

Extracted from:

Practical Microservices

Build Event-Driven Architectures
with Event Sourcing and CQRS

This PDF file contains pages extracted from *Practical Microservices*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2020 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

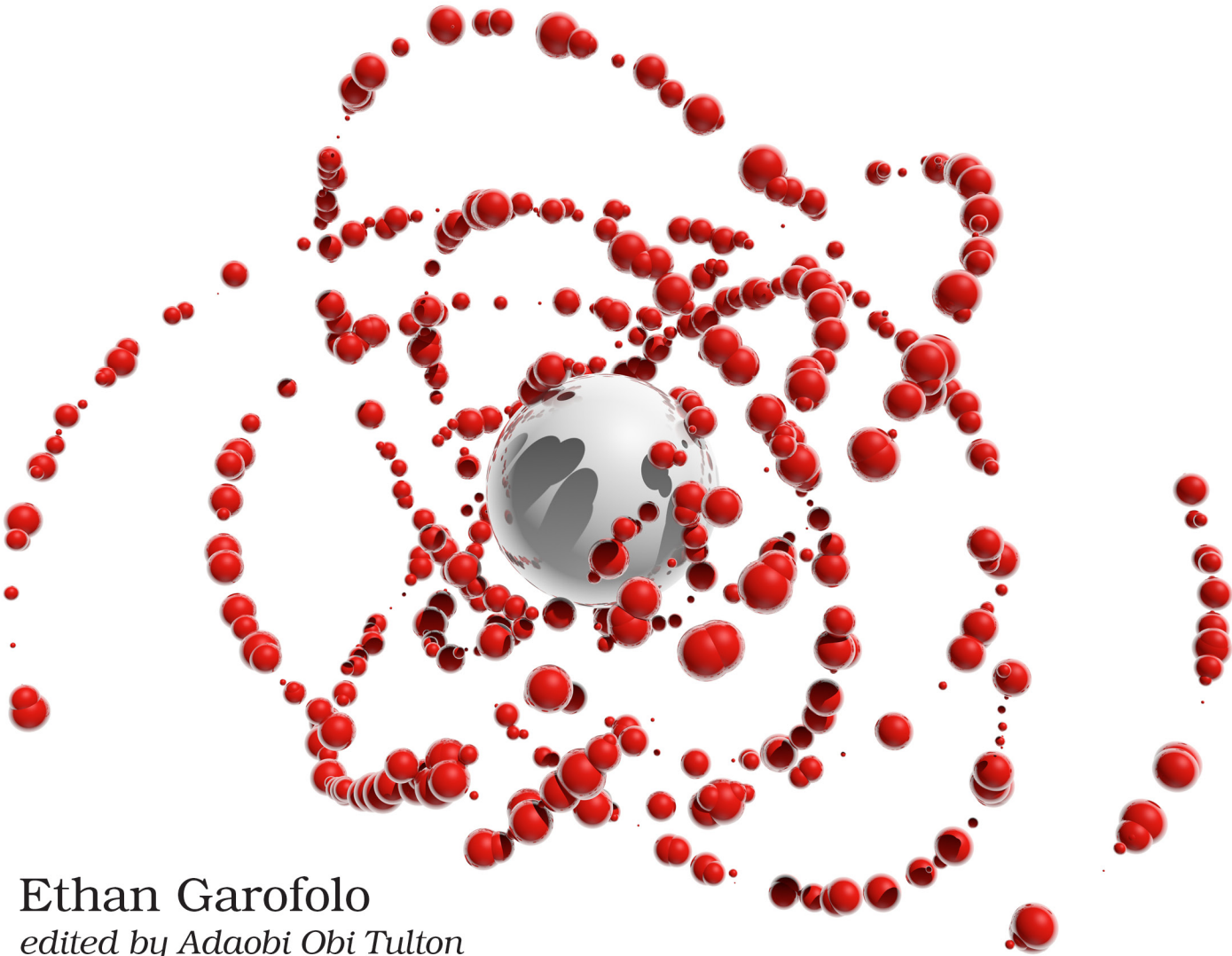
The Pragmatic Bookshelf

Raleigh, North Carolina

The
Pragmatic
Programmers

Practical Microservices

Build Event-Driven Architectures
with Event Sourcing and CQRS



Ethan Garofolo
edited by Adaobi Obi Tulton

Practical Microservices

Build Event-Driven Architectures
with Event Sourcing and CQRS

Ethan Garofolo

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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.

Copyright © 2020 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-645-7

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—April 2020

For Julie, Sofia, Eva, and Angelo—

you are life and joy

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 ?](#).

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 Components, on page ?](#), 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 ?](#).

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 Kafka⁴ 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 never be 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 12](#), 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 ?](#). 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 UUIDs⁶ 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 ?](#), an entity stream only has a single writer.

Here is a pair of such identity streams:

Stream: identity-cd5cb686-e841-4ade-9d4f-dd7baaac2dc6

Registered	AccountLocked	AccountUnlocked
------------	---------------	-----------------

Stream: identity-b6da5cea-f7aa-40ec-b4f7-c788501ff91d

Registered

There are other kinds of streams, though. If all goes as planned, Video Tutorials 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 everything 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 ?](#), 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:command`, 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 ?](#).

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 Component authorized to *write* to streams in the identity category. Those are Component 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 distributing our system in [Chapter 12, Deploying Components, on page ?](#), 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 paragraph. A category has a single owner, even on the rare occasions that a particular owner happens to own another category.

Recording Video Views

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 9](#) 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 maintainability. 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 verification, 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 ?](#).

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 11](#) 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
  }
}
```

- ❶ 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.
- ❷ 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 7](#). There isn't that much to an event.
- ❸ Next, we construct the name of the stream that we're going to write this event to.
- ❹ 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 = createAction({
    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
Line 1 // ...
- const createPostgresClient = require('./postgres-client')
- const createMessageStore = require('./message-store')
- function createConfig ({ env }) {
5   const knexClient = createKnexClient({
-     connectionString: env.databaseUrl
-   })
-   const postgresClient = createPostgresClient({
-     connectionString: env.messageStoreConnectionString
10  })
-   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 environment. We'll add that to `env.js` and `.env` is just a moment. Line 11 then instantiates 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 anything 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 autonomous

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.