Command Query Responsibility Segregation by Example

Carlos Buenosvinos
Christian Soronellas
Keyvan Akbary

CQRS by Example

Command-Query Responsibility Segregation is an architectural style for developing applications that split the Domain Model in Read and Write operations in order to maximize semantics, performance, and scalability. What are all the benefits of CQRS? What are the drawbacks? In which cases does it worth applying it? How does it relate to Hexagonal Architecture? How do we properly implement the Write Model and Read Models? How do we keep in sync both sides? What are the following steps to move towards Event Sourcing? This book will answer all these questions and many more, guided through lots of practical examples. Brought to you by the same authors behind “Domain-Driven Design in PHP”.

Carlos Buenosvinos, Christian Soronellas, and Keyvan Akbary

This book is available at http://leanpub.com/cqrs-by-example

This version was published on 2024-09-02

Contents

Preface

Domain-Driven Design in PHP

In 2014, after two years of reading about and working with Domain-Driven Design, Carlos and Christian, friends and workmates, traveled to Berlin to participate in Vaughn Vernon's Implementing Domain-Driven Design Workshop[1]. The training was fantastic, and all the concepts that were swirling around in their minds before the trip suddenly became very real.

Around the same time, php[tek][2], an annual PHP conference, opened its call for papers, and Carlos sent one about Hexagonal Architecture. The organization rejected his talk. Months later, Eli White — of musketeers.me[3] and php[architect][4] fame — got in touch with him. Eli was wondering if he was interested in writing an article about Hexagonal Architecture for the magazine php[architect]. In June 2014, *Hexagonal Architecture with PHP* was published in the magazine.

In late 2014, Carlos and Christian talked about extending the article and sharing all their knowledge of and experience in applying Domain-Driven Design in production. They were very excited about the idea behind the book: helping the PHP community delve into Domain-Driven Design from a practical approach. At that time, concepts such as Rich Domain Models and framework-agnostic Applications weren't so prevalent in the PHP community. In December 2014, Carlos and Christian started to work on the already well-known Domain-Driven Design in PHP[5] book.

Around the same time, in a parallel universe, Keyvan co-founded Funddy, a crowdfunding platform for the masses built on top of the concepts and building blocks of Domain-Driven Design. Domain-Driven Design proved itself effective in the exploratory process and modeling of building an early-stage startup like Funddy. It also helped handle the complexity of the company, with its constantly changing environment and requirements. After connecting with Carlos and Christian, they discussed the book and Keyvan proudly signed on as the third writer.

Together, we wrote the book we wanted to have when we started with Domain-Driven Design — full of examples, production-ready code, shortcuts, and recommendations based on our experiences of what worked and what didn't for our respective teams. We arrived at Domain-Driven Design via its building blocks, which is why the book orbited around what are today known as Tactical Patterns. And in this book, you'll learn how to use and implement them. In September 2016, after a couple of years of tough work Domain-Driven Design in PHP[6] was finally published online on Leanpub. A year later, in 2017, the book became a physical reality via PackPub[7] and Amazon[8].

[1] https://idddworkshop.com
[2] https://tek.phparch.com
[3] http://musketeers.me
[4] https://www.phparch.com
[5] https://leanpub.com/ddd-in-php
[6] https://leanpub.com/ddd-in-php
[7] https://www.packtpub.com/application-development/domain-driven-design-php
[8] https://www.amazon.es/Domain-Driven-Design-English-Carlos-Buenosvinos-ebook/dp/B06ZYRPHMC

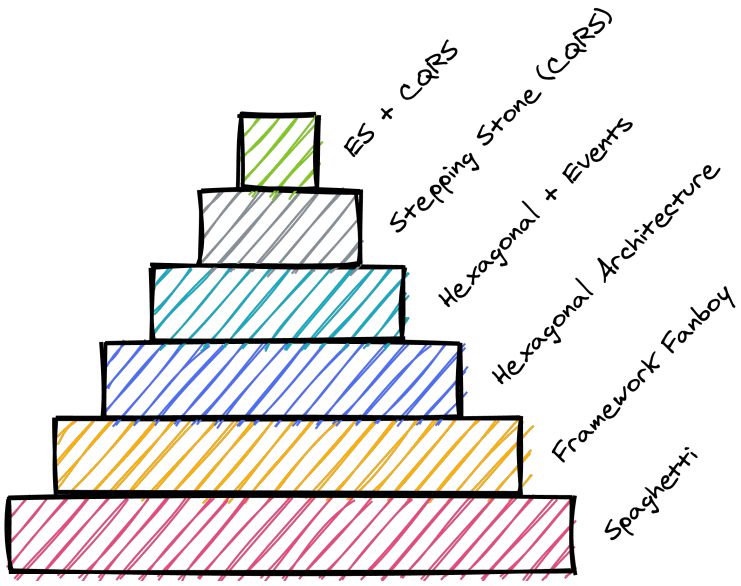
Needless to say, we were heavily inspired by Vaughn Vernon's *Implementing Domain-Driven Design*[9] book (*the Red Book*), and Eric Evans' original book, *Domain-Driven Design: Tackling Complexity in the Heart of Software*[10] (*the Blue Book*). We can't recommend these books enough.

Domain-Driven Design Acceptance Has Rapidly Grown

As Domain-Driven Design authors and early adopters, we're thrilled to see how Domain-Driven Design has taken off, and how its building blocks and its philosophy have permeated the community and become standard. We love how modern architectures such as Hexagonal Architecture, Ports and Adapters, Onion Architecture, and Clean Architecture have empowered developers to build Applications that are easy to evolve, test, and maintain. It's been pleasing to see the trend of using Domain Events as a communication pattern between different Applications and business boundaries.

Due to some characteristic limitations while scaling Hexagonal Architecture Applications, we thought it'd be a great idea to get deep into Greg Young[11]'s work on CQRS, heavily inspired by Bertrand Meyer[12]'s Command-Query Separation (CQS)[13] principle. We highly recommend Greg's workshops and courses on Domain Driven Design, CQRS, and Event Sourcing[14].

In 2017, Carlos presented the Buenosvinos Maturity Model at the Polycon and SymfonyCon Cluj conferences. This model describes the different maturity levels when architecting Applications:



Buenosvinos Maturity Model

[9] http://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577
[10] http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215
[11] https://twitter.com/gregyoung
[12] https://en.wikipedia.org/wiki/Bertrand_Meyer
[13] https://en.wikipedia.org/wiki/Command%E2%80%93query_separation
[14] http://subscriptions.viddler.com/GregYoung

With this model, you can self-assess in which Architectural Style you find yourself and determine what actions to take to move to the next level.

In our experience, companies successfully using Hexagonal Architecture experience lots of benefits against framework coupling or spaghetti code: better semantics and meaningful code, less coupling, and easier testability. However, we've seen that many companies experience limitations that aren't that easy to solve — issues such as performance and dependency management at scale. Here's where this book comes into play.

CQRS by Example

Four years have passed since our first publication. We established what we think is a solid ground for Domain-Driven Design building blocks and Hexagonal Architecture. It's now time to explore how to solve the limitations some projects may face at scale. We think CQRS is the next natural step forward, and we want to guide you through it.

CQRS is usually discussed along with Event Sourcing, a pattern where the Application state is a Projection of the Domain Events that happen through its lifetime. Event Sourcing relies on having a stream of Domain Events to reconstitute state. Triggering Domain Events is a simple addition to operations that modify the state of the Application, such as Commands; however, querying the system now requires some unique mechanisms that aren't that trivial to implement. We can say that in Event Sourced systems, CQRS is mandatory for generating the required information to be able to query later, but the opposite isn't true. We can develop a CQRS system without Event Sourcing, so worry not: This book won't cover Event Sourcing.

In this book, we'll explore Hexagonal Architecture drawbacks, and we'll dive into CQRS by exploring plenty of real examples you can use in your projects. Even though code examples are written in PHP, the patterns and techniques described in this book apply to any programming language and likely any paradigm you may be using.

We can't thank you enough for purchasing this book and being an active contributor to Domain-Driven Design, Hexagonal Architecture, and CQRS!

Who Should Read This Book

If you're a developer, architect, or tech lead, we highly recommend this book. It'll provide you with some pretty interesting tools for your daily toolbox and may reveal an alternative architectural approach for your application. If you don't have much experience, getting into CQRS and modern architectural patterns may prepare you for what's to come throughout your career. For the average reader, understanding the benefits of CQRS and the boundaries between your framework and your Application may be crucial for writing code that's easier to maintain in the real world (e.g. framework migrations, testing). Experienced readers will definitively have some fun exploring Projections and read layers to increase the performance of Applications.

Additionally, the book is loaded with tons of details and examples, such as how to properly design and implement all the building blocks of CQRS — including Commands, Command Handlers, Command Buses, Queries, Query Handlers, Query Buses, Domain Events, Event Buses, Projections, Read Thin Layers, and more.

Summary of Chapters

The book is arranged with each chapter exploring a separate tactical building block of CQRS. It also includes an introduction to CQRS, Domain-Driven Design, and the example project we'll use throughout the book, Cheeper, which is a Twitter clone that's experiencing some issues. Each section below outlines the questions that the corresponding chapter answers.

Chapter 1: CQRS and Domain-Driven Design

What is Domain-Driven Design? What role does it play in complex systems? Is it worth learning about and exploring? What are the main concepts a developer needs to know when jumping into it? How does CQRS relate to Domain-Driven Design?

Chapter 2: A Journey Toward CQRS

What's the foundation of CQRS? What was the context in which CQRS was created? What problems does CQRS solve? In which scenarios is CQRS useful and in which ones does it not pay off?

Chapter 3: Anatomy of CQRS

What are the main building blocks of CQRS? What is the Command Side? What is the Query Side? What are the overall strategies to keep the Read Model and the Write Model in sync? What are some real use cases, their main components, and flows of information?

Chapter 4: Command Side and the Write Model

What is a Command? What about a Command Handler? Why are they important? What's the role of the Command Bus? What are Async Commands and Sync Commands?

Chapter 5: Query Side and the Read Models

What is a Query? What is a Query Handler? Do we need a Query Bus? What is a Read Thin Layer? What are the properties of a good Read Model? How many Read Models can I have? What is a Projection?

Chapter 6: Synchronizing the Write and Read Models

What options do I have to sync the Write Model and the Read Models? Do I need to sync them? Can I use strategies other than messaging? Is messaging the best approach?

Chapter 7: The Full Picture

How can I put into practice all the previously learned concepts? How many asynchronous steps are needed for simple use cases? How many for complex ones?

Chapter 8: Optimizations and Edge Cases

How can I optimize the build time of Projections? What challenges will I face when dealing with Events? How can I recover from duplicate or lost messages?

Chapter 9: CQRS and Event Sourcing

What heuristics do I need to consider using CQRS? How do I move into Event Sourcing? What are the criteria to decide if I should stop at CQRS or move forward?

Chapter 10: Demo Time

How can I demo multiple use cases, understanding in detail what's happening in every step? How do I interact with an Application? How do I consume synchronous and asynchronous Commands? How do I consume Events? How do I consume Projections?

Code, Typos, and Examples

There's an organization at GitHub called *DDD Shelf*[15]. In this organization, you'll find *Cheeper*[16], a Twitter clone application used throughout the book to teach you all the concepts around CQRS. You'll also find additional snippets and tools.

If you find any issue or fix or have a suggestion or comment while reading this book, you can open an issue in the *CQRS By Example Book Issues*[17] repository. We fix them as they come in. If you're interested, we encourage you to follow our projects and provide feedback.

[15] https://github.com/dddshelf
[16] https://github.com/dddshelf/cheeper-ddd-cqrs-example
[17] https://github.com/dddshelf/cqrs-by-example-book-issues

Acknowledgements

First of all, we'd like to thank all our friends and family. Without their support, writing this book would've been an impossible task. You're wonderful, and part of the success of this book is also yours.

It's no surprise our design skills are quite limited. Our beautiful cover was crafted by our friend and master designer Carlos Tallón[18], and we owe him a huge thanks.

We are three Spaniards who wrote a book in English, so if you'd guess our English is far from perfect, you'd be correct. The final version of this book has been edited by a professional copy editor, Natalye Childress[19]. She has done a great deal of work fixing typos, rewording our phrases, and rethinking the book structure. Thank you so much. Our book is far easier and more enjoyable to read now.

Thanks to Marco Pivetta[20] for his foreword and feedback on the book. The book is now better thanks to his contributions.

A special mention to Greg Young: Your work has been an incredible source of information and inspiration for us.

Last but not least, we'd like to express our gratitude to all the people who have reported issues, made suggestions, and otherwise contributed to our GitHub repository[21]. To all of you, thank you. You've helped us make this book better. More importantly, you've helped the community grow and helped other developers be better developers. Thanks to Mohammad Emran Hasan, Ashish K. Poudel, Eduardo Anton Santa-Maria, Mohamed Cherif Bouchelaghem, Jose Samonte, Ruben Calvo, Sami Jnih, Diego García, and David Navarro.

[18] https://twitter.com/carlostallon
[19] http://www.natalye.com/
[20] https://twitter.com/Ocramius
[21] https://github.com/dddshelf/cqrs-by-example-book-issues

About the Authors

Carlos Buenosvinos

Carlos is an Extreme Programmer (XP) and DevOps with more than 20 years of experience in developing Web and Mobile Applications. For the last ten years, he has played various leading roles such as Tech Lead, VP of Engineering, and CTO. He has mentored engineering and product teams of up to 150 members in multiple different markets such as E-commerce, E-Learning, Payment Processing, Classifieds, and Recruiting Market.

As an employee and consultant, he has contributed to the success of start-ups and well-established brands. Some examples are SEAT, NewWork/XING, Atrápalo, GMV, PCComponentes, Cash Converters, Emagister, O2O, Opositatest, Techpump, Packlink, eBay, Lowpost, Vendo, Riplife, and many more.

He is the happy creator of Ansistrano[22], the most starred Ansible Galaxy role. He is also the author of the book Domain-Driven Design in PHP[23]. He is also a conference speaker and organizer of the DevOps Barcelona Conference[24] and the PHP Barcelona Conference[25].

His main areas of expertise are Agile Team Management (Scrum and Kanban), Best Development Practices (Extreme Programming, Domain-Driven Design, and Microservice Architectures), and Digital Transformation (Agile, XP, and DevOps).

You can follow him at Twitter[26], at his blog[27] or at GitHub[28].

Christian Soronellas

Christian is an Extreme Programmer and has over 15 years of experience helping tech companies succeed from a broad variety of roles, from Software Engineer to CTO. He has helped companies such as Privalia, Emagister, Atrápalo, Enalquiler, PlanetaHuerto, PcComponentes or Opositatest. He is the author of the book Domain-Driven Design in PHP[29] as well as a conference co-organizer of DevOps Barcelona Conference[30] and PHP Barcelona Conference[31]

[22] https://ansistrano.com
[23] https://leanpub.com/ddd-in-php
[24] https://devops.barcelona
[25] https://php.barcelona
[26] https://twitter.com/buenosvinos
[27] https://carlosbuenosvinos.com
[28] https://github.com/carlosbuenosvinos/
[29] https://leanpub.com/ddd-in-php
[30] https://devops.barcelona
[31] https://php.barcelona

You can follow him at Twitter[32] or at GitHub[33].

Keyvan Akbary

Keyvan is an Engineering Leader and programmer with more than 15 years of experience crafting products customers love and helping teams succeed. He understands technology as a medium for providing value, not the end itself. He has a passion for Distributed Systems, Software fundamentals, SOLID principles, Clean Code, Design Patterns, Domain-Driven Design and, Testing; as well as being a sporadic Functional Programmer. For the last 7 years he has also focused on growing teams in high scaleup product companies, advocating for customer-centric product development, Extreme Programming, DevOps, Lean, and Kanban.

He has worked on countless projects as a freelancer, on video streaming at Youzee, tradesman marketplace at MyBuilder, in addition to founding his own crowdfunding startup Funddy, and leading FinTech teams at Wise. Currently, he is leading engineering in the ride-hailing space as Head of Engineering at Cabify.

He is also the author of "Domain-Driven Design in PHP"[34] and "The Manager's Manual"[35].

You can follow him at Twitter[36], at LinkedIn[37], at his blog[38] or at GitHub[39].

[32] https://twitter.com/theUniC
[33] https://github.com/theUniC/
[34] https://leanpub.com/ddd-in-php
[35] https://leanpub.com/the-managers-manual/
[36] https://twitter.com/keyvanakbary
[37] https://www.linkedin.com/in/keyvanakbary/
[38] https://keyvanakbary.com
[39] https://github.com/keyvanakbary/

Anatomy of CQRS

In previous chapters, we explained the relationship between Domain-Driven Design and CQRS and other Architectural Styles, focusing on Hexagonal Architecture as a foundation for CQRS. Now, it's time to provide a proper overview of CQRS: what it is, what the moving parts are, which benefits and drawbacks it has, and what the valid use cases for such an interesting Architectural Style are.

Cheeper Use Case Analysis

If you were to create Cheeper from scratch, you might end up with your favorite Architectural Style, i.e. the one you're most comfortable with. Maybe you'd choose Hexagonal Architecture with all the Domain-Driven Design Tactical Patterns like Aggregates, Repositories, and Application Services. In our experience, the code, style, and organization possibilities are endless.

In spite of this, we're almost certain that everyone would end up with the same or a very similar Data Model. Such a model would likely be in fifth normal form (5NF)[40] to reduce redundancy, as we've been told data redundancy is a bad thing.

Starting with a version of Cheeper developed using Hexagonal Architecture and with a Data Model following the 5NF, let's compare some hypothetical needs and the database load for some Cheeper use cases. There are two primary factors to compare: the number of requests received, and performance. These will be enough to understand the overall system overhead; a use case with good performance and low traffic will have a low overhead, and a use case with bad performance that's called frequently will have a high overhead.

Use Case	Type	# Requests	Performance	Overall System Overhead
Sign Up Author	Write	Very Low	Good	Very Low
Follow Author	Write	Low	Good	Low
Post Cheep	Write	Medium	Good	Low
Count Author Followers	Read	High	Bad	High
Read a Cheep and Responses	Read	High	Bad	High
Fetch Author Timeline	Read	Very High	Bad	Very High

Sign Up Author, *Follow Author*, and *Post Cheep* are simple use cases to implement with a good overall performance. Their implementation consists mainly of creating a new instance of a specific Entity (Author, Follow, or Cheep) and storing it in the database: a bunch of `SELECT` operations ending with an `INSERT`, and we're done. These simple use cases aren't called often either. Many users will end up using *Sign Up Author* or *Post Cheep*, but if we were to compare these actions against *Fetch Author Timeline* or *Count Author Followers*, it's pretty clear these other use cases are invoked orders of

[40] https://en.wikipedia.org/wiki/Fifth_normal_form

magnitude less.

In contrast, *Fetch Author Timeline* and *Count Author Followers* are frequently used. If you're a Twitter user, stop and reflect on how you use it; most of the time is spent reading Timelines. From time to time, you'll check on Authors, and for that, you'll need to display the Follower count too. Due to this high load of requests, performance gets worse. These use cases turn into database Queries that take time and are complex in terms of `JOIN`s. As a result, the database will not only require forever-increasing computational power that scales with the user base, but write and read operations will affect one another negatively in terms of performance.

On Twitter, this cross-table aggregation is difficult, if not impossible, to run. The data is too big to be held by a single database instance, and it's sharded into different servers, making `JOIN` infeasible.

The 5NF aims to make things more consistent. This is mainly enforced by `INSERT`, `UPDATE`, and `DELETE` operations. However, it isn't ideal for reads and operations such as `SELECT`, because consistency benefits writes and not reads. This is the main reason applications end up with slow and complex join-aggregated-like Queries.

A way to alleviate performance issues on reads is by using a cache. Keep in mind that using a cache brings other problems to the table. In an application like Cheeper, it'd be difficult to have a high hit ratio for people Timelines, as these change frequently. We can keep the cache for minutes, but this will hurt the experience; people won't be able to see new Cheeps unless the cache refreshes. The cache could be invalidated at the right time too — like when you follow a new Author, or post or delete a Cheep — to squeeze the best possible experience out of it, but it'll be complex. Even then, Timelines are complicated beasts, and they'll require hitting the database at some point. These hits will have to aggregate data from a lot of places, making it difficult to scale at the expense of user experience.

Given this, maybe we can consider an alternative approach to improving performance in use cases like *Count Author Followers* or *Fetch Author Timeline*. Let's explore how we'd face these issues using CQRS.

Cheeper à la CQRS

Starting with the challenge of improving overall performance — specifically, the *Count Author Followers* and *Fetch Author Timeline* use cases — let's see how the flow would go through a typical CQRS application.

Reducing the Number of Hits to the Storage

As a rule of thumb, for every use case, we should always try to keep the number of hits to the storage mechanism to a minimum. These aren't only slow, but they'll be a challenge to scale. For any given

use case, how many requests would you like to perform to the storage? Ideally one single request. How many `JOIN`s would you like this Query to perform? The fewer the better. That means no `JOIN`s, no `WHERE`s, nothing. How would you like this Query to appear? For viewing a Timeline, just a plain lookup Query by an Author identifier with all the information flattened and ready to be presented to the user would be great.

The terminology "ready to be presented to the user" is important here. While some read use cases would require a tabular structure, like listing the Cheeps of a Timeline, others would require a nested structure, like unfolding a single Cheep with its interactions (e.g. Cheep information with a nested list of Cheeps). In Domain-Driven Design and CQRS, these read use cases are known as Queries.

Choosing the Right Storage

Considering that you only want to perform a single Query, what kind of storage would you choose to store such information? Well, it depends on the read use case. In the case of the Timeline, it could be a table hosted under the same database for your application or even in an external database. For viewing a Cheep, maybe document-oriented storage like Elastic or Mongo would play well. For the Followers counter, Redis could be a nice option. The takeaway is that even though Relational databases cover most of our Data Model needs, we shouldn't limit ourselves to a single storage mechanism. Every read use case may potentially have an optimized storage type and Data Model.

Filling the Storage

Who or what fills these secondary storages? There must be some *process* responsible for preparing and transmitting the data. There are multiple options for implementing such a process; a simple one could be to have a scheduled cron job process *projecting* every Author Timeline into Elasticsearch, or *projecting* the count of Followers for every Author into Redis. The problem with a cron-like job is that it's a bit of overkill, as there will be times it won't have anything to process, and there will be times it'll have too many things to do. What's worse is the cadence for execution will delay spreading changes in the same way a cache would. We need a perfect balance. Syncing more often than the speed at which Timelines change means we'll be wasting CPU. Syncing less often means changes to Timelines will take time to be indexed into Elasticsearch, and customers will see stale information (eventual consistency). We need a precise process to be triggered at the right time, just when the changes that affect the data happen.

Let's assume for now that this process exists: a process that calculates the Author Timelines, and another process that calculates and stores the Follower count for Authors.

Triggering the Process

Who or what triggers this kind of process? A message or Event will do. For Author Timelines, we can trigger an Event every time there's a new Cheep, so the previously defined process can update or fully rebuild the Timeline of the affected Authors. The same mechanism would apply for counting the

number of Followers, reacting upon "Author followed" and "Author unfollowed" Events. In Domain-Driven Design, we call these Domain Events; these represent something that happened in the past that's relevant in terms of the business operation. In our Cheeper application, there could be Events such as "Cheep posted," "Author followed," "Bio updated," "Cheep answered," and more.

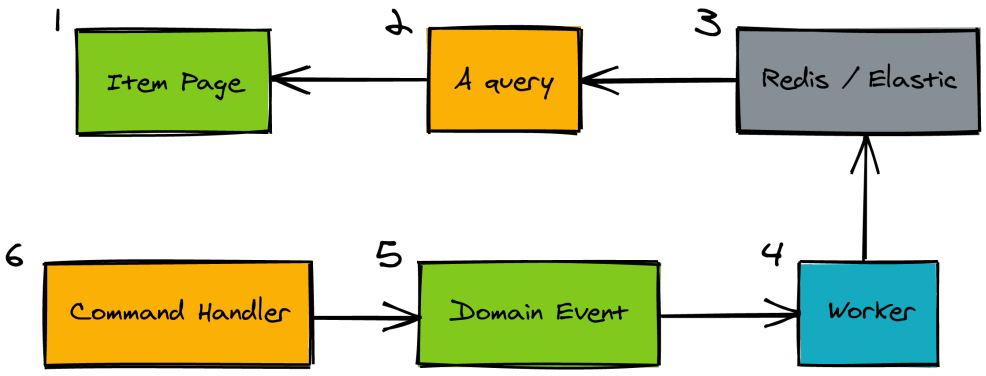
Emitting the Event

For the *Fetch Author Timeline* Query use case, we'll have to trigger Projections each time a new Cheep is posted, edited, or deleted. So, who or what emits the necessary Domain Event?

The answer is the corresponding write use cases: *Post Cheep*, *Update Cheep*, and *Delete Cheep*. These write use cases are what we call Commands in Domain-Driven Design and CQRS. The Domain Events that trigger the process for projecting the data "ready to be read" for the *Count Author Followers* use case are "Author followed" and "Author unfollowed." These Events will be triggered by the corresponding Commands: Follow Author and Unfollow Author. But in the end, who's invoking these Commands? Our customers. Users may use the UI of a client for our API or the website to post a new Cheep. Customer actions invoke their corresponding Commands and then trigger a specific Domain Event, and these Events also trigger a process (also known as a worker, process manager, or Event Saga) that ends up projecting the data in an optimal format and storage, ready to be consumed from the Query side, allowing all the required information to be fetched with a single request.

Circle of Life

This is, in essence, what CQRS is all about: For every Command, there's a Domain state change that triggers Events, and these Events trigger the syncing process between the Write Model and the Read Models that allow optimal querying for our application. The great thing about this approach is that it doesn't matter how much additional information we append into a specific Query; the Read Model is always optimized to answer blazingly fast. In comparison, with a normal 5NF-based application, every new piece of information in a read use case will require an additional database Query or `JOIN`, further degrading the performance of the system. More features, less performance. Does this sound familiar?



CQRS Flow

Restaurant Booking System

Once we worked for a company that had a classified restaurant website. You were able to search for restaurants, choose one, and show everything customers would expect to read on the restaurant page. This included basic information about the restaurant, menus, offers, pictures, customer reviews, and more. Considering all the different pieces of information, how many Queries do you think the application made to the database on this single page? Well, around 620 Queries in the development environment and 250 in the production environment with all the caching enabled. 250 Queries! 250 Queries with tons of complex `JOIN`s for a single restaurant page! As an exercise, starting from the Query that gets the information for the restaurant page, try to find the Domain Events and Commands that will trigger the Projections for optimal querying.

We like this approach to explain how the flow of CQRS works. It's important to understand that Queries (the read use cases) and Commands (the write use cases) are two sides of the same coin; one of them doesn't make sense without the other. With this simple and powerful concept, we optimize the application for great performance without losing maintainability. Now, it's time for a deep dive into CQRS.

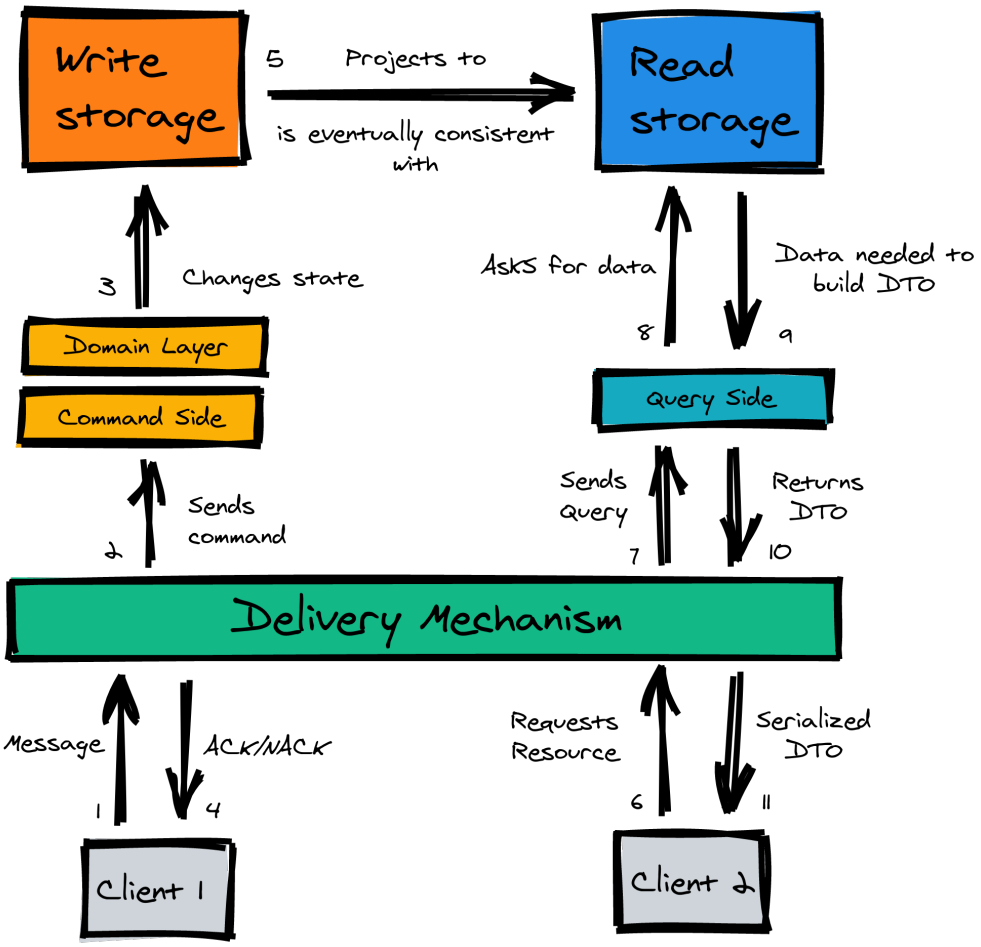
CQRS Overview

CQRS stands for **Command-Query Responsibility Segregation**. Coined and originally developed by Greg Young, it's an architectural pattern built upon the shoulders of Hexagonal Architecture and the Command-Query Separation[41] principle by Bertrand Meyer. At its heart, CQRS defends having different Data Models for writes and reads. With this, Greg Young proposes making all the Application Views an Infrastructure concern — turning the entire Delivery Mechanism into a Port, and each View into an Adapter. The Model should be split into two different parts:

- The **Write Model** — Also known as the **Command Model** or the **Write side**, it performs the writes and takes responsibility for the Domain Logic.
- The **Read Model** — Also known as the **Query Model** or the **Read side**, it takes responsibility for the reads within the application and treats them as something that should be outside of the Domain Model, i.e. an Infrastructure detail.

Our users will interact with either side, depending on what type of use case they need to perform. They'll make use of the Query Side while requesting information from the Domain, and they'll make use of the Command Side when they perform some action that modifies the state of the application.

[41] https://en.wikipedia.org/wiki/Command%E2%80%93query_separation



Typical CQRS Architecture

As you'll discover while reading this book, one of the biggest challenges while implementing CQRS is the synchronization of the Write side with the Read side.

Commands and Queries

Continuing with the two Models, in CQRS, we can effectively classify our application requests as:

- **Commands** — Requests that change the state of a system but don't return a value.
- **Queries** — Requests that return a result and don't change the state of the system.

In Cheeper, and in almost all the applications you've developed and used, you can classify use cases either as ones that modify the state of the Domain, or ones that read the state. Posting a Cheep, updating an Author bio, or following another Author are examples of requests that modify the state of the application, i.e. Commands. Loading your Timeline, counting the number of Followers, or fetching an Author bio are examples of requests that read the application state, i.e. Queries.

This classification isn't something new. As mentioned before, this split was already proposed as a design principle by Bertrand Meyer in his book *Object-Oriented Software Construction*[42], and it's called *Command Query Separation (CQS)*.

Command-Query Separation (CQS)

Asking a question should not change the answer - Bertrand Meyer.

According to Wikipedia[43], "Every method should be either a *command* that performs an action, or a *query* that returns data to the caller, but not both."

We can see CQRS as scaling CQS to the entire application.

Commands and Queries will have a code representation in our application. They'll be implemented as Data Transfer Objects (DTOs):

```
final class PostCheepCommand
{
    //...
}
```

```
final class TimelineQuery
{
    //...
}
```

Using Commands and Queries offers many benefits, including better decoupling from the framework and the chosen Infrastructure, more meaningful and semantic code in terms of user experience, and improved possibilities for adding asynchronous operations.

Other CQRS Components

In CQRS, there aren't only Commands and Queries, but a handful of other concepts and components that make the entire flow work. Let's take a quick look:

- *Queries* — Requests that represent which information the user wants to get, and the parameters required to fetch it.
- *Commands* — Requests that represent the action the user wants to perform with all the parameters required.

[42] https://www.amazon.com/dp/0136291554/ref=cm_sw_r_tw_dp_U_x_g56YEbZG065YT
[43] https://en.wikipedia.org/wiki/Command%E2%80%93query_separation

- *Domain Events* — Messages that represent something that happened in the Domain that will make other CQRS components react. They're usually triggered by Command Handlers, and they're key for synchronizing the Write and the Read Models with one another.
- *Projections* — Specialized, persisted, and optimal Read Models generated for every Query.
- *Query Handlers* — The mechanism responsible for accessing and preparing the information the user wants to get for a specific Query.
- *Command Handlers* — The mechanism responsible for performing the actual business logic for a specific Command.
- *Event Handlers* — The mechanisms responsible for listening to Domain Events and triggering Commands for orchestrating long-running processes or triggering Projections. These are also known as Process Managers or Sagas.
- *Projection Handlers* — The mechanisms responsible for generating the Projections, also known as Projectors.

Customers will interact with Cheeper most likely through a client, such as a Single-Page Application (SPA) or a Mobile Application. These frontend clients will interact with an API with the help of a framework of your choice. In turn, this framework will transform these API requests into Commands or Queries, depending on the use case, and it'll invoke the corresponding Command Handler or Query Handler and return the appropriate results if necessary.

Behind the scenes, the Domain Events — triggered by the affected Command Handlers and Entities — will be handled by one or multiple Event Handlers that will invoke Projection Handlers that will in turn generate the Read Models optimized for the Query Side. Query requests will simply go and fetch the data ready to be presented to customers.

Two Sides of the Same Coin

Every time a Command is **sent** to the Write Model, the Domain Model is evaluated, and the resulting state changes are persisted into the Write Storage, which is different from the Read Model storage. Why do we all of this? Doesn't it sound overengineered? Well, we have different needs in our system, and having these two different Models is quite convenient:

	Consistency	Storage	Scalability
Write Model	Immediate consistency	Data is saved, usually in a normalized form	Small number of transactions
Read Model	Eventual consistency	Data can be denormalized when generated / updated, so there's not too little performance cost when queried.	Usually, there are a lot more transactions, so scalability here is critical.

The Write side and Read side are two sides of the same coin. The main goal is to get the most performance and composition without compromising maintenance. Let's explore these a bit more.

The Command Side

The Command side is the one that sends messages to the Domain layer. Once it's split from the Query side and freed from reading concerns, Aggregates suddenly no longer need to expose internal state, and Repositories have no Query methods aside from the usual `ofId`. We've fixed our anemic problem; just by extracting Queries, we're left with rich Domain Models.

Commands are somewhat specialized DTOs that carry relevant information needed to fulfill specific Domain operations with semantic names named after the Ubiquitous Language. As Commands may travel through many types of channels, it's better to use primitive types instead of more complex types. We're effectively transitioning from asking the Domain for stuff to instructing it to execute Commands on behalf of the client.

Commands and Queries need handlers to be interpreted and executed. Command Handlers are responsible for executing the Domain behavior associated with the Command sent. Command Handlers don't perform the Domain Logic; they only transform the outside world data into Domain concepts and coordinate these concepts to activate the intended Domain Logic. We recommend the use of invokable classes for Command Handlers, as this enforces a single handler interface for all Commands, and it allows you to think about the appropriate name in terms of the Ubiquitous Language.

Separating Commands from Query requires more work, but the performance benefits make up for it. In many cases, separating the two will make it easier to optimize Queries on the Query side than if we'd left them on the Command side. In addition, it lowers the conceptual complexity when working on the Domain Model, as the Command side only has to worry about state changes.

The Query Side

The Query side is all about optimally fetching data to build DTOs that will later be serialized in a format the client understands. Before CQRS, this process was done by loading Aggregates and getting their internal structure to satisfy the UI needs. Due to the impedance mismatch between the Domain Model and the View, the loading of entire Aggregates just to satisfy the UI needs may be extremely inefficient and difficult.

Impedance Mismatch

The object-oriented paradigm, which is what we usually use to represent the Domain, is based on proven engineering principles. The Relational paradigm, which tend to use for our Data Model, is based upon mathematical principles. Because of these underlying principles, the two technologies don't work seamlessly together.

The object-oriented paradigm works with relationships, whereas in the Relational paradigm, you join the data rows in tables.

This impedance mismatch between the Domain Model and the Relational Data Model comes with a big cost. This is what Object Relational Mappers (ORMs) such as Doctrine somehow try to mitigate. However, even if the mapping between the two gets alleviated by ORMs, Developers need to be familiar with both and know how to reconcile their differences.

One of the benefits of having a separate Read Model is that it won't suffer from impedance mismatch, as there's no need to translate from the Data Model to the Domain Model. Rather, data can be accessed directly.

Query Handlers query the Read Model store directly and return the data itself, with no need for transformations or complex mappings. The Delivery Mechanism can then serialize it back to the clients.

Syncing the Command and Query Sides

If Commands trigger Domain behavior on the Write Model, how does the Read Model get notified about such changes? The simplest approach to this is to let both sides write and read from the same storage. The problem with this approach is that we don't get the performance benefits we're looking for. The Read Model isn't optimized for the Query operations, so to retrieve the information, we fall into the multiple Queries and complex `JOIN`s scenario, which was outlined in the example of the restaurant booking system above.

Another option is to treat them as if they were two integrated systems; separate data sources will allow Data Models to be optimized independent of each other. Many well-known integration patterns can help to keep either immediate or eventual consistency between both sides. Every second spent thinking about how to synchronize the Models before the implementation will be well spent. However, changing how things get synchronized afterward will be an expensive undertaking. In this case, there should be some kind of process that takes notifications from the Write Model and updates the Read Model, thereby making the Read Model **eventually consistent** with the Write Model (after a certain time window). There may be a window of time where the UI may present stale information to the customer. Caches behave in a similar way. They don't represent the source of truth, rather, there's a propagation delay that makes them eventually consistent.

These kinds of processes, speaking in CQRS terminology, are called Write Model Projection Handlers, or just Projectors, while the optimized Read Models are called Projections. We *project* the Write Model onto the Read Model. This process can be synchronous or asynchronous, depending on

your needs, and it can be done thanks to another useful Tactical Design Pattern: Domain Events. Write Model Projections take published Domain Events and update the Read Model based on the aggregated information.

Wrapup

We've seen what CQRS stands for and looked at the main elements of CQRS. We also gained understanding as to the purpose of the Write and Read Models and how important it is to keep them in sync. Finally, we got a grasp on how complicated syncing the Models is.

In the upcoming chapters, we'll dive into the specifics of how to send Commands and Queries effectively, and we'll explore what message buses are all about. We'll also follow up on how to synchronize the Write and Read Models, and how to manage the Read Model's eventual consistency.

Demo Time

In this chapter, we'll see how to run Cheeper and follow what we've seen in previous chapters step by step. We'll begin without any data — no Authors, no Cheeps, and no relationships and follows between Authors. We'll see how to Sign Up Authors, make them follow people, and how get the Follower Count and Author Timelines. We'll also see how the data gets stored and propagated through MySQL, RabbitMQ, and Redis.

Getting Started

The best way to get everything running in all kinds of environments — macOS, Windows, or Linux — is by using Docker[44], which is free for individuals. Download it[45] and follow the installation instructions.

Docker Compose and Makefile

All the services required to run Cheeper are run and coordinated by Docker Compose, and you can find the definitions in the `docker-compose.yaml` file. Although Cheeper is a PHP Application that runs with the Symfony PHP framework, it requires MySQL for the Domain Model storage, RabbitMQ as the messaging system, and Redis for Projections. We'll use `Makefile` for command-line tasks.

We'll interact with Cheeper via HTTP requests to its API. Through this chapter, we'll use HTTPie[46] as the HTTP client, but feel free to use curl[47], wget[48], or your favorite HTTP client to send requests to Cheeper. To install `HTTPie` on your system, please follow the installation instructions[49].

Starting the Application

Once Docker and `httpie` are installed, the next step is to clone the Cheeper repository onto your machine:

[44] https://www.docker.com/
[45] http://docs.docker.com/get-docker/
[46] https://httpie.io/cli
[47] https://curl.se/
[48] https://www.gnu.org/software/wget/
[49] https://httpie.io/docs/cli/installation

```
git clone https://github.com/dddshelf/cheeper-ddd-cqrs-example
```

To start the Application in Docker with all its dependencies, you can run:

```
cd cheeper-ddd-cqrs-example
make start
make deps
```

As we're starting the Application for the first time, we'll need to create the database in MySQL, flush all Redis entries, and clear all RabbitMQ queues. We have a Makefile task just for that. Open a new terminal in the project and run:

```
make infrastructure
```

If you need to stop all services, run:

```
make stop
```

Nothing Up My Sleeve

Up to this point, there's no data in our Application — No Authors, no Cheeps, and no following relationships between Authors.

We could try to get the Author Follower Count for a non-existing Author with ID `a64a52cc-3ee9-4a15-918b-099e18b`

```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/followers-counter
```

However, we'd get an HTTP 404 Not Found response:

```
{
    "data": {
        "message": "Author \"a64a52cc-3ee9-4a15-918b-099e18b43119\" does not exist"
    },
    "meta": {
        "message_id": "c7f70ca1-30fe-4072-99dd-8aad781cd386"
    }
}
```

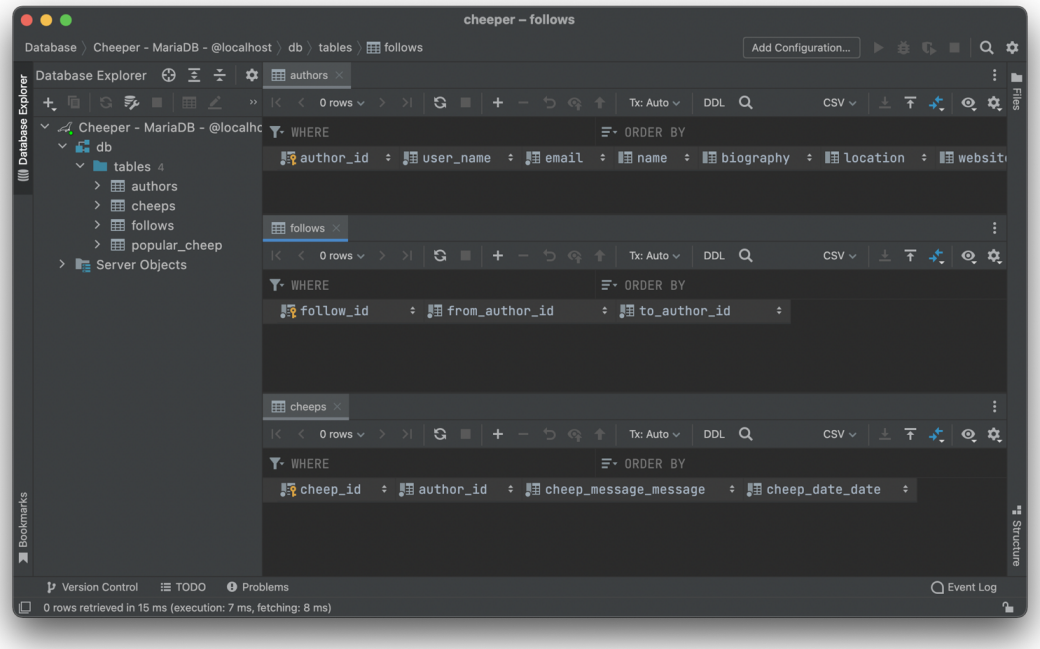
The same thing happens if we try to get a non-existing Author Timeline:

```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/timeline
```

The difference, in this case, is that we decided to return an empty Cheep list instead of an HTTP 404 Not Found response by design:

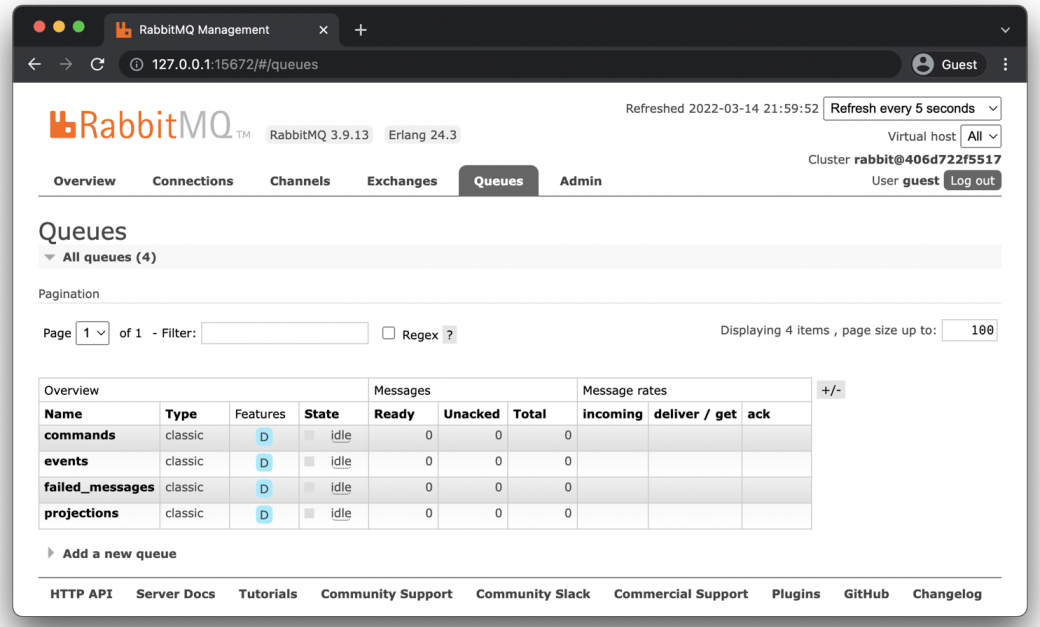
```
{
    "data": {
        "cheeps": []
    },
    "meta": {
        "message_id": "04d6e178-34b9-4788-be0d-90e21c3a5693"
    }
}
```

Although we have the schema initialized in MySQL with all the tables and constraints set up, the database is empty of data.



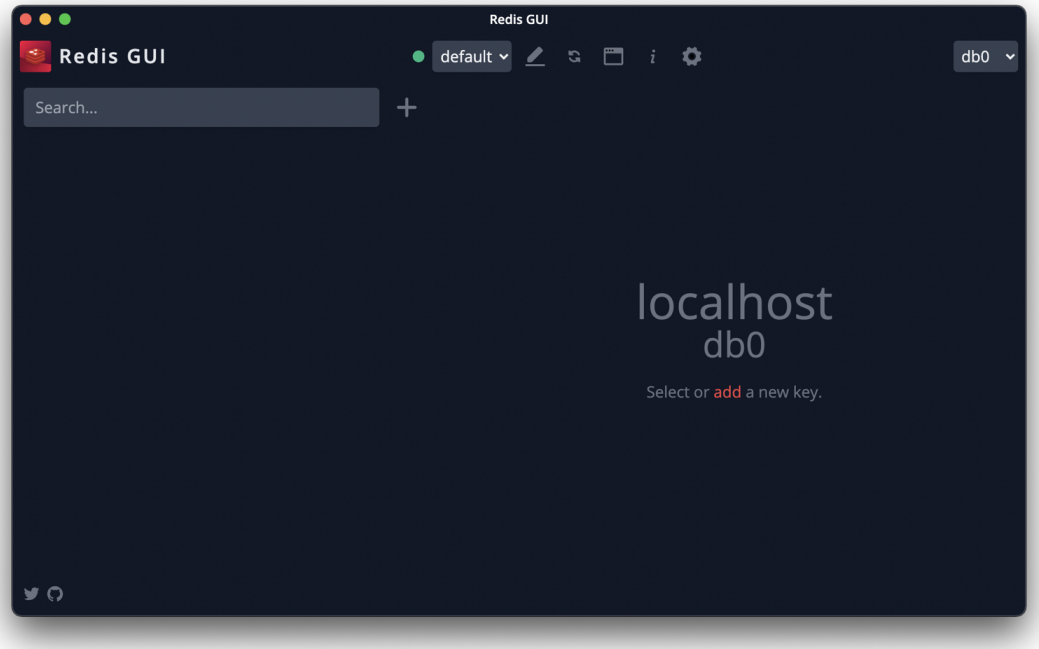
MySQL is empty

RabbitMQ has all the queues ready, but no messages waiting.



RabbitMQ has no messages only the queues

There are no keys in Redis.



Redis is empty

Signing Up New Authors

The way we configured the Author Sign Up Command in Cheeper is to be run synchronously. Once we send an HTTP request to perform the operation and wait for the response, it'll be completed. At the end of the process, an Author will be stored in MySQL without further interactions. The Symfony Controller receives the HTTP requests, creates the `SignUpCommand` with all the necessary information to perform the Command, and finally sends it to the Command Bus. The Command Bus finds the appropriate Command Handler, the `SignUpCommandHandler`, and executes it immediately, doing all the necessary work to Sign Up an Author:

```
http --json --body POST http://127.0.0.1:8000/chapter7/author \
    author_id="a64a52cc-3ee9-4a15-918b-099e18b43119" \
    username="bob" \
    email="bob@bob.com"
```

```
{
    "data": {
        "author_id": "a64a52cc-3ee9-4a15-918b-099e18b43119"
    },
    "meta": {
        "message_id": "8437fb8a-4bdd-49d4-8da1-fb0f4054a3e8"
    }
}
```

```
http --json --body POST http://127.0.0.1:8000/chapter7/author \
    author_id="1fd7d739-2ad7-41a8-8c18-565603e3733f" \
    username="alice" \
    email="alice@alice.com"
```

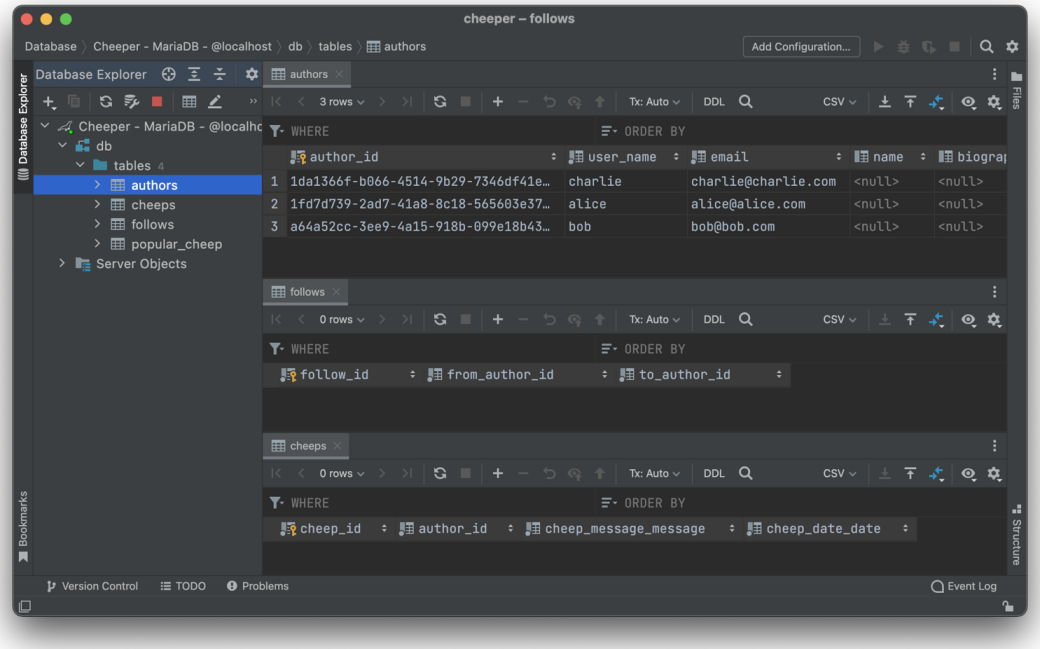
```
{
    "data": {
        "author_id": "1fd7d739-2ad7-41a8-8c18-565603e3733f"
    },
    "meta": {
        "message_id": "35644e59-142f-4bb2-9a3b-7f937f7fc672"
    }
}
```

```
http --json --body POST http://127.0.0.1:8000/chapter7/author \
    author_id="1da1366f-b066-4514-9b29-7346df41e371" \
    username="charlie" \
    email="charlie@charlie.com"
```

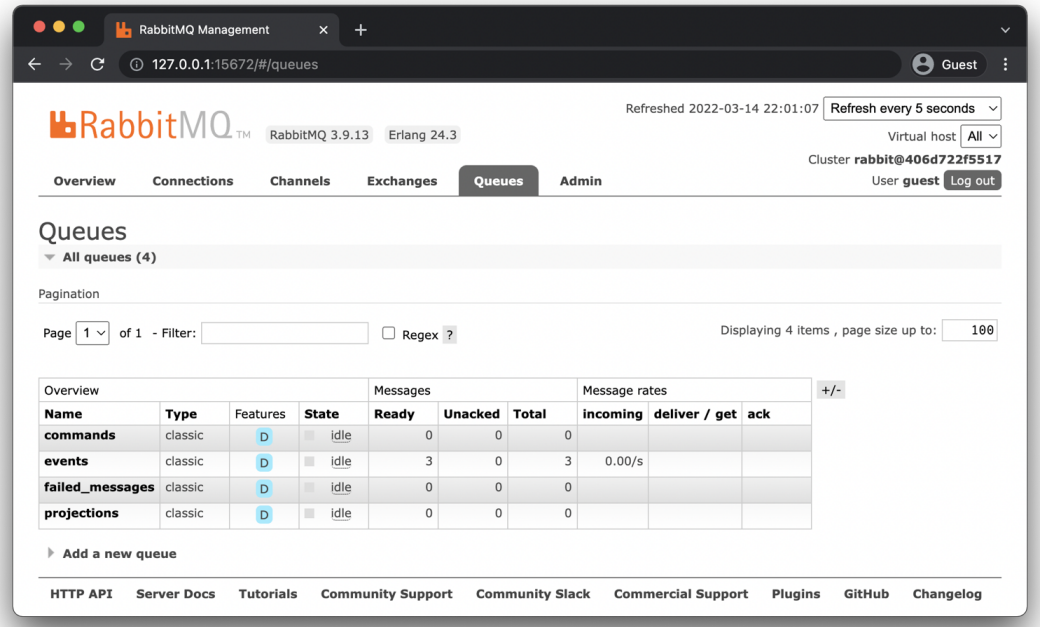
```
{
    "data": {
        "author_id": "1da1366f-b066-4514-9b29-7346df41e371"
    },
    "meta": {
        "message_id": "b5e159bd-38a2-4b22-ac2d-1894efe800d4"
    }
}
```

Through this process, multiple `NewAuthorSigned` Domain Events are raised by the Command Handler, and they end up being stored in RabbitMQ. These Events are waiting to be consumed by

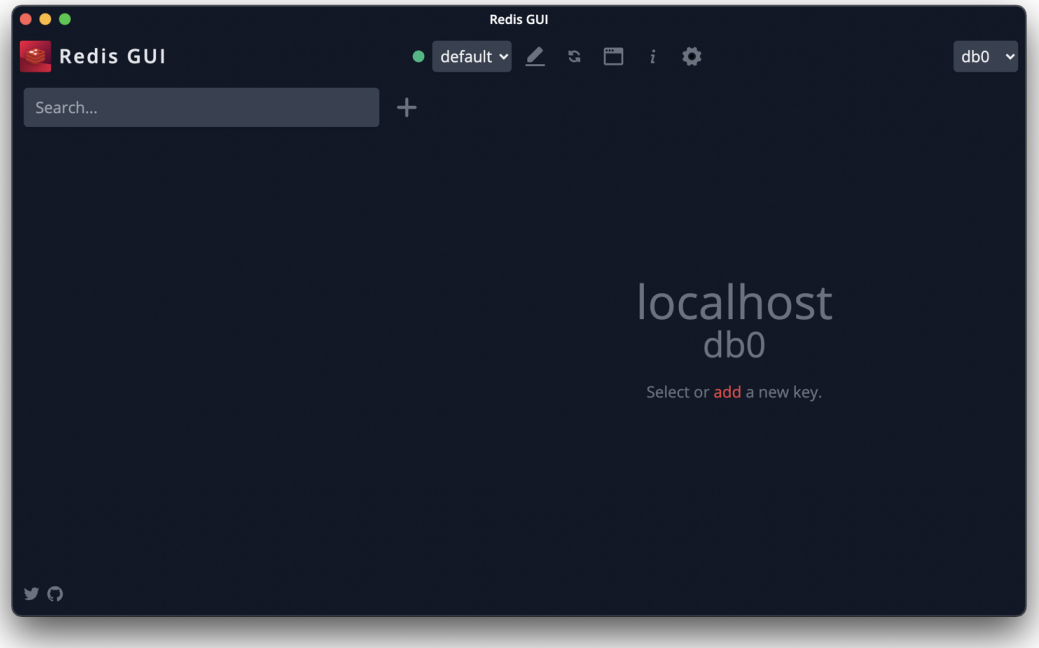
asynchronous processes in a RabbitMQ queue. Remember that these Events are the ones necessary to build the Author Follower Count Projection, so until they get consumed, the Follower Count for these newly created Authors will still return an HTTP 404 Not Found response.



The three Authors in the database



Three NewAuthorSigned Events awaiting to be processed



Redis is still empty

Consuming `NewAuthorSigned` Events

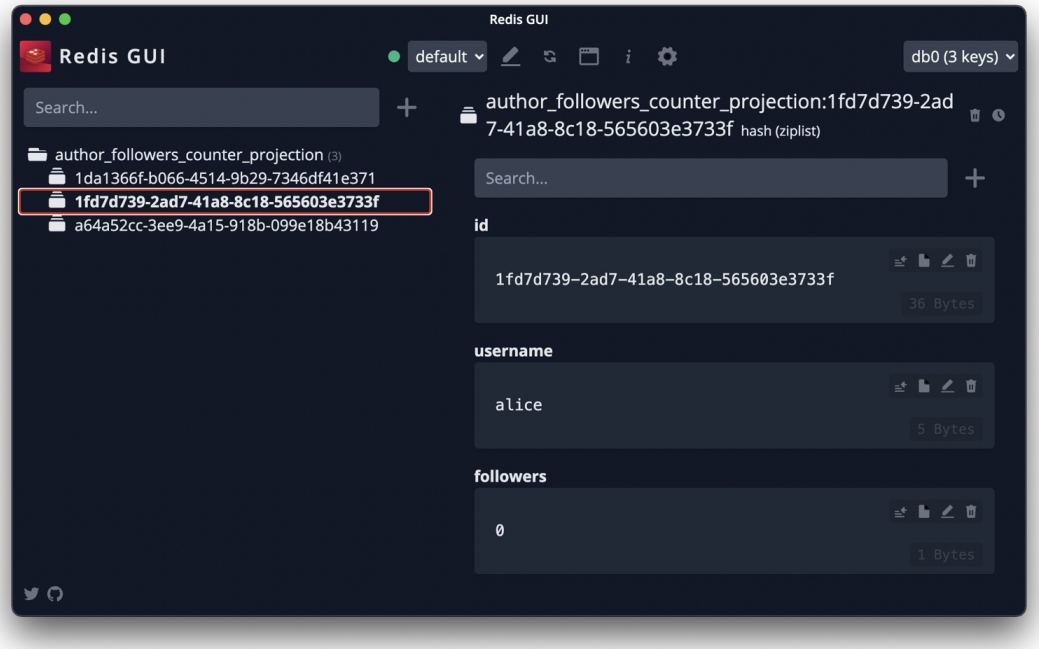
The next step will be to consume the Domain Events pending processing in RabbitMQ. For that, we can go into the `app` Docker service and ask Symfony Messenger to consume the Events from the `events_async` channel:

```
docker compose exec app php bin/console messenger:consume events_async --limit 3 -vv
```

This will return something like what's shown in the following logs. A worker receives the message and delegates its processing on to the appropriate Event Handler:

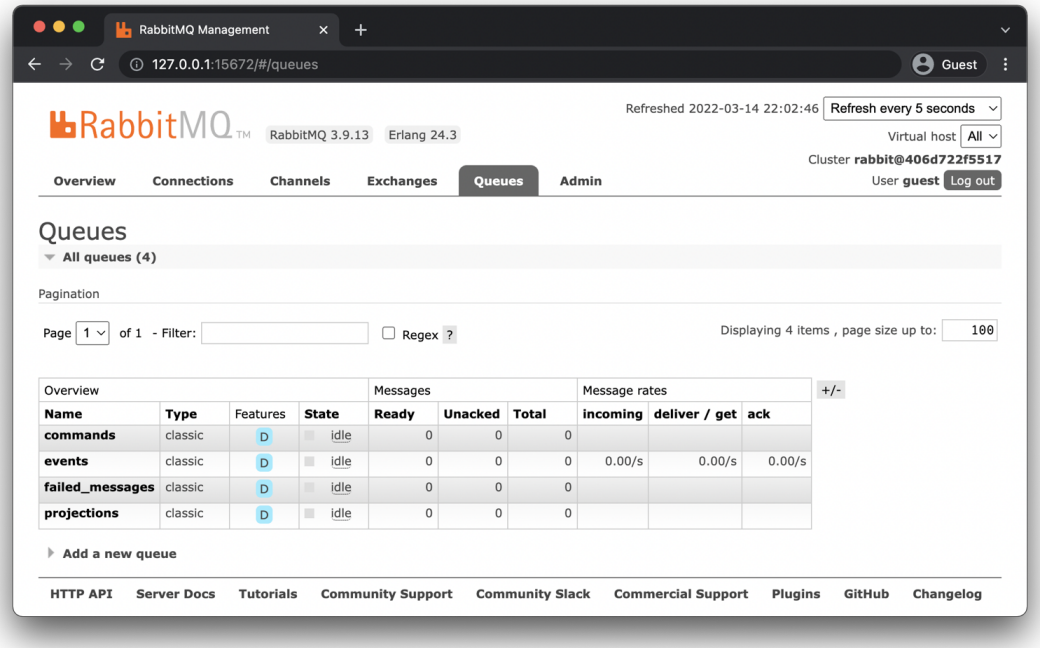
```
INFO [messenger] Received message ...\NewAuthorSigned
INFO [messenger] Message ...\NewAuthorSigned handled by ...\NewAuthorSignedEventHand\
ler
INFO [messenger] ...\NewAuthorSigned was handled successfully
INFO [messenger] Received message ...\NewAuthorSigned
INFO [messenger] Message ...\NewAuthorSigned handled by ...\NewAuthorSignedEventHand\
ler
INFO [messenger] ...\NewAuthorSigned was handled successfully
INFO [messenger] Received message ...\NewAuthorSigned
INFO [messenger] Message ...\NewAuthorSigned handled by ...\NewAuthorSignedEventHand\
ler
INFO [messenger] ...\NewAuthorSigned was handled successfully
INFO [messenger] Stopping worker. ... "events_async"
INFO [messenger] Worker stopped due to maximum count of 3 messages processed
```

Because we created three Authors in the previous step, there are three `NewAuthorSigned` Domain Events waiting to be processed. These types of Events are handled by the `NewAuthorSignedEventHandler`, which will trigger another message to generate the `CreateFollowersCounterProjection`. Projections in the system are configured to be run synchronously, so there's no need to store them in RabbitMQ or tell Symfony Messenger to consume the messages. Each Author will finally end up with their own Follower Count Projection in Redis initialized to `0`.



Projection is calculated

Now that all Domain Events are processed and their corresponding Projections are calculated, there are no messages left to be processed.



RabbitMQ has no pending messages

We can once again run the request to get the Follower Count for an Author, and we'll no longer get an empty response:

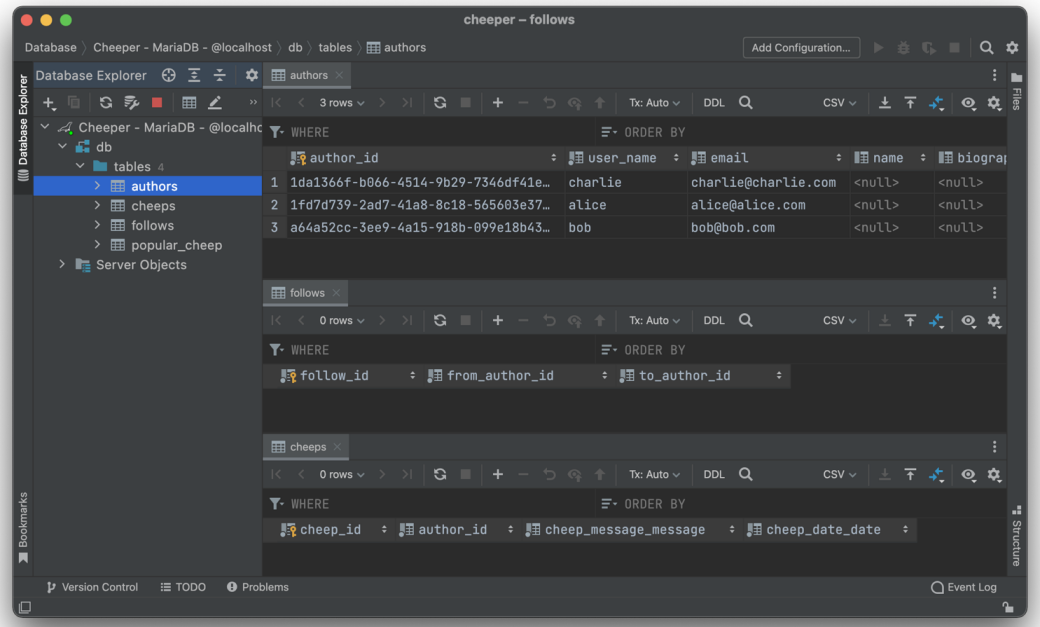
```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/followers-counter
```

```
{
    "data": {
        "authorId": "a64a52cc-3ee9-4a15-918b-099e18b43119",
        "authorUsername": "bob",
        "numberOfFollowers": 0
    },
    "meta": {
        "message_id": "ab9b470b-6d81-4c66-84bd-f52305d50edd"
    }
}
```

However, the Timeline is still empty. This is because nobody on the platform has posted a Cheep yet, which means there are no following relationships between Authors, and there are no Timelines built:

```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/timeline
```

```
{
    "data": {
        "cheeps": []
    },
    "meta": {
        "message_id": "c7aee96e-64c5-435e-82eb-4ad93e5e7639"
    }
}
```



No changes in the database

Following Other Authors

Author Timelines are built over the relationships they have with other Authors. To begin creating the Timeline, we can start there.

Following an Author is a Command that, unlike the Sign Up Author Command, is configured to be run asynchronously. This means that once the request to Cheeper's API sends the Command to be processed by an Application worker, it'll return immediately, without waiting for a response.

Although the HTTP request gives a response almost instantaneously, the database won't change until the Command is processed:

```
http --json --body POST http://127.0.0.1:8000/chapter7/follow \
    follow_id="8cc71bf2-f827-4c92-95a5-43bb1bc622ad" \
    from_author_id="1fd7d739-2ad7-41a8-8c18-565603e3733f" \
    to_author_id="a64a52cc-3ee9-4a15-918b-099e18b43119"
```

```
{
    "data": {
        "from_author_id": "1fd7d739-2ad7-41a8-8c18-565603e3733f",
        "to_author_id": "a64a52cc-3ee9-4a15-918b-099e18b43119"
    },
    "meta": {
        "message_id": "34fbcd28-9b85-4f4d-bd5f-13d78cef6652"
    }
}
```

```
http --json --body POST http://127.0.0.1:8000/chapter7/follow \
    follow_id="f3088920-841e-4577-a3c2-efdc80f0dea5" \
    from_author_id="1da1366f-b066-4514-9b29-7346df41e371" \
    to_author_id="a64a52cc-3ee9-4a15-918b-099e18b43119"
```

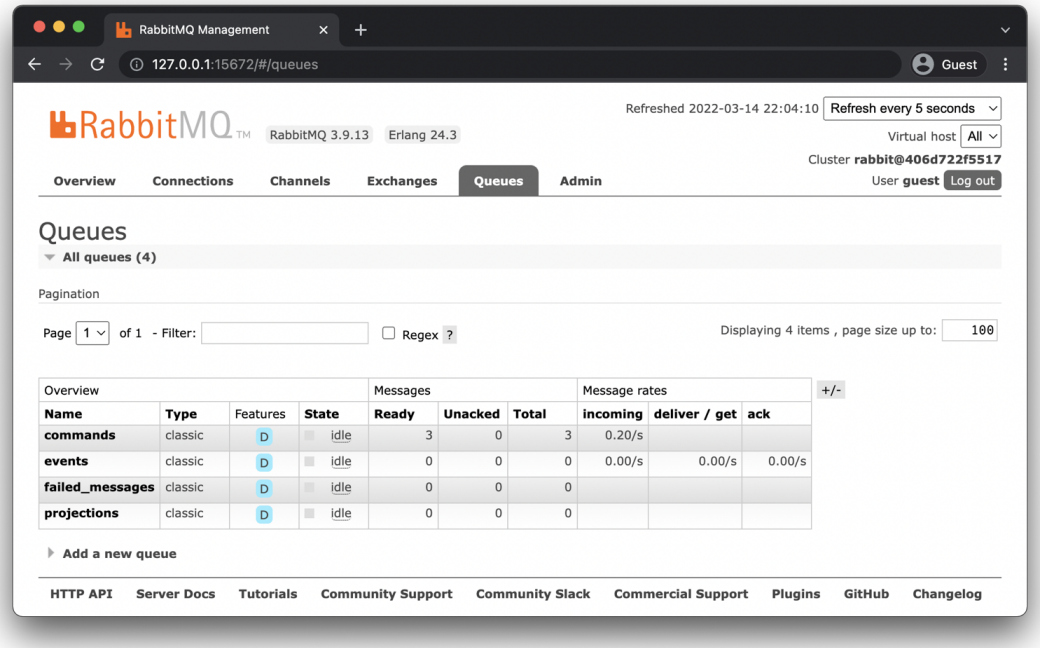
```
{
    "data": {
        "from_author_id": "1da1366f-b066-4514-9b29-7346df41e371",
        "to_author_id": "a64a52cc-3ee9-4a15-918b-099e18b43119"
    },
    "meta": {
        "message_id": "4327f5e1-6256-4856-bb0c-c33c59d97b13"
    }
}
```

Now's a good time to test the resiliency of our system. We can simulate a duplicated request and verify that our system is idempotent later on:

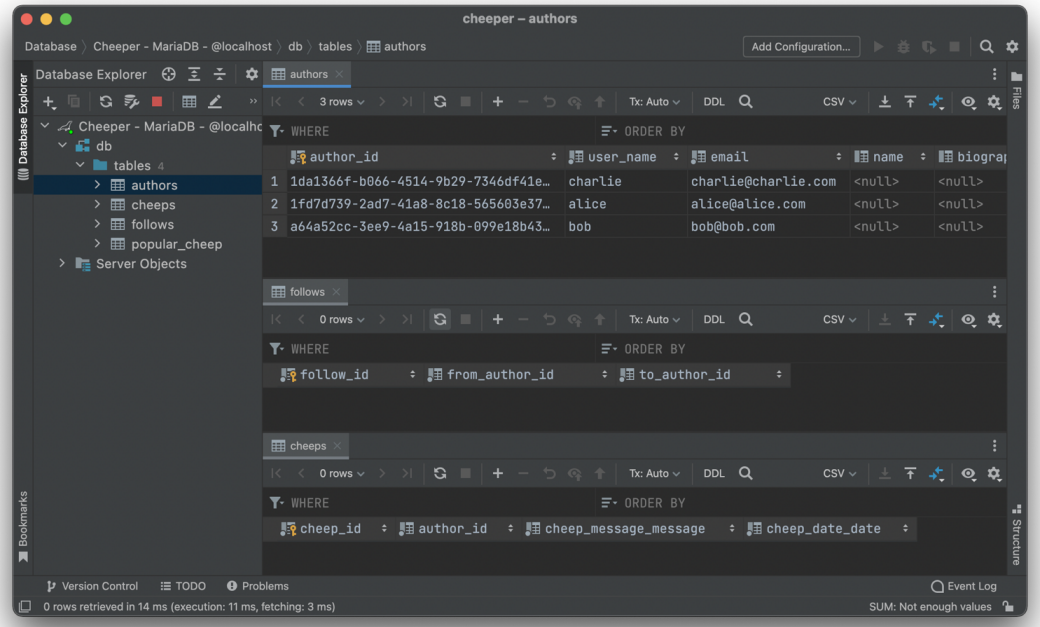
```
http --json --body POST http://127.0.0.1:8000/chapter7/follow \
    follow_id="f3088920-841e-4577-a3c2-efdc80f0dea5" \
    from_author_id="1da1366f-b066-4514-9b29-7346df41e371" \
    to_author_id="a64a52cc-3ee9-4a15-918b-099e18b43119"
```

```
{
    "data": {
        "from_author_id": "1da1366f-b066-4514-9b29-7346df41e371",
        "to_author_id": "a64a52cc-3ee9-4a15-918b-099e18b43119"
    },
    "meta": {
        "message_id": "4cd40043-fd48-4a63-a559-e1773a385d28"
    }
}
```

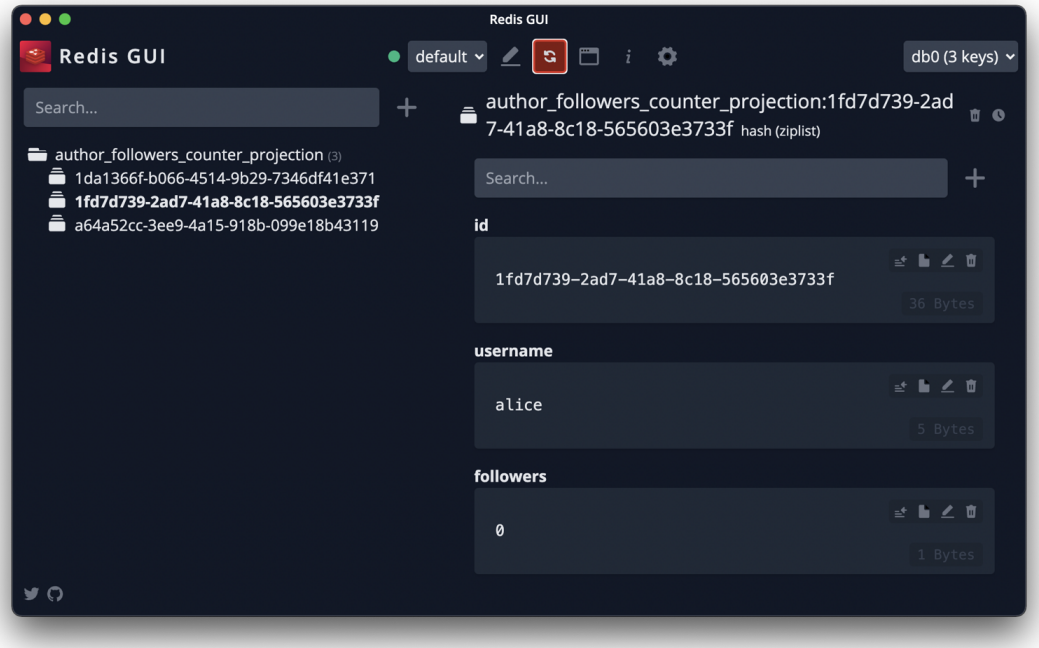
At this point, we should have three `FollowCommands` waiting to be consumed in the `commands` RabbitMQ queue. In the same way as with Domain Events, we'll need to kick off a worker to consume these messages.



RabbitMQ has three commands pending



Follows are not yet in the database



Redis has no changes

Consuming `FollowCommand` Commands

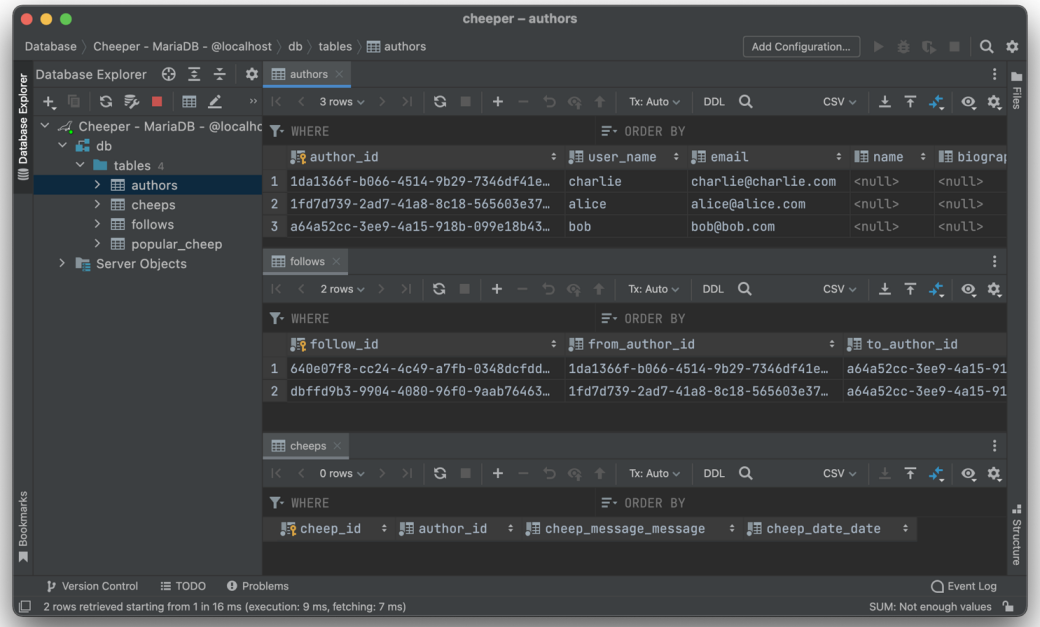
The command line to process `FollowCommand` is similar to the one we used to consume Domain Events. However, this time we have to specify that Symfony Messenger should consume messages from the `commands_async` channel:

```
docker compose exec app php bin/console messenger:consume commands_async --limit 3 -\
vv
```

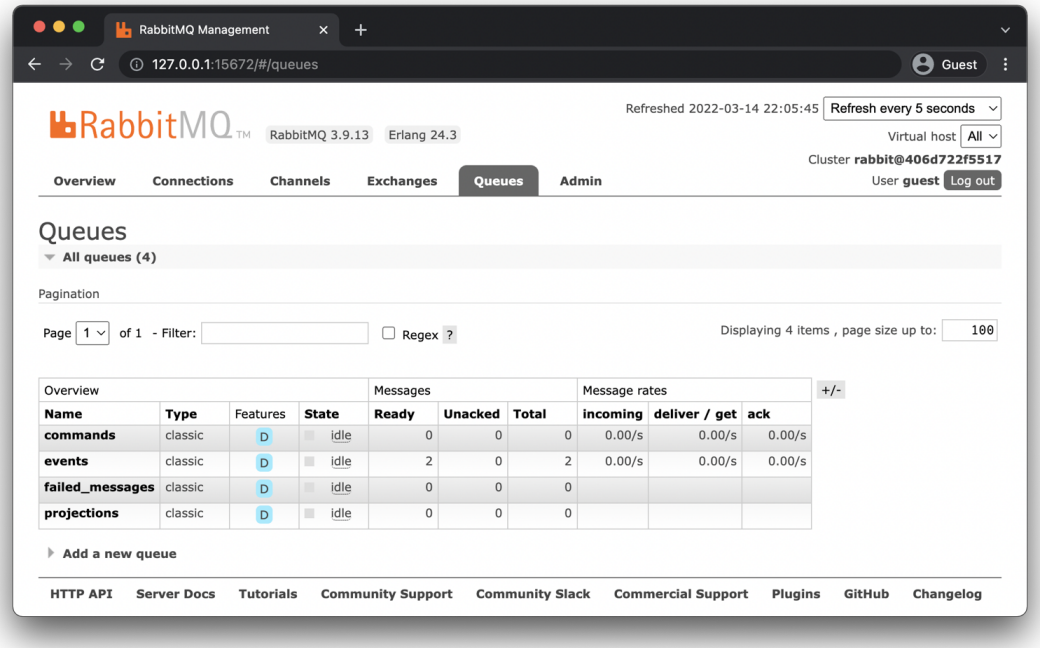
Here's the resulting output:

```
INFO [messenger] Received message ...\FollowCommand ...
INFO [messenger] Sending message ...\AuthorFollowed with events_async sender ...
INFO [messenger] Message ...\FollowCommand handled by ...\FollowCommandHandler ...
INFO [messenger] ...\FollowCommand was handled successfully
INFO [messenger] Received message ...\FollowCommand
INFO [messenger] Sending message ...\AuthorFollowed with events_async sender ...
INFO [messenger] Message ...\FollowCommand handled by ...\FollowCommandHandler ...
INFO [messenger] ...\FollowCommand was handled successfully
INFO [messenger] Received message ...\FollowCommand
INFO [messenger] Message ...\FollowCommand handled by FollowCommandHandler ...
INFO [messenger] ...\FollowCommand was handled successfully
INFO [messenger] Stopping worker. ... "commands_async"
INFO [messenger] Worker stopped due to maximum count of 3 messages processed
```

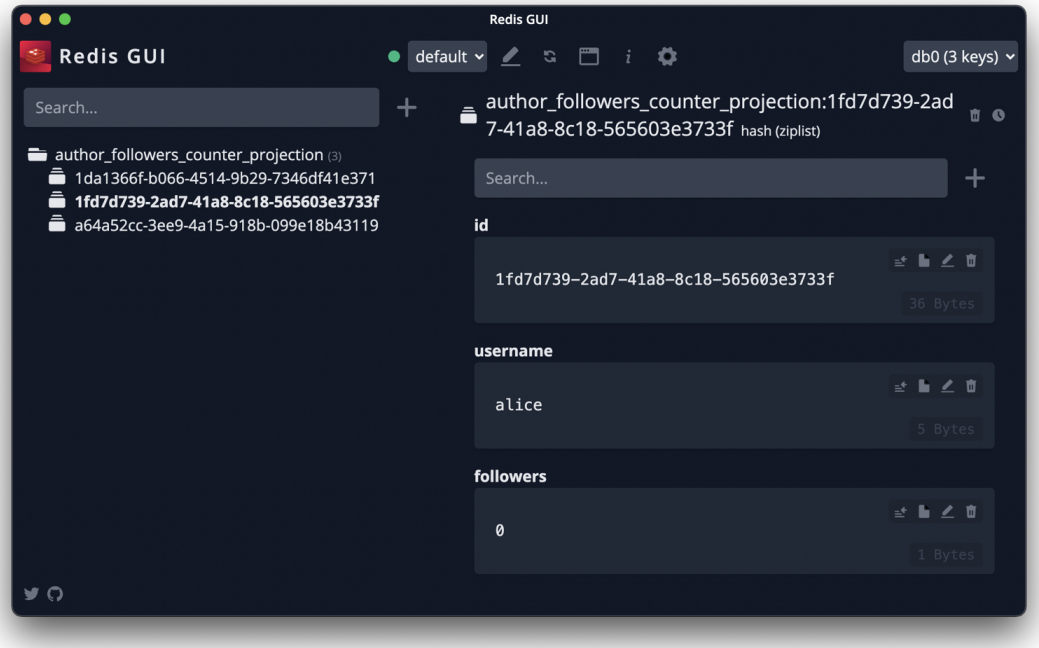
Two new following relationships will be created, and two new `AuthorFollowed` Domain Events will be triggered for processing.



Follows are now in the database



RabbitMQ has two pending Events



Redis has no changes

Verifying an Author's Followers

Although the Author `bob` has two Followers, the follow count for him hasn't been updated:

```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/followers-counter
```

```
{
    "data": {
        "authorId": "a64a52cc-3ee9-4a15-918b-099e18b43119",
        "authorUsername": "bob",
        "numberOfFollowers": 0
    },
    "meta": {
        "message_id": "0e9df870-894f-4d37-a140-4d598941fa4f"
    }
}
```

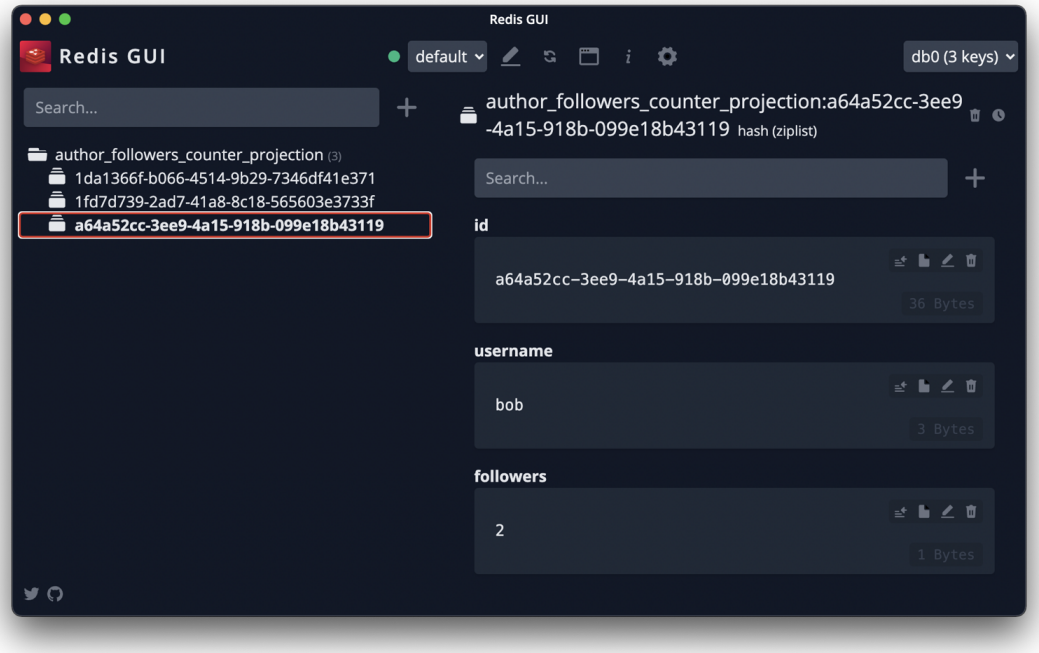
The number of Followers is still `0` because the `AuthorFollowed` Domain Events are waiting to be consumed once again.

Consuming `AuthorFollowed` Events

As we did before, we need to tell Symfony Messenger to spin off a worker to consume the `AuthorFollowed` Events waiting in the `events_async` channel transport:

```
docker compose exec app php bin/console messenger:consume events_async --limit 2 -vv

INFO [messenger] Received message ...\AuthorFollowed
INFO [messenger] Message ...\AuthorFollowed handled by ...\AuthorFollowedEventHandle\
r ...
INFO [messenger] ...\AuthorFollowed was handled successfully
INFO [messenger] Received message ...\AuthorFollowed
INFO [messenger] Message ...\AuthorFollowed handled by ...\AuthorFollowedEventHandle\
r ...
INFO [messenger] ...\AuthorFollowed was handled successfully
INFO [messenger] Stopping worker. ... "events_async"
INFO [messenger] Worker stopped due to maximum count of 2 messages processed
```

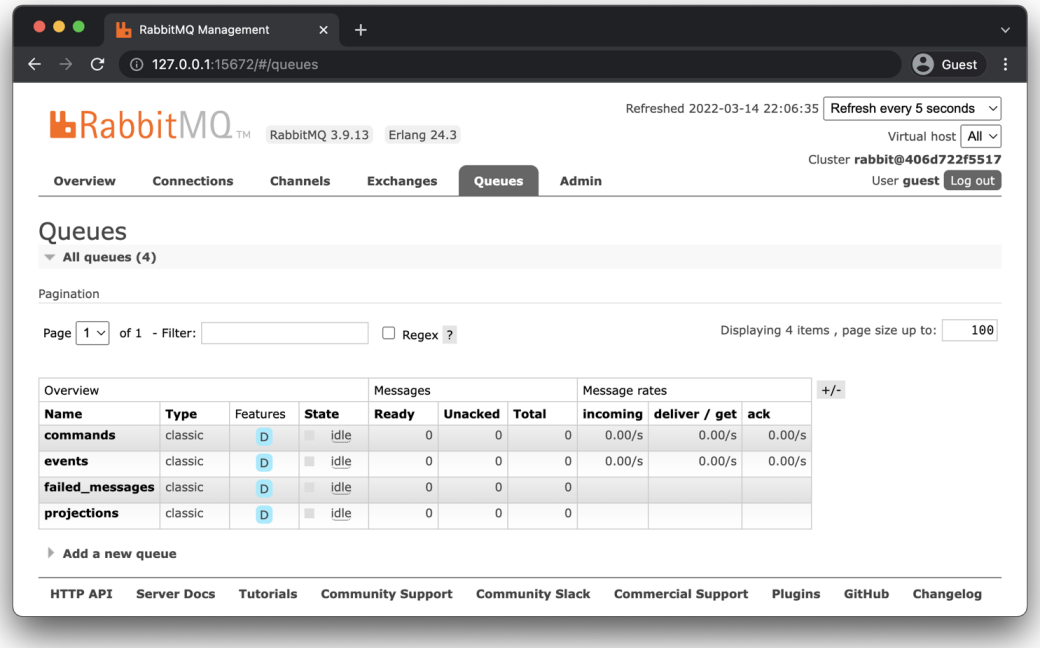
Now all the Events have been handled and the Follower Count Projection for `bob` is updated. Remember that Projections are configured to run synchronously, so there's no need to tell Symfony Messenger to consume the Projection messages triggered by the Event Handler.



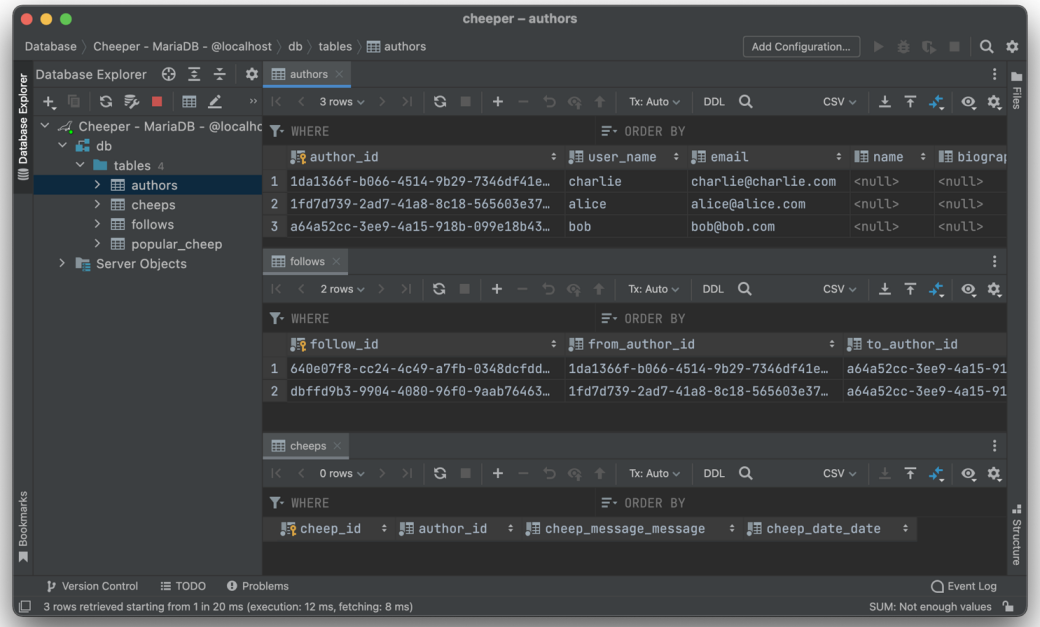
Redis Projection has been updated

```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/followers-counter
```

```
{
    "data": {
        "authorId": "a64a52cc-3ee9-4a15-918b-099e18b43119",
        "authorUsername": "bob",
        "numberOfFollowers": 2
    },
    "meta": {
        "message_id": "00bb7fe5-f2ba-4cc2-bbca-9fa4d26fbf69"
    }
}
```



RabbitMQ has 0 pending messages



Database has no changes

Posting Cheeps

It's time to show how a Cheep posted by `bob` ends up updating his Follower Timelines. In the same way as with the Follow Author use case, Post Cheep is an asynchronous Command, which means that the HTTP response will be instantaneous and that the Command will be waiting inside RabbitMQ to be processed by a worker:

```
http --json --body POST http://127.0.0.1:8000/chapter7/cheep \
    cheep_id="28bc90bd-2dfb-4b71-962f-81f02b0b3149" \
    author_id="a64a52cc-3ee9-4a15-918b-099e18b43119" \
    message="Hello world, this is Bob"
```

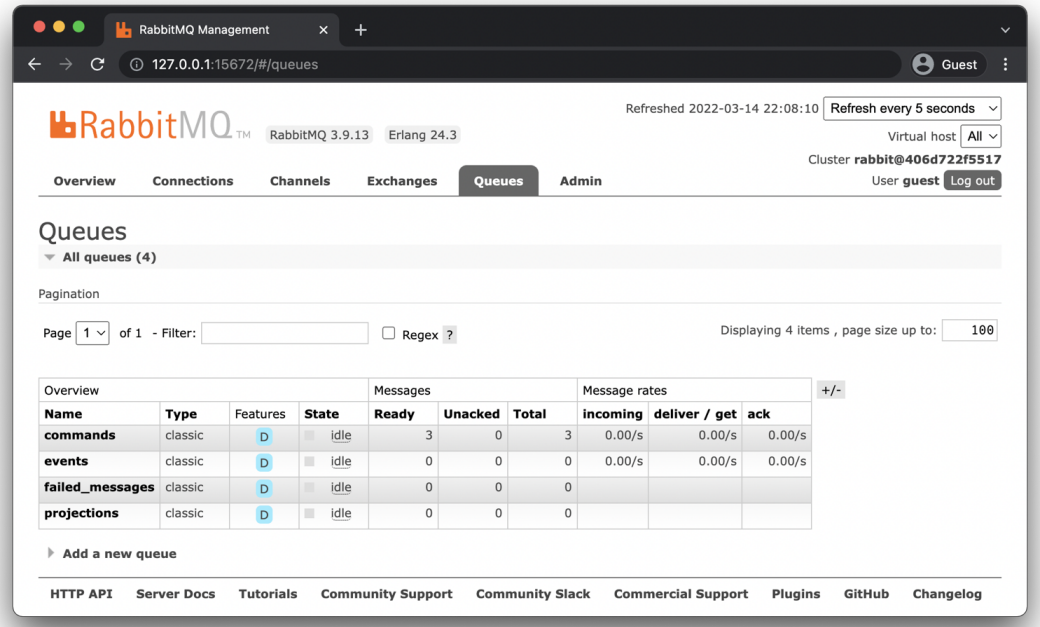
```
{
    "data": {
        "cheep_id": "28bc90bd-2dfb-4b71-962f-81f02b0b3149"
    },
    "meta": {
        "message_id": "9ef17e3c-6532-4b06-8fc4-73c78c4eccdd"
    }
}
```

```
http --json --body POST http://127.0.0.1:8000/chapter7/cheep \
    cheep_id="04efc3af-59a3-4695-803f-d37166c3af56" \
    author_id="1fd7d739-2ad7-41a8-8c18-565603e3733f" \
    message="Hello world, this is Alice"
```

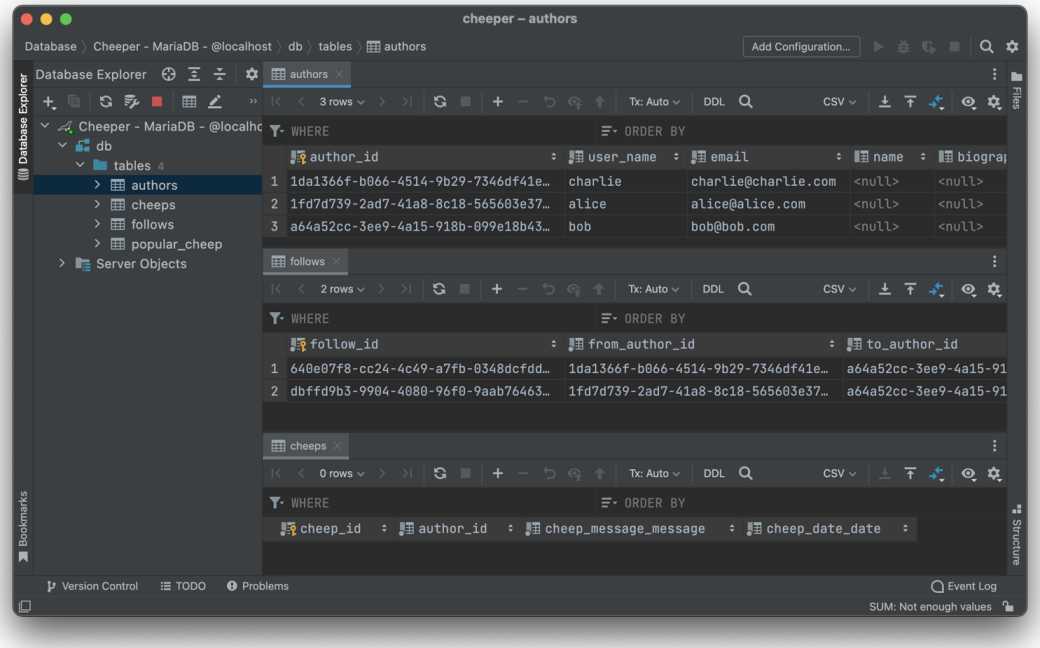
```
{
    "data": {
        "cheep_id": "04efc3af-59a3-4695-803f-d37166c3af56"
    },
    "meta": {
        "message_id": "fb8fd10f-c096-4e30-a46b-16a2d93b8f89"
    }
}
```

```
http --json --body POST http://127.0.0.1:8000/chapter7/cheep \
    cheep_id="8a5539e6-3be2-4fa7-906e-179efcfca46b" \
    author_id="1da1366f-b066-4514-9b29-7346df41e371" \
    message="Hello world, this is Charlie"
```

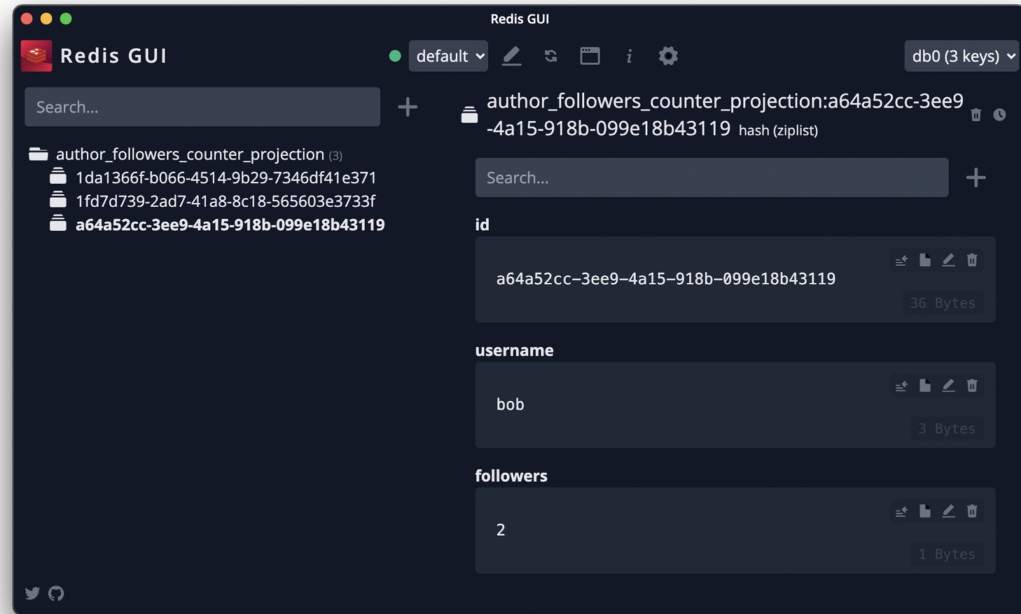
```
{
    "data": {
        "cheep_id": "8a5539e6-3be2-4fa7-906e-179efcfca46b"
    },
    "meta": {
        "message_id": "6fba149b-1f70-489d-a4ea-1aa3e23b2b3a"
    }
}
```



RabbitMQ has three Commands waiting



Database has no changes



Redis has no changes

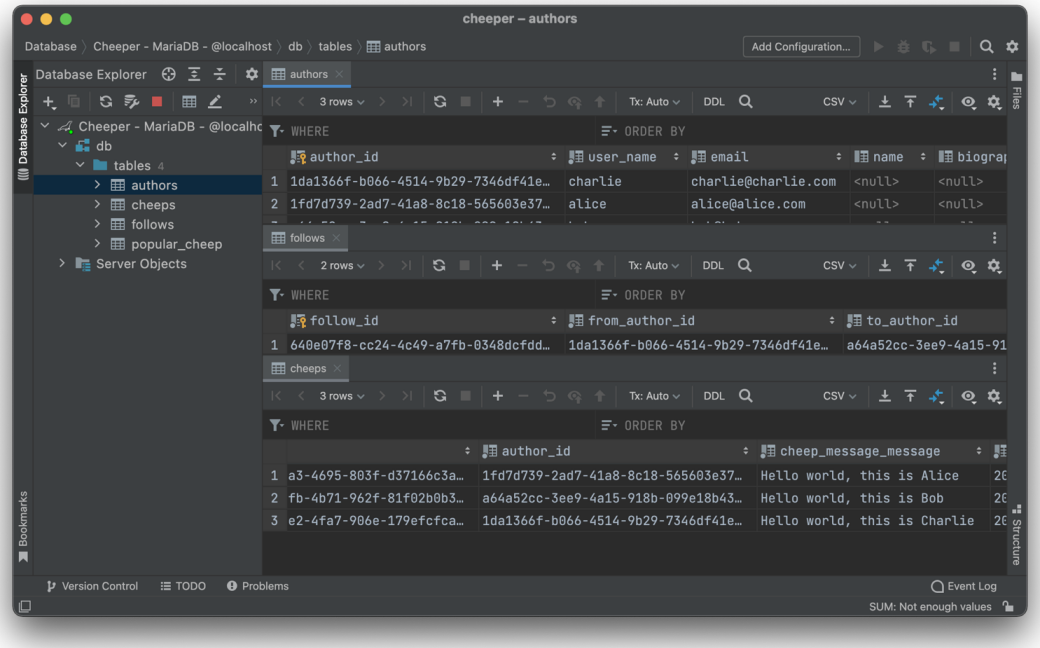
Consuming `PostCheepCommand` Commands

As with other messages configured to be processed asynchronously, we'll need to instruct Symfony Messenger to consume the three `PostCheepCommand` messages waiting to be processed in the `commands_async` channel:

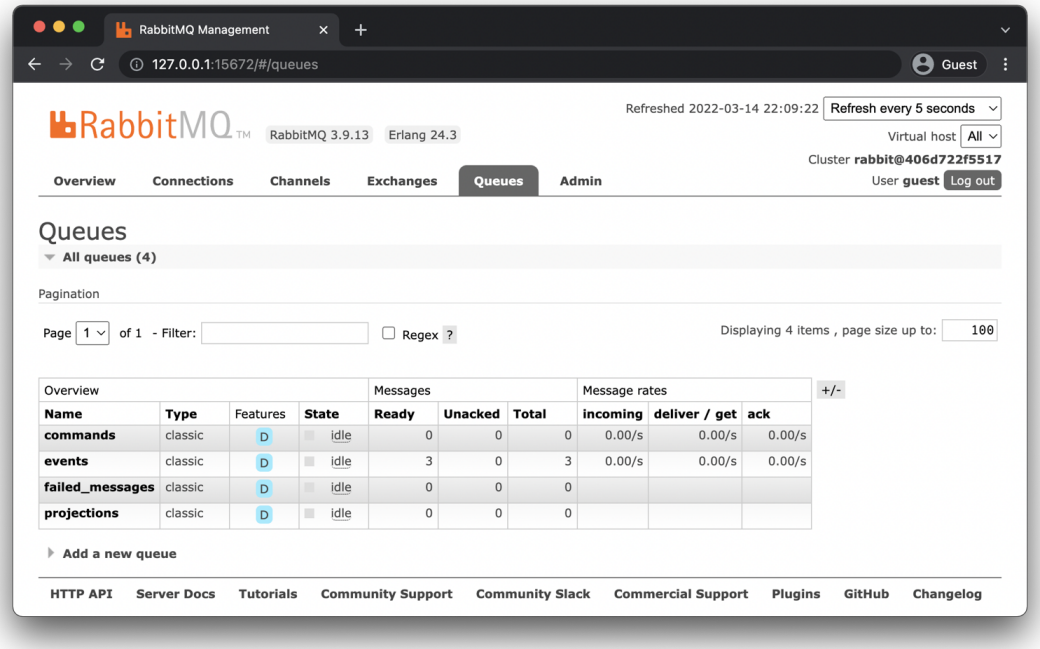
```
docker compose exec app php bin/console messenger:consume commands_async --limit 3 -\
vv

INFO [messenger] Received message ...\PostCheepCommand
INFO [messenger] Sending message ...\CheepPosted with events_async sender ...
INFO [messenger] Message ...\PostCheepCommand handled by ...\PostCheepCommandHandler
INFO [messenger] ...\PostCheepCommand was handled successfully
INFO [messenger] Received message ...\PostCheepCommand
INFO [messenger] Sending message ...\CheepPosted with events_async sender ...
INFO [messenger] Message ...\PostCheepCommand handled by ...\PostCheepCommandHandler
INFO [messenger] ...\PostCheepCommand was handled successfully
INFO [messenger] Received message ...\PostCheepCommand
INFO [messenger] Sending message ...\CheepPosted with events_async sender ...
INFO [messenger] Message ...\PostCheepCommand handled by ...\PostCheepCommandHandler
INFO [messenger] ...\PostCheepCommand was handled successfully
```

```
16  INFO [messenger] Stopping worker. ... "commands_async"
17  INFO [messenger] Worker stopped due to maximum count of 3 messages processed
```

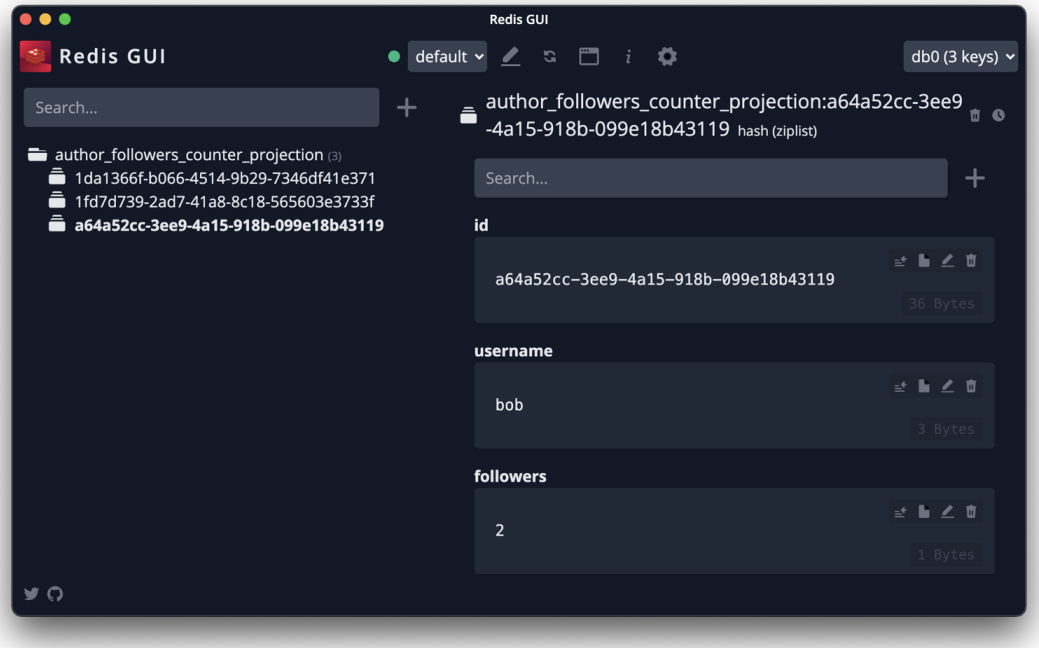
Now we should have the three Cheeps stored in the MySQL Data Model database.



Database has the Cheep messages



RabbitMQ has now three Events waiting

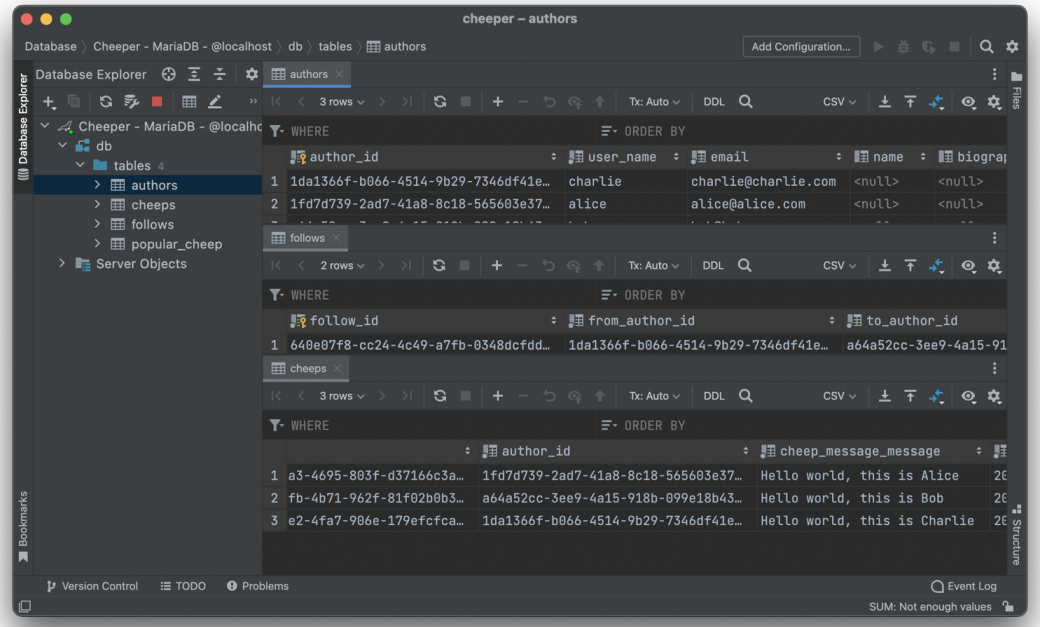


Redis has no changes

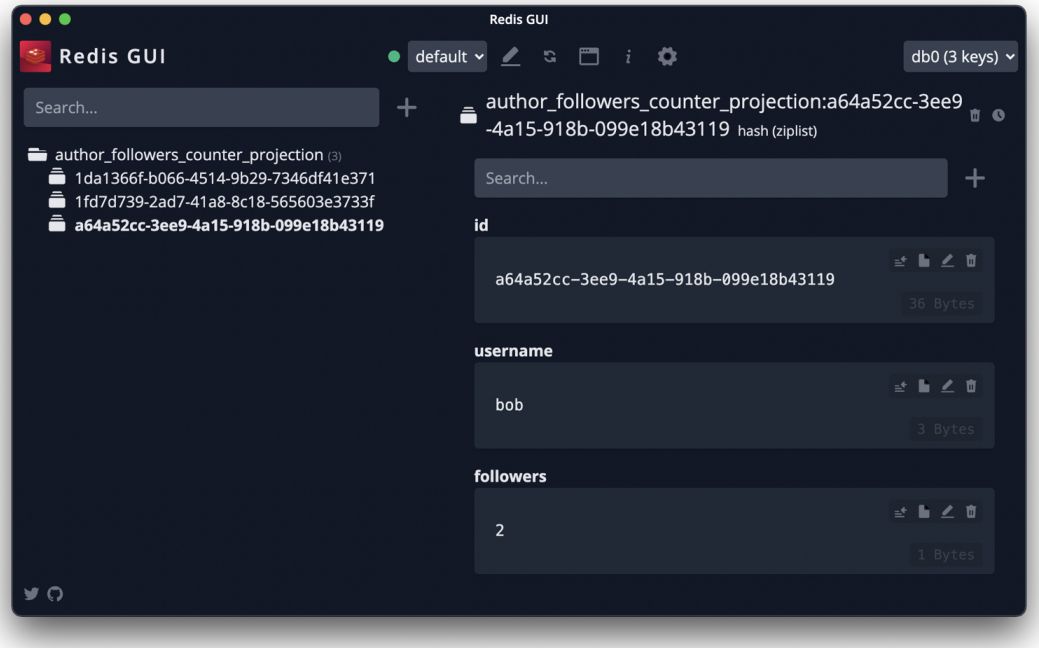
Consuming `CheepPosted` Events

Once the Command Handler finishes, there should be three `CheepPosted` Domain Events waiting to be processed in the `events_async` channel. To consume them, do the following:

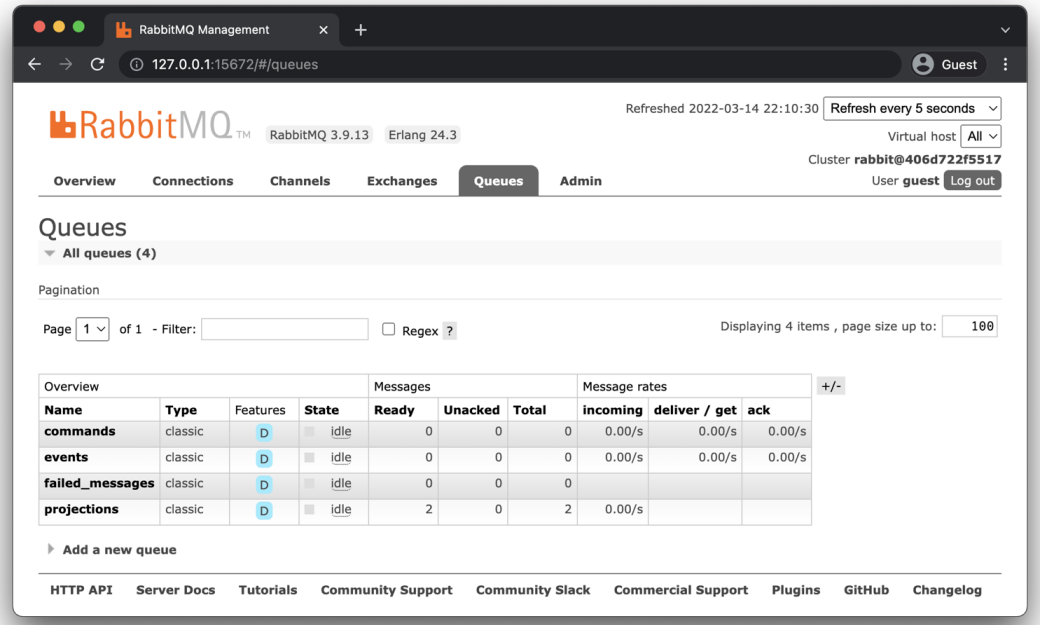
```
docker compose exec app php bin/console messenger:consume events_async --limit 3 -vv

INFO [messenger] Received message ...\CheepPosted
INFO [messenger] Sending message ...\AddCheepToTimelineProjection with projections_a\
sync sender
INFO [messenger] Sending message ...\AddCheepToTimelineProjection with projections_a\
sync sender
INFO [messenger] Message ...\CheepPosted handled by ...\CheepPostedEventHandler
INFO [messenger] ...\CheepPosted was handled successfully
INFO [messenger] Received message ...\CheepPosted
INFO [messenger] Message ...\CheepPosted handled by ...\CheepPostedEventHandler
INFO [messenger] ...\CheepPosted was handled successfully
INFO [messenger] Received message ...\CheepPosted
INFO [messenger] Message ...\CheepPosted handled by ...\CheepPostedEventHandler
INFO [messenger] ...\CheepPosted was handled successfully
INFO [messenger] Stopping worker. ... "events_async"
INFO [messenger] Worker stopped due to maximum count of 3 messages processed
```



Database has no changes



Redis has no changes

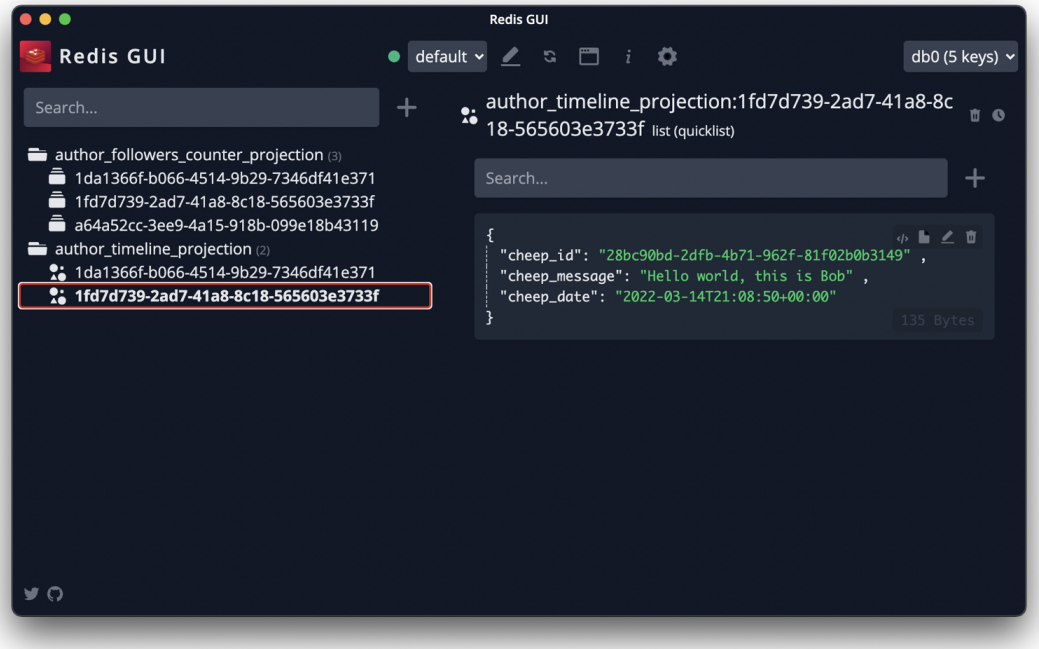


RabbitMQ has now two Projections waiting

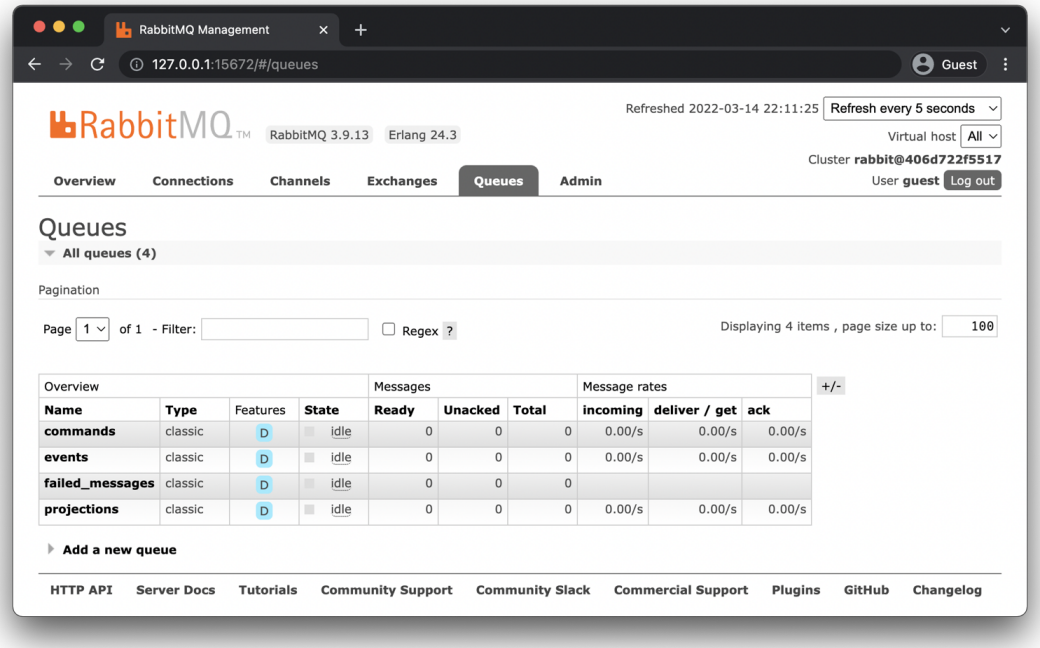
Consuming `AddCheepToTimelineProjection` Projections

The `CheepPostedEventHandler` triggers two `AddCheepToTimelineProjections` — one for each Follower of `bob` — but this time, the Projections will be run asynchronously due to the heaviness of building Timelines. These Projection messages will be stored temporarily in RabbitMQ while waiting to be processed. To start consuming messages from the `projection_async` channel:

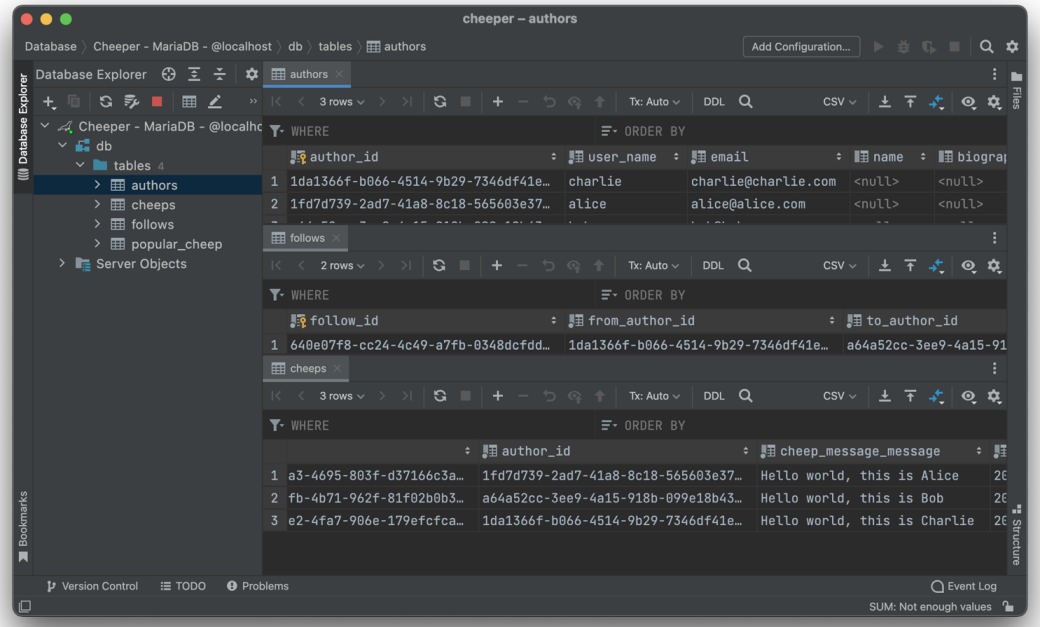
```
docker compose exec app php bin/console messenger:consume projections_async --limit \
2 -vv

INFO [messenger] Received message ...\AddCheepToTimelineProjection
INFO [messenger] Message ...\AddCheepToTimelineProjection handled by ...\AddCheepToT\
imelineProjectionHandler
INFO [messenger] ...\AddCheepToTimelineProjection was handled successfully
INFO [messenger] Received message ...\AddCheepToTimelineProjection
INFO [messenger] Message ...\AddCheepToTimelineProjection handled by ...\AddCheepToT\
imelineProjectionHandler
INFO [messenger] ...\AddCheepToTimelineProjection was handled successfully
INFO [messenger] Stopping worker. ... "projections_async"
INFO [messenger] Worker stopped due to maximum count of 2 messages processed
```



Redis has the two timelines



RabbitMQ has no pending messages



Database has no changes

Verifying an Author's Timeline

Now we can finally verify that Author Timelines are being built correctly. Each Follower of `bob` should have the Cheep we just posted before as part of their Timeline. By design, we decided not to include Cheeps in one's own Author Timeline, so `bob` won't have Cheeps in his Timeline:

```
http --json --body http://127.0.0.1:8000/chapter7/author/a64a52cc-3ee9-4a15-918b-099\
e18b43119/timeline
```

```
{
    "data": {
        "cheeps": []
    },
    "meta": {
        "message_id": "c48c8334-b624-4d5f-a71a-e8d99b3e30ec"
    }
}
```

However, both `alice` and `charlie` — who follow `bob` — should have our previous Cheep included in their Timelines:

```
http --json --body http://127.0.0.1:8000/chapter7/author/1fd7d739-2ad7-41a8-8c18-565\
603e3733f/timeline
```


```
{
    "data": {
        "cheeps": [
            {
                "cheep_date": "2022-03-14T21:08:50+00:00",
                "cheep_id": "28bc90bd-2dfb-4b71-962f-81f02b0b3149",
                "cheep_message": "Hello world, this is Bob"
            }
        ]
    },
    "meta": {
        "message_id": "fdbf9ad6-edc4-4013-a980-f95e7a37850f"
    }
}
```


```
http --json --body http://127.0.0.1:8000/chapter7/author/1da1366f-b066-4514-9b29-734\
6df41e371/timeline
```


```
{
    "data": {
        "cheeps": [
            {
                "cheep_date": "2022-03-14T21:08:50+00:00",
                "cheep_id": "28bc90bd-2dfb-4b71-962f-81f02b0b3149",
                "cheep_message": "Hello world, this is Bob"
            }
        ]
    },
    "meta": {
        "message_id": "79be4121-5432-44fb-a6f0-fe4ced613224"
    }
}
```


Wrapup

In this chapter, we demonstrated how to run and debug the use cases and flows we developed and evolved through the book. We started from scratch with no schemas or data in our databases, and

we ran use cases such as Sign Up Author, Follow Author, Post Cheep, and Fetch Author Timeline. We also looked in detail at how the flow of messages and Buses worked.

A full CQRS flow that persists the Domain Model into the Write Side and prepares the Read Model Projections to be optimally queried has the following incremental steps:

1. Controller
2. Command
3. Command Bus
4. Command Handler
5. Domain Event
6. Event Handler
7. Projection
8. Projection Handler

To fetch the information, the flow of the CQRS components would have the following steps:

1. Controller
2. Query
3. Query Bus

Additionally, we saw how the Infrastructure layer evolved after each step and how the workers responsible for processing asynchronous messages glued everything together. We now not only know how to run Cheeper, but we also understand how every piece in CQRS fits into the architecture and how it works.