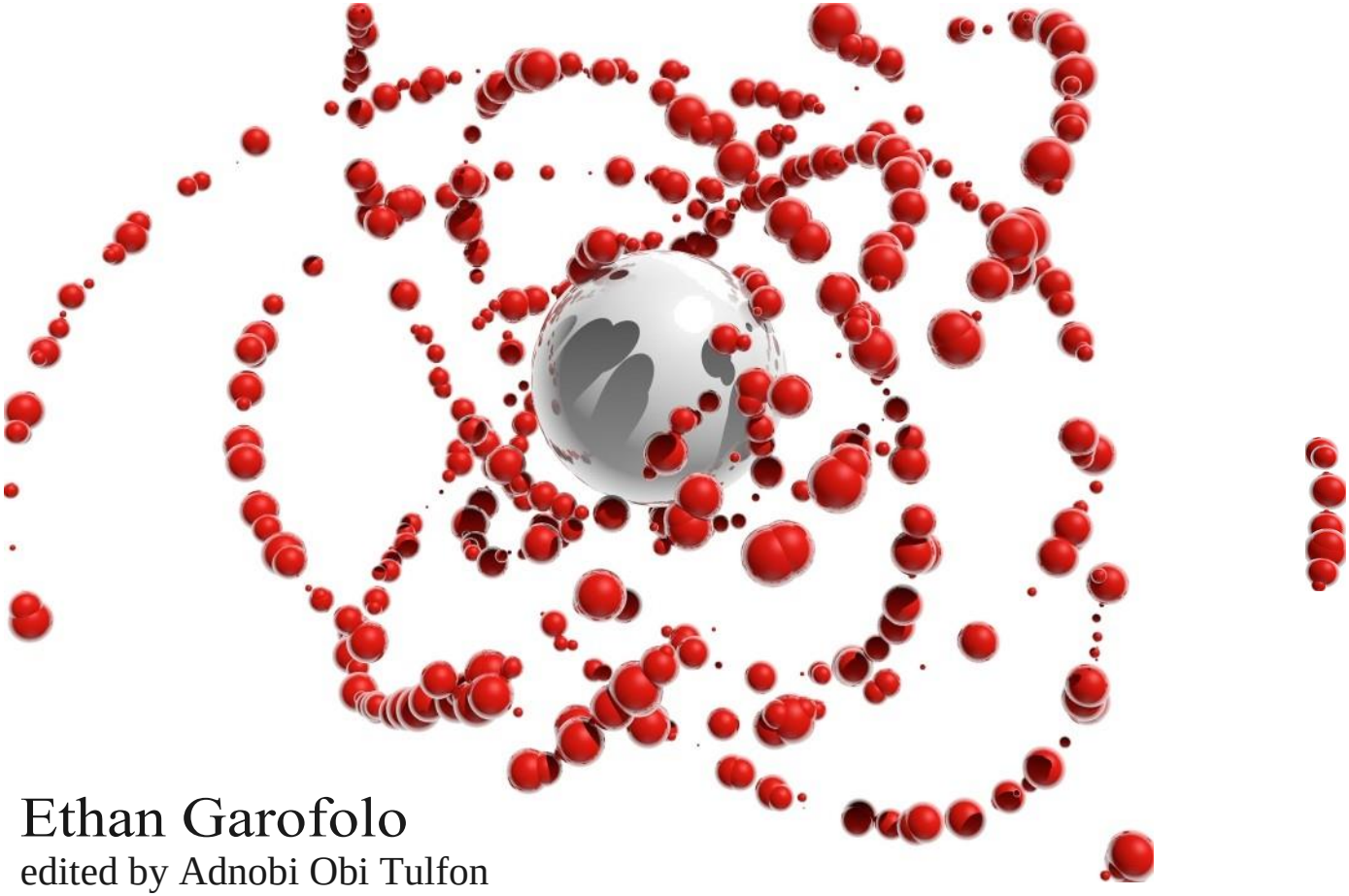
The Pragmatic Programmers

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS



Ethan Garofolo
edited by Adnobi Obi Tulfon

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS

Ethan Garofolo

The Pragmatic Bookshelf

Raleigh, North Carolina

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow Executive
Editor: Dave Rankin Development Editor:
Adaobi Obi Tulton Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*. For

international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-645-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2020

We are in bondage to the law in order that we may be free.

> *Marcus TulliusCicero*

CHAPTER 2

Writing Messages

When last we left our intrepid hero, the maw of the monolith was closing around. As if in a bad version of *Groundhog Day*, teeth bearing the stench of a thousand years were about to lay waste to an exciting greenfield project, and another iteration of the horrific loop was about to begin.

But in that darkest moment, when hope itself had failed, yet one light still shone. Our hero summoned the wisdom of software design principles, unmasked the monster, and restored, let's say, justice to the land.

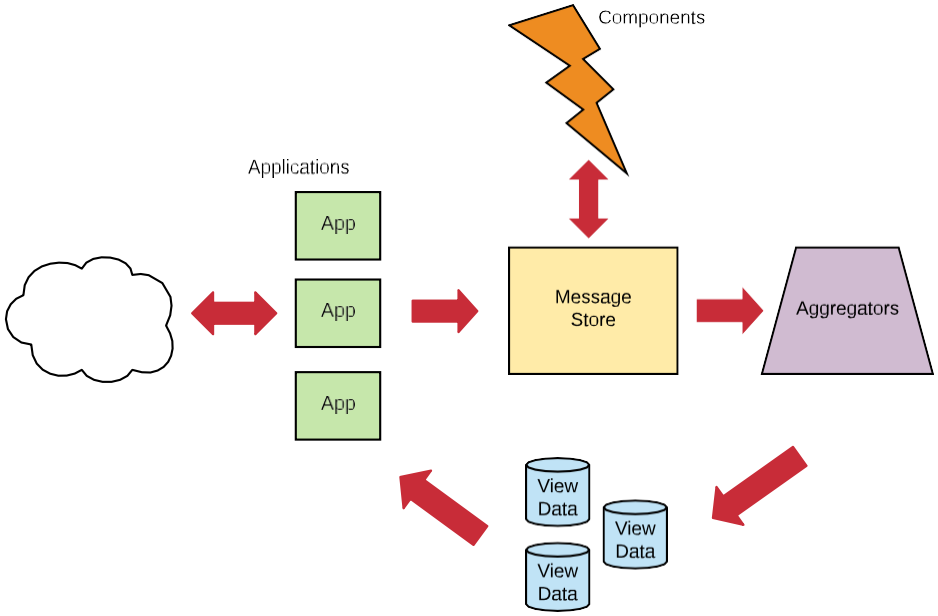
That's pretty much the tale people will tell of your efforts in building Video Tutorials. Tongue less in cheek, in this chapter you are going to unmask the monolith. It turns out "monolith" doesn't just mean "code that's hard to work on." A monolith does tend to be difficult, and you're going to uncover why that is so that you can prevent that great darkness from claiming yet another project. Eternal glory awaits.

Unmasking the Monolith

Software is easy to write, but hard to change. MVC CRUD frameworks lead you to monolithic architectures, which optimize for writing. Those quick results they deliver come at the cost of adding a great deal of coupling into your project.[1] Coupling is the enemy of change. Coupling is what makes it impossible to make a change in subsystem A without fear of breaking subsystem B, C, and D. If you rename that column in your users table, well, hope- fully you've cleared your calendar for the weekend.

That's fine if you're prototyping *applications*, but it's a sandy foundation for long-lived systems. Recall our system map (see the figure on page 26) from the previous chapter.

1. https://en.wikipedia.org/wiki/Coupling_(computer_programming)



Applications are just one part of an overall system. There arealso Components, Aggregators, and View Data to build and maintain. If you don't remember what these different pieces are and what they do, refer back to our list on page 22.

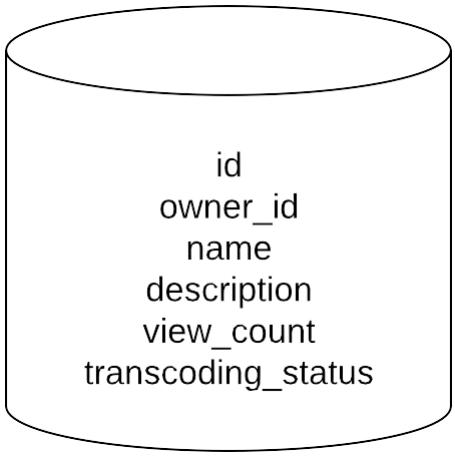
But what is the heart of a monolith's coupling? What is it that makes some- thing a monolith? Is it a question of language? Framework? Database? Codebase size? Monorepo vs. polyrepo? How many servers the system is running on?

Depending on where you land when searching in the blogosphere, those might be posed as the essential questions. Or you might read that "microservices" means "stuff running on different machines." Or Docker. Surely, if you use Docker, you're dealing with microservices. Right? Right?

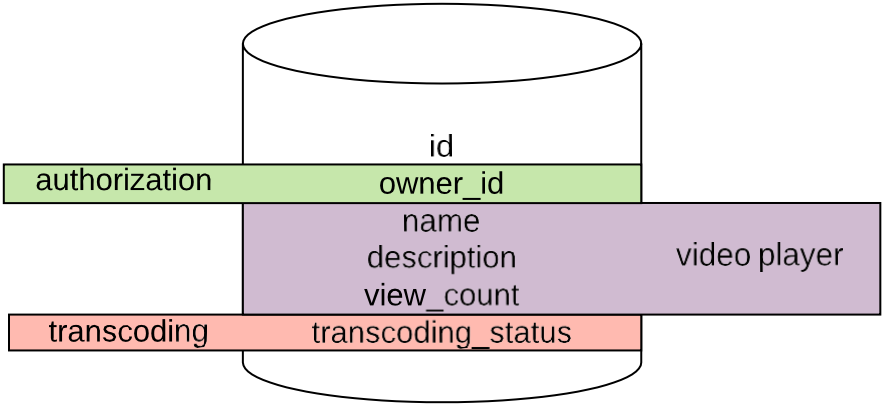
You landed here though, and you're going to learn all throughout this book that while those questions are important, none of them has any bearing as to whether or not something is a monolith. You can build a service-based system with Rails, and you can certainly build monoliths with Docker. *A monolith is a data model and not a deployment or code organization strategy.*

Trying to Compress Water

Remember the videos table from the previous chapter?



It has a mere six columns: id, owner_id, name, description, transcoding_status, and view_count. Do these pieces of data really all belong together? Do they represent a single thing? Can you imagine asingle operation that simultaneously requires all six pieces of data? How many different concerns are represented here?



owner_id is useful if we're trying to make sure that someone trying to do something to this video is allowed to—authorization. But how does that help us when putting the video in the video player that other users will see? It would be nice to see the owner's actual name there, but this table doesn't

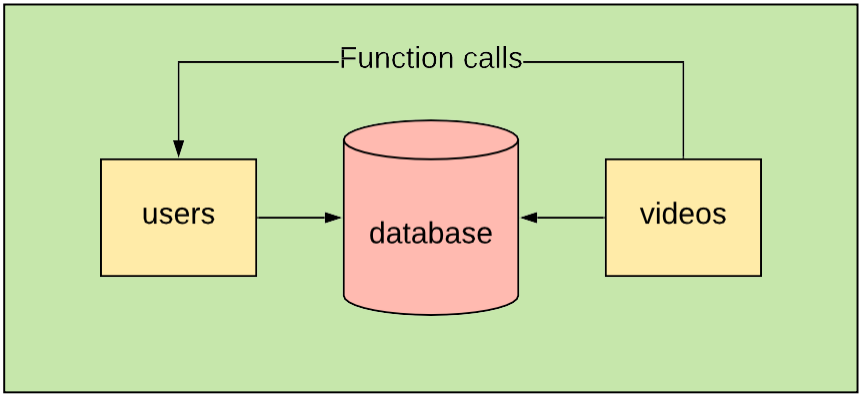
even have that information. Finally, transcoding_status has nothing to do with either of those. Transcoding a video happens in the background, and if a video isn't ready for viewing, why would it show up at all for users who aren't the owner?

The thing represented by this table is an *aggregation* of many different con- cerns. Which means that any changes to this table could potentially break each of those concerns. This will happen every time you rely on a *canonical data model*[2]—the one representation of a system entity to rule them all and in the darkness bind them. You can try to force these data together, in the same way that you can try to compress some measure of water. It can be done, but only at the cost of considerable energy. That's because they don't want to be together.

This data model *is* the monolith, and no change of language, framework, or deployment strategy can fix that. Nor does the blogosphere's general advice in these situations.

Extracting "Microservices"

When you hit your productivity wall, the blogosphere is going to tell you to Just Extract Microservices.™ Okay. Here's our fledgling system with a hypo- thetical users concern added to the mix:

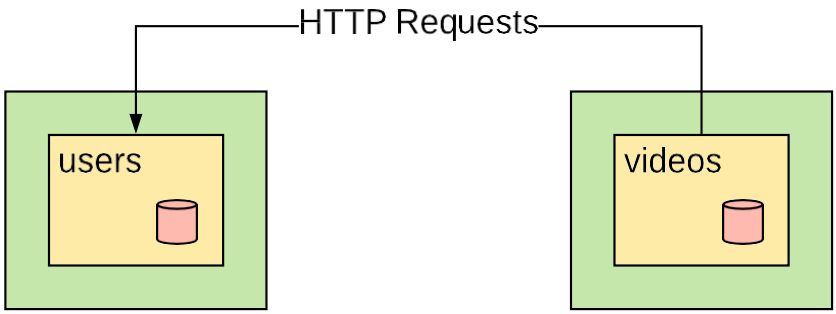


All of these pieces being in the green box represents that their running on the same server. users and videos are run-of-the-mill MVC "models" that have a table of the same name inside of the big, honkin' database sitting in the middle of it all. videos, of course, makes function calls to this users "model" to

2. https://www.innoq.com/en/blog/thoughts-on-a-canonical-data-model/

get data about users. Imagine this project scaling to all sorts of "models," and then we hit the all-too-common "we can't work on this anymore, so we print our resumes and go find another job."

Or desperate to make it better we try to implement the blogosphere's advice and just extract the users and videos "microservices":



More green boxes, more problems. We've paid a price here. Two servers. Two databases. Those function calls from before? They're HTTP calls now. It's okay to pay a price to receive value, but what value did we receive here?

Absolutely nothing.

The data model *has not changed*. And what's worse, if this so-called "users" service goes down, it's bringing the so-called "videos" service down with it because "videos" *depends on* "users." "Videos" now has two reasons to fail. BOO!

What you have in that image is not microservices.

It is in every aspect still a monolith, only it is distributed now—a *distributed monolith*. Every aspect of working on this system will be more difficult than it was before. Your operational costs just went up (!). What you used to be able to do with a JOIN in a database you have to re-implement in app code (!!). You now need expensive tooling and orchestration to make this work even in development (!!!).

Simply putting a database table behind an HTTP interface does not produce a service, micro or otherwise. So what does?

Defining Services

The defining characteristic of microservices is *autonomy*. Nice word, but what does it mean in this context?

First of all, let's get back to calling them Components. Good. Now, Components don't respond to questions. We have things we ask questions of in software. They're called "databases," and the process of asking them something is called "querying." Even if you stick a database behind HTTP, it's still a database, architecturally speaking.

Components don't ask questions of anything else. If they did, they would *depend* on that thing and would no longer be autonomous. If you have to connect to something else to get data to make a decision, then you are not autonomous.

You might be saying, "That's all well and good, but that sounds like the properties of both a black hole and a white hole—no information in, no information out. Not only a paradox, but entirely useless in software." There is a way, and that way is through *asynchronous messages*.

Getting Components to Do Things

Asynchronous messages are what make service-based architectures possible. We'll just call messages from here on, but they come in two flavors: *commands* and *events*.

Commands are *requests to do something*. Maybe that's transferring some funds, or maybe that's sending an email. Whatever the desired operation is, commands are merely *requests*. The Component that handles a given com- mand chooses whether or not to carry it out.

Components produce *events* in response to commands. Events are records of *things that have happened*. That means whatever they represent, it's already done. You don't have to like it, but you do have to deal with it.

Commands and events, collectively messages, are moved around in the system via *publish-subscribe*, or pub/sub for short.[3] In pub/sub, publishers publish data, and they're not sure who subscribes to it. Subscribers similarly don't know who published the information they consume.

When you write a Component, as you will in Chapter 9, Adding an Email Component, on page 133, you document which commands your Component handles as well as which events your Component publishes. With those messages defined, interested parties can request your Component do some- thing and can also respond to the events your Component publishes.

3. https://en.wikipedia.org/wiki/Publish–subscribe_pattern

Representing Messages in Code

So what does a message look like? We can represent them as JSON objects:

```
{
  "id": "875b04d0-081b-453e-925c-a25d25213a18",
  "type": "PublishVideo",
  "metadata": {
    "traceId": "ddecf8e8-de5d-4989-9cf3-549c303ac939", "userId":
    "bb6a04b0-cb74-4981-b73d-24b844ca334f"
  },
  "data": {
    "ownerId": "bb6a04b0-cb74-4981-b73d-24b844ca334f", "sourceUri":
    "https://sourceurl.com/",
    "videoId": "9bfb5f98-36f4-44a2-8251-ab06e0d6d919"
  }
}
```

This is a command you'll define in Chapter 10, Performing Background Jobs with Microservices, on page 157.

At the root of this object we have four fields:

id
: Every message gets a unique ID, and we use UUIDs for them.

type
: A string and something you choose when you define your messages. When we said earlier that events represent things that have happened, it's the type that tells us what that thing that happened was. And in the case of commands, the type tells us we want to have happen.

metadata
: An object that contains, well, metadata. The contents of this object have to do with the mechanics of making our messaging infrastructure work. Examples of fields we'll commonly find in here include traceId, which ties messages resulting from the same user input together. Every incoming user request will get a unique traceId, and any messages written as part of that user request will get written with that request's traceId. If there are any components that write other messages in response to those messages, then those messages will have the same traceId. In that way, we can easily track everything that happened in the system in response to a particular user request. We'll put this into action in Chapter 13, Debugging Compo- nents, on page 207, which deals with debugging strategies. We will also

commonly have a userId string, representing the ID of the user who caused the message to be written.

data
: A JSON object itself, and the "payload" of the event. The contents of a message's data field are analogous to the parameters in a function call.

You can tell that this event is a command because the type is in the imperative mood. This is a convention we will always follow. Since a command is a *request* to do something, its type is in the imperative mood. This is in contrast to the event that might get generated in response to this command:

```
{
  "id": "23d2076f-41bd-4cdb-875e-2b0812a27524",
  "type": "VideoPublished", "metadata": {
    "traceId": "ddecf8e8-de5d-4989-9cf3-549c303ac939", "userId":
    "bb6a04b0-cb74-4981-b73d-24b844ca334f"
  },
  "data": {
    "ownerId": "bb6a04b0-cb74-4981-b73d-24b844ca334f", "sourceUri":
    "https://sourceurl.com/",
    "videoId": "9bfb5f98-36f4-44a2-8251-ab06e0d6d919"
  }
}
```

Notice the type on this event is in the past tense. That's because events are things that have already happened.

Naming Messages

Giving types to our messages is the most important of the engineering work we'll do. As we add features to our system, we'll use messages to represent the various processes the system carries out. The preceding messages come from the video cataloging process we'll build in Chapter 10, Performing Background Jobs with Microservices, on page 157.

We come up with types for the messages in collaboration with our company's business team. Message types are named after the business processes they represent. Furthermore, we select types using language familiar to experts in the domain we are modeling. This is not something we can do alone as developers.

If we were modeling banking, we might have messages like TransferFunds, AccountOpened, and FundsDeposited. We absolutely will not have types that contain "create," "update," or "delete." We're purging that CRUD from our vocabulary.

Storing State as Events

Up to this point, this concept of commands and events may be familiar. You may have already used technology such as Apache Kafka[4] to have components communicate via events. We're going to take it further though.

You may receive a command like PetPuppies, a command that should neverbe rejected. When a command is processed, the output is one or more events, such as PuppiesPet. If we wanted to know whether or not the puppies have been pet, how could we tell? Take a moment and think about it…

All we'd have to look for is that event. Instead of treating the messages as transient notifications, discarding them when we're done, we *save* them. Then we can constitute and reconstitute current state or state at any point in time to our heart's content. This is *event sourcing*—sourcing state from events.

If you've done MVC CRUD apps, then you're probably used to receiving incoming requests and then updating one or more rows in a database. At any given point, you were storing the *current state* of your system, having discarded all knowledge of how the system got into that state. Because we're already going to use messages to communicate between portions of our system, why not keep them around? Then we could use the events to know what the state of our system is now, and at any point in the past.

Storing Messages in Streams

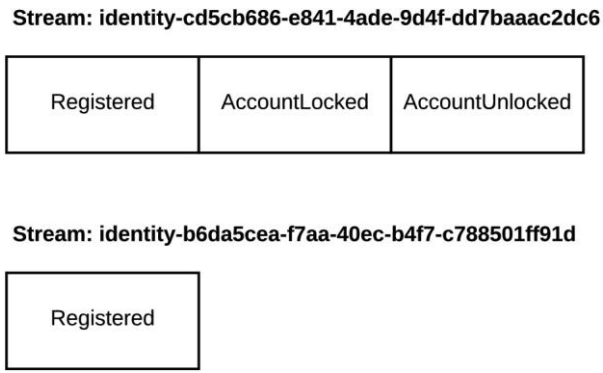
One last note about messages before we begin writing them. When we start writing messages on page 36, we'll organize them into what we call *streams*. Streams group messages together logically, usually representing an entity or process in your system. Within a stream, messages are stored in the order they were written.

For example, Video Tutorials users will have an identity. We'll explicitly model that identity in Chapter 6, Registering Users, on page 83. All the events related to a particular user's identity will be in the same stream, and those are the only events that will be in that stream. Using the naming conventions of the Eventide Project,[5] a Ruby toolkit for building autonomous microservices, we call this type of stream an *entity stream*. We use UUIDs[6] as identifiers in our system, specifically version 4 UUIDs, and so a natural name for one of these user identity streams would be identity-81cb4647-1296-4f3b-8039-0eedae41c97e.

4. https://kafka.apache.org/
5. http://docs.eventide-project.org/core-concepts/streams/stream-names.html
6. https://en.wikipedia.org/wiki/Universally_unique_identifier

While any part of the system is free to *read* the events in such a stream, a property we'll use in Chapter 8, Authenticating Users, on page 119, an entity stream only has a single writer.

Here is a pair of such identity streams:



There are other kinds of streams, though. If all goes as planned, Video Tuto- rials will have more than one user, each with a stream of the form identity-UUID. Every event in such an entity stream is also part of the identity *category stream*. To get the category stream that an entity stream belongs to, just take every- thing to the left of the first dash. So for identity-81cb4647-1296-4f3b-8039-0eedae41c97e, identity is the category. The identity category stream contains every *event* written to every identity in our system.

We talked about commands on page 30, and commands are also written to streams. They aren't written to entity streams, though—they are written to *command streams*. In the case of this identity Component, we'll write to streams of the form identity:command-81cb4647-1296-4f3b-8039-0eedae41c97e. This is an *entity command stream*, and it only contains commands related to a particular entity. What is the category of this stream? Is it the same as the entity stream from before? Again, to get a category from a stream, take everything to the left of the first dash. For this entity command stream, that gives us identity:com- mand, which is not the same as identity. So no, entity streams are not in the same category as entity command streams.

Streams, like messages, don't get deleted. Messages are added to them in an append-only manner.

Now, if there's anything we can take from the 1984 *Ghostbusters* film, crossing the streams is Bad™ and continued abstract talk about streams will likely lead to that. Now that we have the basics of messages, let's get to a concrete example and resolve the cliffhanger we started on page 20.

Defining Component Boundaries

Stream boundaries are Component boundaries. If we have a stream category such as identity, what we're saying is that there's going to be a single Compo- nent authorized to *write* to streams in the identity category. Those are Compo- nent boundaries. Similarly, if we have command streams in the category identity:command, only that same Component is authorized to handle those commands. These strict rules are part of avoiding monoliths. In a monolith, anyone can add columns to the users table and make updates to its rows. Not so in our architecture! The lack of these boundaries are why we can't just Extract Microservices™ from a monolith. As you'll see when we look at dis- tributing our system in Chapter 12, Deploying Components, on page 195, it's precisely these boundaries that allow us to extract things.

Sometimes, although not in this book, a Component will own more than one entity. This is rare, and it also doesn't break our rule in the previous para- graph. A category has a single owner, even on the rare occasions that a par- ticular owner happens to own another category.

Recording VideoViews

How are we going to record that a video was viewed? Since we're not going to just use an MVC CRUD-style database table, it stands to reason that we're going to write a message. Should that be an event or a command?

Questions are best answered by going back to fundamental principles. In our initial talks with the business team, one of the longer-term (read: outside the scope of this book) visions is to have users buy memberships to see premium content, and the creators of that content would get paid based on how many times their videos were viewed. So we know an event eventually needs to be written, and since we're recording that a video was viewed, VideoViewed seems like a good type for this kind of event.

The next question then, who writes that event? Should the record-viewings application write events directly, or should it write commands that some Component picks up and handles? We previously discussed on page 33 how a given event stream is populated by one and only one writer. So far, it could go either way.

Let's say that we have the application write the events and that it writes to streams of the form viewingX, where X is a video's ID. Are we capturing the necessary state? Check. Is there only a single writer to a given stream? Check. So far so good.

What if we wanted to run potential video views through some sort of algorithm to detect fake views before committing them to system state? Obviously none of *our* users would ever do that, but, you hear stories. Investors like to know we've addressed this kind of thing, and content creators would want to know the view counts are legit.

That seems like a bit much to put into a request/response cycle and something that we would want to put into a Component. That Component would currently do nothing besides receiving, say RecordVideoViewing commands, and writing VideoViewed events. We don't need to support this right now, so why take on that extra burden?

The only reason to do so would be if this choice affects long-term maintain- ability. Does it? If we had the application write the viewed events and later decided to move this into a Component, what would we need to change?

1. Refactor the Application to write a command to a command stream rather than an event to an event stream.
2. Write the Component.

We'd have to do step 2 anyway, and step 1 doesn't sound that hard. If we were doing this CRUD style, we might have had to set up a different API to call into with the video view. We'd have the same issue where verifying the view takes too long to put into a request/response cycle, so that API would likely have done some sort of background job. When it's done with the verifi- cation, maybe it would make another call into our application to record the result of that verification? Or we modify our application to pull from a different database? Or we directly couple our application to that API via a shared database table? Those all sound like messes from a design perspective, let alone the operational concerns of having to make two network hops. With a pub/sub flow that stores state as events and makes sure a given event stream only has a single writer, we're able to proceed confidently in the short term without setting ourselves up with a costly refactoring later.

Let's not worry about the Component right now and just have the Application record video viewings. We'll get to our first Component soon enough in Chapter 6, Registering Users, on page 83.

Writing Your First Message

Okay, when we left the record-viewings application, we were in the function that would record that a video was viewed. Here's where we left off:

first-pass/src/app/record-viewings/index.js

```
function createActions ({ db
}) {
  function recordViewing (traceId, videoId) {
  }

  return {
    recordViewing
  }
}
```

We decided on page 35 that we'll use VideoViewed events to record video views. We'll write these events to streams of the form viewing-X, where X is the video's ID. So all we need to do in this function is build the event we're going to write and then write it to the Message Store:

The Code Directory Has Changed

From this point going forward in the book, the code is in the code/video-tutorials/ folder.

If you're using Docker for the database, be sure to stop the container for the first-pass folder and switch to code/video-tutorials and re-run docker-compose rm -sf && docker-compose up.

video-tutorials/src/app/record-viewings/index.js

```
function createActions ({
  messageStore
}) {
  function recordViewing (traceId, videoId, userId) {
    const viewedEvent = {
      id: uuid(),
      type: 'VideoViewed',
      metadata: {
        traceId,
        userId
      },
      data: { userId,
        videoId
      }
    }
    const streamName = `viewing-${videoId}`
    return messageStore.write(streamName, viewedEvent)
  }
  return {
    recordViewing
  }
}
```

1. First of all, notice that we've taken out the reference to the db and replaced it with a messageStore reference. We're writing an event and not updating a row in a videos table.

2. Then we construct the event. It's just a plain old JavaScript object that can be serialized to JSON. You'll notice the fields we mentioned on page 31. There isn't that much to an event.

1. Next, we construct the name of the stream that we're going to write this event to.

4. Finally, we actually call messageStore.write to write the event. That function takes the streamName that we want to write to and the message we want to write.

There's one last change we need to make in this file. In the top-level function we also need to change the db reference to messageStore:

video-tutorials/src/app/record-viewings/index.js

```
function createRecordViewings ({
  messageStore
}) {
  const actions = createActions({
    messageStore
  })
  // ... rest of the body omitted
}
```

The top-level function receives messageStore and passes it along to actions.

(Re)configuring the Record-Viewings Application

And we also have a change to our configuration to make. We need to pull in the Message Store, instantiate it, and pass it to the record-viewings application:

video-tutorials/src/config.js

```
// ...
const createPostgresClient = require('./postgres-client') const
createMessageStore = require('./message-store') function
createConfig ({ env }) {
  const knexClient =createKnexClient({
    connectionString: env.databaseUrl
  })
  const postgresClient = createPostgresClient({ connectionString:
    env.messageStoreConnectionString
  })
  const messageStore = createMessageStore({ db: postgresClient })

  const homeApp = createHomeApp({ db: knexClient })
  const recordViewingsApp = createRecordViewingsApp({ messageStore })
```

```
15    return {
-       // ...
-       messageStore,
-     }
- }
```

Line 2 requires the code that will create our database connection *to the Message Store*, and line 3 requires our Message Store code. We set up the database connection at line 8 by giving it the connection info we get from the environ- ment. We'll add that to env.js and .env is just a moment. Line 11 then instanti- ates the Message Store by passing it the postgresClient reference. Line 14 passes messageStore to instantiate the recordViewingsApp, and then we add messageStore to the config function's return value at line 17.

A quick change in src/env.js:

video-tutorials/src/env.js

```
module.exports = {
  // ...
  messageStoreConnectionString:
    requireFromEnv('MESSAGE_STORE_CONNECTION_STRING')
}
```

And be sure to add the corresponding value to .env:

```
MESSAGE_STORE_CONNECTION_STRING=
  postgres://postgres@localhost:5433/message_store
```

Make sure to put that on a single line. So, good to go, right?

Hanging a Lantern

In showbiz, when writers call attention to glaring inconsistencies, that's called "hanging a lantern on it." We have one, ah, slight problem. We don't actually have the Message Store code yet. Let's punt that to the next chapter because this one is already pretty long, and we've covered a lot in it.

What You've Done So Far

You unmasked the monolith. You learned that monoliths are data models and that they speed up the near term at the expense of the long term by introducing high levels of coupling. You also learned that the most commonly recommended methods for dealing with this coupling don't actually do any- thing to address the coupling.

Then you got your feet wet with the basics of message-based architecture. Messages are what make autonomous microservices possible, and autonomouse

microservices are what make long-term productivity possible. This was no small amount of learning.

Taking those principles then, you came back to our problem at hand—how do we record video views without writing ourselves into an inescapable corner? You learned about streams and concluded the chapter by writing a `VideoViewed` event to a stream in the `viewings` category. You stepped into a whole new world.

Aside from building the mechanics of the Message Store, which we'll start in the next chapter, being able to analyze business processes and model them as messages is the most important skill you can acquire. That's where the real engineering is done. To that end, choose some workflow that exists in the project you're currently working on. Instead of thinking of it in CRUD terms, can you model it as a sequence of events? Strike create, update, and delete from your vocabulary, and state what your current project does in terms that non-developers would use. Does it get you thinking differently about the project? Does it reveal holes in your understanding?

Lanterns are great and all, but we need something to store our messages. As we continue the journey to record video views and display the total view count on the home page, our next step is to go from mythical Message Store to actual Message Store. You may find it mundane, or you may find it exciting. Regardless of which, you'll find it in the next chapter.