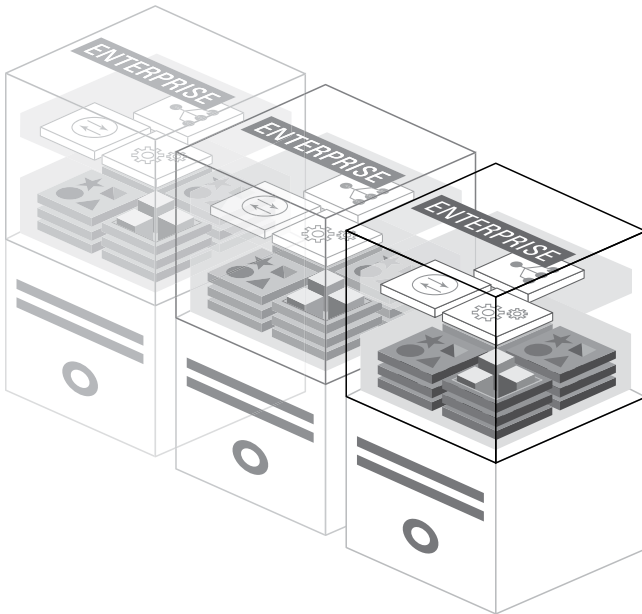


FIGURE 3-3: A single database spanning multiple nodes in the cluster, which allows the database to scale horizontally.

The cluster manages high availability at the database level by setting shards as either master or replica instances. The cluster is aware of placement of these shards and can ensure that a single node failure will not bring down both master and replica.

Persistence is also controlled at the database level, to allow for a tunable persistence. Redis Enterprise can be configured to be entirely ephemeral or persisted with periodic snapshotting, or individual write-level Append Only File (AOF) persistence. This enables both databases geared for caching and for primary database workloads to exist in the same cluster.

From the perspective of a microservices architecture, a single cluster can provide databases to many different services, each with its own isolated instance, tuned for the given workload. From an operational standpoint, this can be deployed in a number of different ways, either directly or through a Kubernetes orchestration layer on Google Kubernetes Engine (GKE), RedHat OpenShift, or vanilla Kubernetes.

IN THIS CHAPTER

- » Defining CRDTs and Active/Active
- » Exploring synchronized data and how data changes
- » Using Active/Active for primary data storage
- » Caching with and scaling writes with Active/Active
- » Knowing where to use (and not use) Active/Active

Chapter 4

Active/Active and Microservices

Data cannot move faster than the speed of light (299,792.5 km/second). As a consequence, the physical distance between a user and a server, and a server and a database, is critical for low latencies. In this chapter, we explore how an Active/Active database can cheat these physical limits and allow for geo-local latencies in multiple points without having to worry about conflicting writes.

What Are CRDTs, and What Is Active/Active?

You may have never heard of CRDTs or Active/Active. Indeed, it's an *active* area of research (see what I did there?). As with anything new, these are often both overgeneralized and overcomplicated

by anyone first learning about them. To begin, let me break down what each term means and how they relate to each other.

» **CRDT** stands for *conflict-free replicated data type*. This class of data can be replicated across two or more machines and then recombined in a predictable manner without losing data. In other words, your data can exist in two different places and, well, that's okay.

Usually this is a very bad thing because you can't tell which piece of data is correct. CRDTs must have some structure and certain (predetermined) semantics, as well as implicit metadata that let the database reason about the resolution should two or more versions ever get out of sync. Redis relies on the concept of data structures and already maintains much of this metadata.

» **Active/Active** describes the architecture of a cluster in which one can both read and write to two different machines for the same piece of data.

Imagine asking who's in charge on a construction site only to have two people (Foreman Red and Foreman Dis) respond, "I'm the foreman responsible for all tasks." You could think of this as an Active/Active construction site — requiring careful coordination to make sure work was effectively completed without missing a step.

From the perspective of Redis Enterprise, CRDTs are the method in which an Active/Active architecture is achieved.

Returning to our construction site, the two foremen could work together by constantly chatting about the tasks at hand, but that would be a lot of communication overhead. On the other hand, both foremen share a very specific set of rules for every type of task, allowing for one to immediately understand what's going on with any given task. If Foreman Dis was out sick, you could always just ask Foreman Red — they're interchangeable and can work at the same time. This is, in a bit of a stretched metaphor, how Redis Enterprise does Active/Active.

Another, less abstracted, example would be a shopping cart on an e-commerce site while traveling:

1. Imagine, as a user, that you're on a train that winds between Copenhagen and Rome. You do some online shopping on your phone.
2. As you speed along, you add and remove items from your cart.
3. On your journey you're routed to different data centers based on your location for lower latency.

Without Active/Active, two options exist:

- a. The server can write quickly to a copy of the data at each data center, which may cause an item from your cart to be overwritten by out-of-date data.
- b. The database could have one master copy the shopping cart that exists at one location, losing the geo-local advantage of your data centers because the database could be physically far from the servers.

With Active/Active, you get the best of both worlds because you can read and write to any copy of the data, and the CRDT-based data resolution means you won't inadvertently overwrite as the databases synchronize.

4. You get to keep filling your cart — everyone wins.

Synchronizing Data across Clusters

Redis Enterprise is a cluster made up of nodes that equates to a container, virtual machine, or physical machine. Each node is responsible for a portion of the data in a database. Nodes, themselves, are made of shards that also contain a further subdivision of the data, which is managed by a core of a central processing unit (CPU). So, for example, suppose you have a four-node cluster and each node has two shards. It's possible to have a database that has eight keys, and each of these keys would be managed by a different core. In this situation, the data is never touched by two different CPU cores.

When using Active/Active Redis Enterprise, clusters are joined together, and each cluster maintains an entire representation of data. When a change is made to one cluster, that data is automatically synchronized with the rest of the participating clusters. Later, you can add more clusters, and they'll synchronize the data

as required. A cluster can be offline for any amount of time and rejoin, at which point the data will become synchronized from any previous state.



REMEMBER

The clusters cannot be fully consistent with each other as they may be global-scale distances apart, and Redis Enterprise does not wait for global consensus (making this flavor of Redis Enterprise “consensus-free”). Because of the properties of CRDTs, they always synchronize to the same state, even if individual keys become very disparate.

Understanding How Data Changes

CRDTs can be difficult to understand without an example. Let’s consider the situation shown in Table 4-1.

TABLE 4-1 Resolving Counter Conflicts

Step	Operation	Cluster A Value	Cluster B Value
0	<i>Initial Sync Value</i>	Not set (0)	Not set (0)
1	> INCR foo	1	Disconnected
2	> INCRBY foo 2	3	
3	> DECR foo	2	
4	> INCRBY foo 10	Disconnected	10
5	> INCRBY foo 13		23
6	<i>Sync</i>	25	25

Because Active/Active Redis Enterprise knows you’re semantically treating the key *foo* as a counter, it can calculate the correct value. For example:

- » After Step 3, *foo* has a counter value of 2 according to Cluster A, and Cluster B is not set (effectively a 0).
- » After Step 5, Cluster A still has a counter value of 2, while Cluster B has a counter value of 23.

- » After Step 6, when a synchronization occurs, Cluster A and Cluster B have come to the same value because it's the sum of the known events instead of deciding if Cluster A or Cluster B has the correct value.

The preceding scenario is not complicated to understand, but as you get into other data types — like Sorted Sets, Sets, or Lists — more complex semantics are needed to ensure a predictably synced state. Let's take an example, shown in Table 4-2, of a group membership represented as a Redis SET.

TABLE 4-2 Resolving Set Conflicts

Step	Cluster A Value	Cluster B Value
0	> SMEMBERS group []	> SMEMBERS group []
1	> SADD group paul	
2		> SADD group jorma
3	> SMEMBERS group [paul]	> SMEMBERS group [jorma]
4	Sync	
5	> SMEMBERS group [paul, jorma]	> SMEMBERS group [paul, jorma]
6	> SREM group paul	> SADD group grace
7	> SREM group grace	
8	Sync	
9	> SMEMBERS group [jorma, grace]	> SMEMBERS group [jorma, grace]

In Table 4-2:

- » Steps 1 through 5 can be reasoned about simply: Sets exhibit *add win* behavior resulting in a union of the two conflicting states.
- » Steps 6 through 9 follow the *observed remove* rule, which means the database will remove members of a set only

when the instance has already been made aware of that member.

In this case, `grace` was added to the group in Cluster B but not Cluster A.

- » With the observed remove rule, the command in Step 7 is ignored because Cluster A has, until that point, not been aware of the member `grace`.

This is a different circumstance than in Step 6, because Cluster A knows about `paul` because of the synchronization that occurs on Step 4.



TIP

Thankfully, all these rules are not something you need to code yourself. Redis Enterprise takes care of this automatically. You can find supported types and how each one resolves at <https://docs.redis.com/> in its “Developing for Redis Enterprise” section.

Fitting Active/Active into a Microservices Architecture

A microservices architecture has many connected services yet has the same performance demands as any other piece of software: Delivery as near to 100ms end-to-end is critical for an instant experience. Consequently, this connection between the service and the database is critical and, as such, you want your data to reside as close to your services as possible.

The distance traveled from your user to the location in which the application (and all the services) is should be as small as possible. With Redis Enterprise Active/Active, you can have multiple geo-local installations of all your services, and all services write and read from instances that may be in the same data center or even the same rack. Each service is connected to a complete local, active copy of the data. Redis Enterprise synchronizes the data between the connected clusters, so it doesn't matter if you read and write to a data center in Vancouver or one in New York — the data set is maintained without conflict.



In a microservices architecture, each service must maintain its own data. This means you may have 25 load-balanced instances of your BookLending service all connected to the same database, a database that may be Active/Active and synchronized across many clusters. However, it's an *antipattern* to have more than one service operating on the same data. So, you can't have the BookLending and the User service both manipulating the same data — this would be considered *tight coupling*.

Primary data storage with Active/Active

Services often have their own storage needs that can be easily addressed with Redis. As an example, an activity tracking service may keep track of interactions with items by a unique identifier. This type of data is easily structured in Redis by using simple counters and a key based on the identifier. Many copies of the service may exist; to ensure geo-local latencies, they may be distributed to several data centers across a wide area. Active/Active can be used to maintain the data across the data centers in a way that prevents conflicts and erroneous information.

The topology in Figure 4-1 shows three clusters, each connected to four instances of the same service. Users would be routed to the gateway (not shown) that's physically nearest to them, which is in turn connected to the required services. Because each of the services has a Redis Enterprise Cluster within its own data center, reads and writes are routed only to the local service.

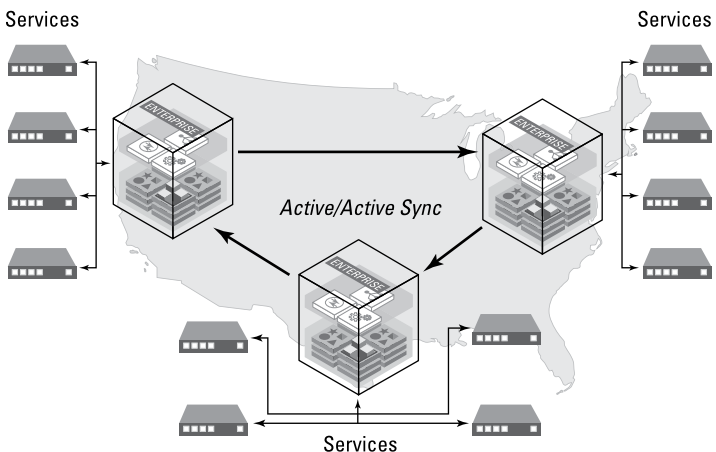


FIGURE 4-1: Three clusters, each connected to four instances of the same service.

An interaction with an item by a user in the east is recorded to that database, and Active/Active will automatically synchronize to the other clusters. Users connected to the other data centers will be working with the same data, yet writes and reads are without extra latency.



REMEMBER

An additional advantage of this topology is that an extra degree of flexibility and fault tolerance can be achieved. In a situation where the local cluster becomes unavailable due to a disaster, services can be pointed at another Active/Active cluster and only suffer the latency penalty.

In Figure 4-2, the cluster in the south has gone down and yet the services are still up. In this case, the services in the south are being redirected to the cluster in the west. When the cluster in the south is restored or replaced, it will automatically resynchronize, and the connected services can reconnect to the local cluster. This is possible because the data underlying the cluster is replicated across multiple servers.

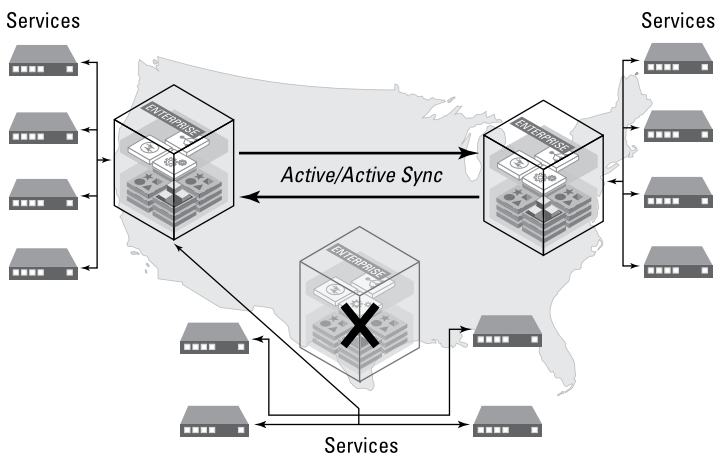


FIGURE 4-2: Although one cluster is down, the services are still up.

Caching with Active/Active

Commonly, a service may rely on a primary database and a cache. The database could be a traditional relational database, a document database, or even Redis itself. This database may have specific transactional or query requirements that cannot be met by an Active/Active Redis Enterprise database itself. In this case, you can employ an Active/Active database as a caching layer.

In this topology, writes are made to the primary database and reads are cached into the Active/Active database. While the primary database is centralized, the cache writes and any invalidations occur on an Active/Active database. This ensures that as many operations as possible can bypass the bottleneck of the centralized server and hit the cache first. Active/Active ensures that each geo-local cache is as up to date and not invalid as possible.

Figure 4-3 shows a cache setup with one cluster on the West Coast and another on the East Coast of the United States:

- » Requests that come into the services on the West Coast have a cache provided by the cluster on the West Coast, and the same setup is on the East Coast.
- » If a particular cache value is invalidated by an update to the primary database, the cache would be populated directly to only one cluster. Through the Active/Active syncing process, both clusters would have the same cached values automatically — no need to manually update the West or East Coast caches individually.
- » This prevents the primary database from populating the cache twice, and it provides local latencies to the services.

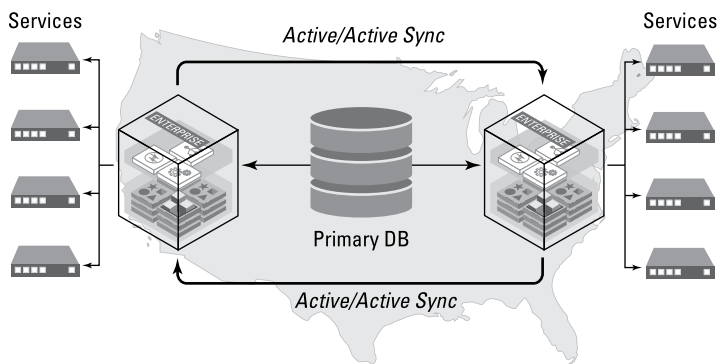


FIGURE 4-3: Separate clusters prevent the primary database from populating a cache twice.

Scaling writes with Active/Active

In some use cases — such as Internet of Things (IoT) or analytical data gathering — data may arrive extremely rapidly. In a

microservices architecture that powers these types of use cases, the write speed is critical. Although Redis (without Active/Active) is widely used in these scenarios, extremely heavy writes can be distributed by an Active/Active database.

In a traditional clustered environment with Redis, any single key will reside in a single shard only for writing purposes. When the write throughput driven to a single shard exceeds its capacity, spreading the load over multiple clusters means that a single key resides in two or more places. The single key's conflicts can be resolved automatically with CRDT resolution mechanisms provided by the Active/Active database.

Shown in Figure 4-4 is an example of a single key (`foo`) being written to simultaneously by many services. The services on the left are connected to one Active/Active Redis Enterprise cluster, while the ones on the right are connected to another.

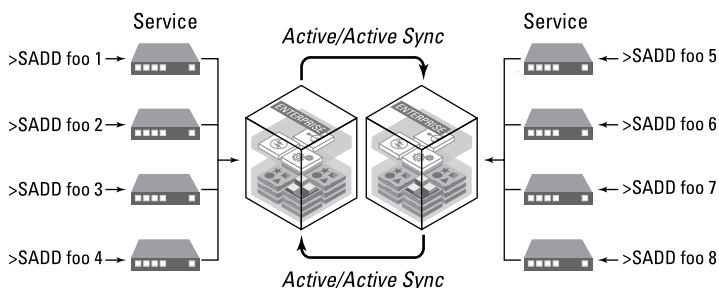


FIGURE 4-4: The single key (`foo`) is written to simultaneously by many service instances across two clusters.

These two clusters would be able to achieve completely parallel workloads, even working on the same piece of data.

In contrast, a traditional Redis Enterprise cluster would sequentially process each request on the same piece of data: If all eight requests came in precisely at the same time and each request took, for example, one millisecond to process, the eighth request would respond as complete in the eighth millisecond. With Active/Active, the workload would be split, so the eighth request would be processed at the fourth millisecond using the same assumptions.



Data is often more equally distributed than described in this section. Traditional Redis Enterprise clusters can scale writes by increasing the number of shards. While Active/Active would not be greatly detrimental to write performance in a case where the data had many keys distributed equally among the shards, scaling writes with Active/Active would see the greatest benefits in a situation with only a few, yet very active keys.

Knowing Where to Use (and Not Use) Active/Active

CRDTs and, by extension, Active/Active are fascinating technology, but these are not solutions for every problem. The most important aspect to understand is that Active/Active will yield results that aren't immediately consistent. Because of the physical limits of signals, writes and reads processed from one location still need to travel to the other location before they'll be resolved. The transmission between the two locations takes time. Consequently, data will be incorrect for small amounts of time before synchronization.

Take, for example, an account balance:

1. After synchronization, the account is known to have a balance of \$100 by both clusters. A user connected to Cluster A wants to withdraw \$60 from this account.
2. The service checks Cluster A's copy of the data, sees that it has more than the amount required, and processes the withdraw.
3. At the same exact time, another user request to withdraw \$60 from the same account comes in with a service connected to Cluster B. The service calculates that the account has ample funds and also processes this withdraw.
4. Milliseconds later, Cluster A and Cluster B synchronize, and the account balance is now -\$20.

This balance is *correct*, but Cluster A or Cluster B erroneously approved the withdraws.

This is a case where it would be wholly inappropriate to use an Active/Active database.



TIP

Currently, not all data types are available in Active/Active databases, and Redis modules aren't available either. If you need to rely on an unsupported data type or module, think through whether your data can be modeled in a fashion that would work for the Active/Active compatible data types.

IN THIS CHAPTER

- » Describing the service
- » Looking at the service's language and setup
- » Exploring the lending and returning events
- » Testing the service

Chapter 5

Building a Service

In this chapter, we show an example of a service implemented in Redis using Python. As with any microservices architecture, each service individually is pretty simple; the power comes from the composition of services into the architecture.

We use the example of an automated book-lending library — imagine requesting a book and a robotic arm fetching the book from a vast warehouse and depositing it into a bin right in front of you.

This service example illustrates the software steps and logic that enable a portion of the overall service; this chapter also provides a simulator of the other services that interact with the example service. Find all of this on GitHub at <https://github.com/RedisLabs/redis-microservices-for-dummies>.

Getting Clear on What This Service Does

The service is the Lending Service. The overall architecture of the entire application could be very complex, handling a variety of lending library operations like fines, membership, and so on, but this service is responsible for just two primary things:

- » Processing book-lending requests
- » Processing book returns

REDIS AND ROBOTS?

Combining robots, Redis, and a microservices architecture might seem odd. Although the Lending Service example is a complete work of fiction, the idea of using these together is not. The Los Angeles-based company Elementary Robotics uses Redis in a microservices architecture to control robots in industrial applications.

These processes need to understand what books the library has and if these books are lendable or returnable. It also has to understand if a given user has the ability to take the book out based on a maximum lending limit. Importantly, it needs to do this in a fault-tolerant way — if the service goes out mid-processing, it won't lead to a corrupt state or allow invalid actions to occur.

Because this is a synthetic example, here are a few assumptions and simplifications:

- » Books are unique by ID, and this library has any book you can imagine.
- » Books are always available as long as they're not already lent out.
- » Lending Service can only “store” a limited number of books and is used for short-term storage only.
- » When books are not lent out or in short-term storage, they're managed by the faux Shelving Service.

Understanding the Language and Setup

The Lending Service is written in Python 3 using `asyncio` to provide nonblocking asynchronous operations (`async / await`). Additionally, it takes advantage of Redis's built-in scripting language Lua to provide concurrency control and to simplify the overall service.

In the root directory of the GitHub repo is the `main.py` file. This file will launch both the service and the simulator, so you can see

how it all works together. In the `/services` directory are the following two files:

- » `lending_service.py`: The example service
- » `shelving_service.py`: A partially implemented service just used to illustrate how `lending_service.py` works



REMEMBER

We focus only on how `lending_service.py` works. The `shelving_service.py` provides only the bare minimum surface area with which `lending_service.py` interacts. If you're planning to build a robotic library lending service, you should probably look elsewhere for a boilerplate!

In the `/lua` directory, you'll find the scripts that will be executed by Redis. These scripts are stored as separate files in the repository but will be transmitted to and compiled on the Redis server where they're also cached. Executions of the script will be invoked by the `EVALSHA` Redis command. All the Lua scripts are invoked by `lending_service.py`.

For the examples, you'll need an instance of Redis. If you have Redis installed on your machine, this could be a localhost instance, or you could connect to a remote server as well such as Redis Enterprise. Redis Enterprise is compatible with open source, and you don't need to do anything special or rewrite your application to take advantage of the enterprise-grade features.



TIP

If you don't have an instance of Redis handy, you can always sign up for a free 30MB starter instance at <https://redis.info/dummies-free>.

To get the service up and running, follow all the script installation steps in the GitHub repo's README file. After you've installed the dependencies, run the `main.py` file with a unique process identifier as the first argument:

```
$ python3 main.py myuniqueid
```



TIP

You may need to apply command-line switches if you have a remote Redis server or have set a server password. Here are some common options for the example code:

- » `-a [your address]` specifies a remote host name or IP address.

- » `--password [your password]` passes an authentication password to the Redis server.
- » `-h` displays the help for the command-line arguments.

This will launch the Lending and Shelving services for the example. In a production scenario, you would deploy both services individually instead of running a single Python script. This will seemingly do very little, only outputting a line that reads "Ready to process events...". Internally, the services are now ready to process events but don't yet have any events to process.

Processing Events

The Lending Service's two actions (lending books and accepting returns) have specific actions that occur in Redis both in messaging and in storage. The messaging actions are based around Redis Streams, and the storage is modeled with Sets, Counters, and Hashes.

Lending books

To process lending requests, we have a few sub-steps to consider. In the source code, this is in the file `services/lending_service.py` and in the class `LendingService` and method `process_lending_request`.

When reading from the stream, the method expects `user_id` and `book_ids` as fields in the stream entry. Although not implemented in the example code, the entry data would be accepted as part of an internally exposed HTTP application programming interface (API) endpoint for the `LendingService`.

The HTTP server would then insert the entry into the stream using the `XADD` Redis command and end the HTTP connection. The service endpoint isn't waiting for the actual internal process to complete, but rather waiting only for the entry to be added to the stream (a very short amount of time).

The `process_lending_request` method first looks for any pre-reserved books in a set with `SMEMBERS` and stores them in an internal processing array. This step is required to ensure that, in the case of a crash during processing, any previously reserved books

are processed. Then it takes the books requested with `book_ids` and individually performs the following actions:

- » Checks if the book has been previously lent in a hash with `HEXISTS`.
- » Tries to retrieve the book from automated storage by atomically moving the book from the general book set into a pre-reserved books set, adding it to the internal processing array as well.
- » Tries to get the book from the Shelving Service by calling the `shelving_service.get_book` method, which is a mocked-up synchronous, idempotent REST API call to another service. Pending the successful result of this call, it's added to the internal processing array.

Idempotence is an odd word but a simple concept. It means that if you perform an action for the first time, it will change in some way; but in subsequent times, it will not change. Think of it as a machine that has an on and off button. If the machine is off, pressing the on button is idempotent from an off state — it turns on; from an on state, it's still just on. The reverse is true for the off button. A machine with a power button that toggles the on/off state is an example of a non-idempotent action.



The interesting thing about these steps is that they're tolerant to a crash during any point because the actions are idempotent. So, if your underlying infrastructure dies, the entire set of actions can be replayed without weird/partial states. This is enabled by Redis's set data structure itself not permitting duplicates.

After we have a complete list of books to process in the internal processing array (sourced from any pre-reserved books and from any new ones from this request), we can do one of two things:

- » If the internal processing array is empty, we can just acknowledge that we've processed this entry in the stream with the Redis stream command `XACK`.
- » If we have values in the internal processing array, we have to process the books.

Processing the books requires that we transactionally modify the keys — to do this, we use Redis's Optimistic Concurrency Control

system with the `WATCH` command. This command enables us to implement a user/key-level pseudo lock inside the script. This means that if simultaneous requests from the same user came in, only one would succeed.



TIP

This type of locking is useful in preventing fraudulent attempts at exceeding limits by issuing two or more parallel requests.

At this point, we can safely evaluate if the request exceeds any predefined lending limits:

»» **If the limits are exceeded, then we have to compensate by undoing reservations.** This is accomplished with a Lua script that streamlines several dependent modifications down to a single Redis call. Finally, we delete the book reservations key and acknowledge the stream entry.

This all occurs in a transaction, so while it's running, it's uninterruptible.

»» **If the limits are not exceeded, again start a transaction and increment the lending count for the user.** Then we set the field for book ID and the value for the user ID in a hash object that reflects the state of the borrowed books.

Still in the transaction, we can delete the book reservations key and acknowledge the stream entry.

Returning books

Returning books works similarly to lending. Just like with lending books, the information would come in through an HTTP endpoint, which would then only write to a stream with `XADD`. The service is monitoring the stream and will process those items in order. The stream entry should have the `user_id` as well as the `book_ids` fields.

Returning books starts a transaction and then all books IDs from the `book_ids` field are put into a temporary Redis set.

The `apply_book_return` Lua script is invoked. This script iterates through the temporary set of `book_ids`, deleting the book field/value from the lent book's hashmap if it's lent to the user in question. If the book is not lent to the user, it's removed from the temporary set so that the only items in the temporary set are the successful returns. We can use this count (`SCARD`) to determine the new count of books for a given user.

Next run the `refill_automated_storage` and the `books_to_stream` Lua scripts — these are the same scripts run when lending the books, just composed in a different way.

Finally, we clean up the temporary key and acknowledge the stream entry with `XACK`.

Interestingly, all this was run inside a Redis transaction, so we're completely safe from other instances messing with the data during the entire course of running the return operation.

Invoking the Service

Now that you know how the service works, it's time to take it for a test spin. Because we're showing only how a service works in Redis, we've omitted the actual REST endpoints and deployment; instead, we'll interact with the service with a simple script.

If you were deploying the service, you would create your service in the deployment environment of your choice and then attach your Redis Enterprise database to it and expose all your proper ports and addresses. This can take many forms; with Redis Enterprise's multi-tenancy, you have a lot of options to provide both persistent and ephemeral instances to your services.



TIP

Before you begin, have the `main.py` script running in one terminal window and invoke the service in another window. This will let you see all the output logging of the service while being able to interact with it.

First, borrow a book with the `get_books.py` script. This script requires a few arguments: the operation (`request` or `return`) followed by the user identifier and one or more book identifiers.

For these two arguments, you can just use *Salvatore* and *ofmiceandmen*. Also, connect to a fictional Redis server located at `192.168.0.40` using `wibblewobble` as the password:

```
$ python3 get_books.py \  
-a 192.168.0.40 \  
--password wibblewobble \  
request Salvatore ofmiceandmen
```

The output of the `main.py` window should look something like this (the large number is time based, so it will be different):

```
Ready to process events...
Request: [1566505618781-0] by Salvatore ACCEPTED.
Books:
1) ofmiceandmen
```

After borrowing the book, you should return it. To do that, you can just run the same script with a different set of arguments:

```
$ python3 get_books.py \
-a 192.168.0.40 \
--password wibblewobble \
return Salvatore ofmiceandmen
```

The output of the `main.py` window should add something similar to this line:

```
Book Return [1566505636987-0] PROCESSED
```

If you want to see what's going on internally to the service and you have access to the Redis command-line interface (`redis-cli`), you can see how everything is being processed by Redis. To do this, open yet another window and run the following command:

```
$ redis-cli MONITOR
```



REMEMBER

You'll frequently see `XREADGROUP` commands being displayed. If you scroll back after running these operations, you'll see all the individual commands being run, including the `XADD` and `XACK` commands that are managing the entries, as well as the commands being called from the Python and Lua scripts.

Chapter 6

Ten Key Microservices Takeaways

This book is about the microservices architecture and how you can use Redis to help you develop and operate the architecture with high performance in mind. Here are ten key takeaways regarding Redis and the microservices architecture:

- » **Slow databases don't work in a microservices architecture.** A microservices architecture is made up of many individual parts that communicate with one another, and the individual services should be stateless. The nontrivial bottleneck in this architecture is a slow database.
- » **Redis can be used as a database for a service.** Redis is configurable to be persistent, so you can use it to store data for one or many of your services.
- » **Redis can be used to connect services together.** Redis Streams are ideal for providing a durable, loglike record of state changes. This allows for services to write to their own stream and/or read from the streams of other services.
- » **Redis can be used to cache values of other databases in a service.** Cumulative latency in microservices architectures means that each service needs to operate as fast as possible.

Putting Redis in front of slower databases as a cache prevents expensive database calls.

- » **Redis Enterprise is built for multi-tenancy.** A single cluster can provide isolated Redis databases that are a great fit for a microservices architecture. You can isolate both data and compute resources for your services, so you never have to worry about noisy neighbors.
- » **Your services can be distributed geographically.** Redis Enterprise provides conflict-free replicated data types (CRDTs) that allow for Active/Active databases — you can have instances of your services spread across the globe and CRDTs allow reading and writing of data to the same database without complicated conflict resolution.
- » **You aren't limited by the amount of random access memory (RAM) in your cluster.** Redis Enterprise allows you to extend your RAM into flash memory (solid-state drives, or SSDs). This means you can have much larger data sets than could fit affordably into RAM alone.
- » **Redis data structures can be composed for more complex scenarios.** A single built-in Redis data structure doesn't need to represent your entire data model. Instead, consider each component of your data and which structure could best represent it.
- » **Modules extend Redis to do more.** RediSearch allows for rich, full-text search capabilities. RedisGraph is great for complex, unstructured relationships to be queried. RedisTimeSeries allows you to rapidly record and aggregate over time-series data.
- » **Redis gives you the right tools for the right job.** One example is Redis Pub/Sub, which trades persistence for properties that make it a lightweight and yet robust tool for propagating live notifications between services.

Advanced Microservices Design Patterns

Design patterns enable software architects and developers to apply a well-known solution to a particular problem. Within this appendix is bonus material, including a link to a video showing common design patterns with microservices.

Deploying microservices can be accomplished in numerous ways. However, deployment of microservices to ensure scalability requires an understanding of the best practices and the best technologies to meet the needs of modern applications.

Defining Microservices

Although there are multiple ways to deploy microservices, all microservices have some generally agreed-upon characteristics in common. At a very basic level, microservices divide monolithic architectures into multiple services, each providing its own component and each managing its own data. Components are self-contained and designed around products, not projects. Infrastructure automation is frequently used in order to reduce the time to market for microservices. Microservices are designed with failure scenarios in mind and benefit from an evolutionary design and development flow.

Reviewing Design Patterns

Allen Terleto from Redis created a video for the Kafka Summit 2020 in which microservices and design patterns are discussed. The video shows many common design patterns and even an anti-pattern, as well as how Redis and Kafka are used as

industry-leading technologies meeting the need to deploy microservices at scale. The video is available at <https://redis.com/webinars-on-demand/redis-and-kafka-advanced-microservices-design-patterns-simplified/>.

The design patterns and advanced concepts examined in the video include:

- » Bounded context: Domain-driven design
- » An anti-pattern: Two-phase commit
- » Publish-subscribe pattern
- » Choreography-based saga (state machine)
- » Transactional outbox
- » Capturing telemetry
- » Event sourcing
- » Command Query Responsibility Segregation (CQRS)
- » Shared data

The video does an excellent job of covering both the concepts themselves, as well as the reasons why a given design pattern may be used. After watching the video, you'll have a strong foundation to build and deploy microservices at scale with Redis and Kafka.

Future-Proof Your Cloud Strategy

Redis Enterprise Cloud

Unify hybrid cloud and multicloud deployments

Consistently deliver instant experiences

Scale your business at light-speed



redis.com/cloud

Microservices architecture made easy with Redis

The microservices architecture has a huge number of advantages over legacy architectures. Redis sits at the heart of a high-performance microservices architecture where it provides a whole host of capabilities that cache, connect, and store data for services. Unlock the potential of the architecture without having to worry about latency headaches that come from breaking the monolith. Find out how Redis Enterprise enables you to implement your application globally with one unified, low-latency data layer and can support any real-time application at scale across a myriad of use cases such as real-time analytics, fraud detection, session management, gaming leaderboards, claims processing, and more.

Inside...

- Explore microservices architectural concepts
- Understand how to use Redis to accelerate services
- See Redis as a connector between services
- Find out how to use Redis as a primary store for your microservices



Kyle Davis is the Head of Developer Advocacy at Redis and an instructor at Redis University. He has written extensively about Redis, as well as presented about Redis at domestic and international conferences. **Loris Cro** is the Developer Advocacy Manager at Redis and has written Redis modules, as well as forged the path for the Redis's use of the Zig language.

Go to **Dummies.com**[™]
for videos, step-by-step photos,
how-to articles, or to shop!

ISBN: 978-1-119-82429-9

Not For Resale

for
dummies[®]
A Wiley Brand



WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.