Extracted from:

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS

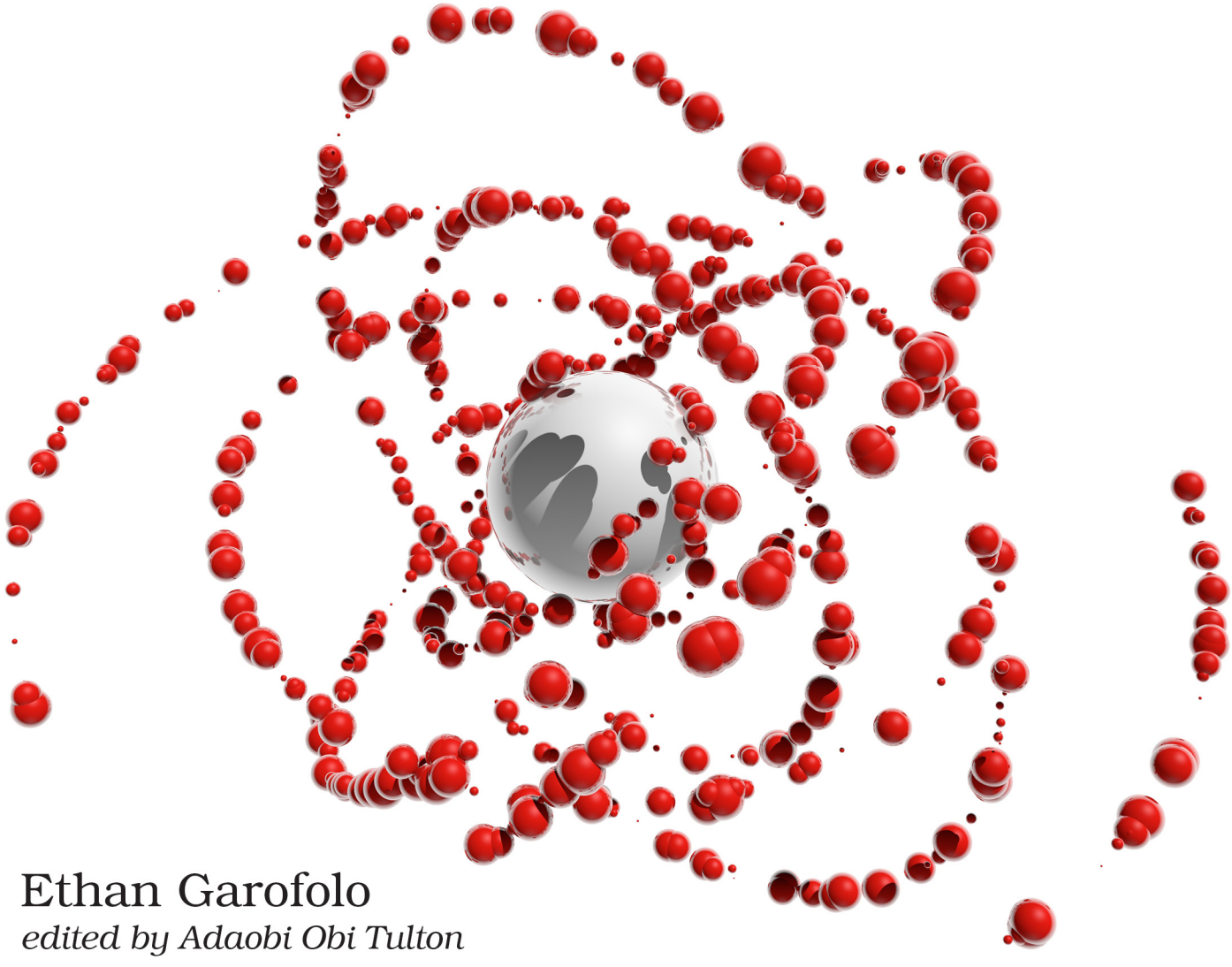
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS



Ethan Garofolo

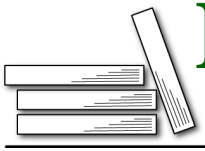
edited by Adaobi Obi Tulton

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS

Ethan Garofolo

The Pragmatic Bookshelf
Raleigh, North Carolina



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*.

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
Book version: B1.0—June 26, 2019

Handling Events

An aggregator's existence is define by a two-step process:

1. Receive an event
2. Handle it by updating a read model somewhere

In our case, we're only going to use PostgreSQL tables for read models, but as you'll see, we're no limited to only relational tables.

(Re)Introducing the RDBMS

Okay, so technically we used a relational database when writing to our message store Chapter 3, Putting Data in a Message Store, on page ?. That was optimized for *writing* data. In this chapter we're going to build tables optimized for *reading* data. You won't find third normal form[1] tables here. Those are optimized for writing. What if we could build our database schema in such a way that every screen only required querying for *a single row?* Let's start with a migration to define the schema we're going to write to:

video-tutorials/migrations/20180303013723_create-pages.js

```
exports.up = knex =>
  knex.schema.createTable('read_model_pages', table => {
    table.string('page_name').primary()

    table.jsonb('page_data').defaultsTo('{}')
  })

exports.down = knex => knex.schema.dropTable('read_model_pages')
```

Two columns, and one of them is a JSON blob. The idea is that there are some mostly static pages on our site. The home page is an example. We want to get the data for these pages with a single query and no joins.

To that end, this migration creates a two-column table that houses key-value pairs. The keys are the strings in the `page_name` column, and the values are `jsonb` objects that default to the empty object but that will be filled out with the data needed to render the pages they represent. For the home page, that might be something like having a `page_name` and `page_data` of `home` and `{ "videosWatched": 42, "lastViewProcessed": 24 }`, respectively. `videosWatched` is the number of videos watched, and `lastViewProcessed` is how we handle idempotence. It is the `id` of the last message incorporated into the read model. So if we see a message with a lower number or equal number, then we know that the read model already incorporates that message.

1. https://en.wikipedia.org/wiki/Third_normal_form

The table name is a tad verbose. Just like how we prepended message_store_ to the message store tables to keep them separate, we do the same thing with our read models by prepending their tables with read_model. That is just a convention we use in this book to keep the difference excruciatingly obvious.

Writing Your First Aggregator

Since this is the first aggregator, here is the basic shape of one:

video-tutorials/src/aggregators/home-page.js

```
function createHandlers ({ queries }) {
  return {
  }
}

function createQueries ({ db }) {
  return {
  }
}

function createAggregator ({ db, messageStore }) {
  const queries = createQueries({ db })
  const handlers = createHandlers({ queries })
  return {
    queries,
    handlers,
  }
}

module.exports = createAggregator
```

Our aggregators handle messages, so they have handlers. They also interact with a database, so they have queries. There is a top-level function, which we name createAggregator that receives dependencies, namely db and messageStore, references to the database and the message store, respectively. The top-level function passes them to the queries and handlers. This shape is not a hard-fast rule for aggregators in general, but is what most of our aggregators will have.

With that shape in place, let's write our message handlers.

Handling Asynchronous Messages

Message handlers are functions that receive a message and *do something*. For services that'll mean carrying out some state-changing business function. For aggregators that means updating a Read Model.

We define an autonomous component's message handlers as a JavaScript object whose keys are the message types the component handles. This

aggregator needs to handle VideoViewed events, and when we get one, we want to increment the global watch count by 1. So let's write that first handler:

video-tutorials/src/aggregators/home-page.js

```
function createHandlers ({ queries }) {
  return {
    VideoViewed: event => queries.incrementVideosWatched(event.id)
  }
}
```

createHandlers receives the queries from the top-level function, and returns an object with key VideoViewed, whose value is a function that takes an event and delegates the appropriate action to queries.incrementVideosWatched. At a glance, we can tell how this aggregator handles this event, and that's good. Let's write that query function:

video-tutorials/src/aggregators/home-page.js

```
function incrementVideosWatched (id) {
  const queryString = `
    UPDATE
      read_model_pages
    SET
      page_data = jsonb_set(
        jsonb_set(
          page_data,
          '{videosWatched}',
          ((page_data ->> 'videosWatched')::int + 1)::text::jsonb
        ),
        '{lastViewProcessed}',
        :id::text::jsonb
      )
    WHERE
      page_name = 'home' AND
      (page_data->>'lastViewProcessed')::int < :id
  `

  return db.then(client => client.raw(queryString, { id }))
}
```

Oof. That's a gnarly query if you're unfamiliar with PostgreSQL jsonb columns, but we can work through it. It has the same structure as any UPDATE query you've worked with before:

```
UPDATE
  read_model_pages
SET
  -- the jsonb part
WHERE
  page_name = 'home' AND
  (page_data->>'lastViewProcessed')::int < :id
```

It's doing an UPDATE against the read_model_pages. It SETs something that we ignore until next paragraph, and it only does it on rows WHERE certain criteria are met. Those criteria are first that the page_name column equals home. Second, we're going to go into the page_data json and make sure its lastViewProcessed property, which we'll explicity treat as an integer, is greater than the id of the event we're handling.

Now, what in tarnation are we SETting? It's actually two calls to PostgreSQL's jsonb_set[2] function. jsonb_set works similarly to JavaScript's Object.assign that we use throughout the book. Let's consider the inner call first:

```
jsonb_set(
  page_data,
  '{videosWatched}',
  ((page_data ->> 'videosWatched')::int + 1)::text::jsonb
),
```

The first argument page_data means we're operating on the page_data column. This is likely *not* a surprise since this is a two-column table, and the other column is not a jsonb column. We setting a property on the object stored in this column. What property? That's the second argument, {videosWatched}.

Now, what value are we going to set it to? Take the videosWatched property of the page data column, page_data ->> 'videosWatched', and cast it to an integer. PostgreSQL doesn't know that this is an integer property, so we tell it that it is by adding ::int, getting us to (page_data ->> 'videosWatched')::int. Then add 1 to it, or (page_data ->> 'videosWatched')::int + 1.

Next, we have to do some more casting because this columns stores jsonb and not integers. Unfortunately, we can't convert directly from integers to jsonb, so we first cast it all to text, ((page_data ->> 'videosWatched')::int + 1)::text, and then finally from text to jsonb, ((page_data ->> 'videosWatched')::int + 1)::text::jsonb. Equivalent JavaScript would be:

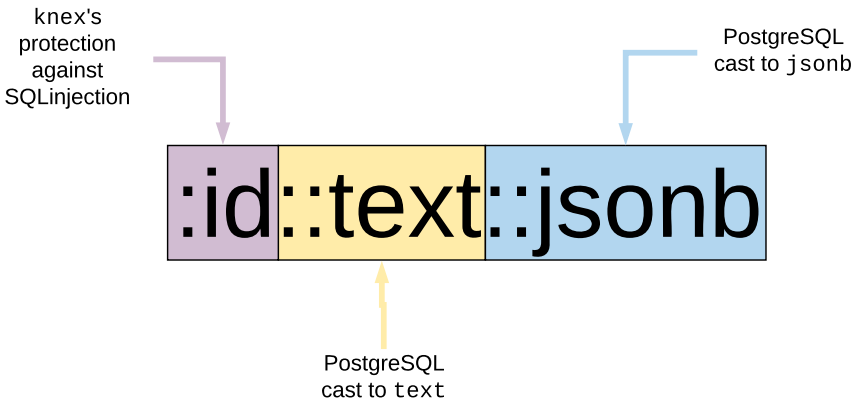
```
const pageData = {
  videosWatched: 0,
  lastViewProcessed: 0
}

const videosWatchedUpdate = {
  videosWatched: pageData.videosWatched + 1
}

const result = Object.merge({}, pageData, videosWatchedUpdate)
```

Now, here's the isolated outer call:

2. https://www.postgresql.org/docs/11/functions-json.html

```
jsonb_set(
  result_of_inner_call,
  '{lastViewProcessed}',
  :id::text::jsonb
)
```

It is similar, only instead of starting with the value in page_data, we start with the value that results from updating the videosWatched count. The return value of the inner call becomes the starting point for the second call. This time we're updating the lastViewProcessed property, and we're setting it to the id of the event we're processing. But again, we have to cast it to text and then again to jsonb. There are a lot colons in that last argument, so here's a visual breakdown of them:



We have :id because of knex—we'll bind that to the value of the event's id. Then there's ::text to get to text and finally ::jsonb to get to jsonb.

Getting Idempotent with It

If you were casually reading along up to this point, this is a heads up that what we're about to say is possibly the most important topic when working with microservices. We're going to talk about *idempotence*.

The word "idempotence" literally means "same power"[3], and the idea is that if a function is idempotent, there are only two states that matter—it having been called zero times, and it having been called one or more times. Additional calls have no additional side effects.

3. https://en.wikipedia.org/wiki/Idempotence

This is as important to microservices as oxygen is to you, dear reader. Messaging. Systems. Fail. You *will* see the same message more than once, and it is physically impossible to guarantee exactly-once delivery. As software developers, we build our abstractions in the sky, but ultimately all of our programs execute on physical hardware. So, *you* as the consumer of messages in a message-based architecture must account for the fact that you'll eventually see the same message more than once. You must write idempotent message handlers. Go idempotent, or go home, as they say.

This handler is idempotent because of the way the increment query is written. Notice the WHERE clause. Every event the aggregator processes will go through this query, and the query only updates rows whose lastViewProcessed property is less than the id of the current event. So, if we see an event a second time, lastViewProcessed will be equal to or greater than said event, and the query becomes a *no-op*[4]. Call it as many times as you want, we're only going to increment the count once for a given message.

As we write additional aggregators and start branching into services, we'll see other idempotence patterns. It isn't always as simple as it was here, but every message handler we write will be idempotent.

Connecting to the Live Message Flow

Now that we have an aggregator, we need to hook it up to the live flow of messages. An aggregator is meant to be constantly running, picking up messages more or less as they occur. To hook this one up to that flow, we head back to the top-level function:

video-tutorials/src/aggregators/home-page.js

```
Line 1  function createAggregator ({ db, messageStore }) {
        -   const queries = createQueries({ db })
        -   const handlers = createHandlers({ queries })
        -   const subscription = messageStore.createSubscription({
        5     stream: 'viewings',
        -     handlers,
        -     subscriberId: 'aggregators:home-page'
        -   })
        -
       10   function init () {
        -     return queries.ensureHomePage()
        -   }
        -
        -   function start () {
       15     init().then(subscription.start)
```

4. https://en.wikipedia.org/wiki/NOP

```
  }

  return {
    queries,
    handlers,
    init,
    start
  }
}
```

Line 4 calls messageStore.createSubscription. Much like when we introduced 'messageStore.write' on page ?, createSubscription doesn't exist yet. We'll write that in the next chapter, Chapter 5, Subscribing to the Message Store, on page ?. For now, we know it as a function that takes three things:

1. A stream to subscribe to. When you hook into the live flow of messages, you do so by observing a particular stream. We'll generally subscribe to category streams like we do here.
2. handlers to handle (in an idempotent manner!) the messages on that stream. As we said on page 4, we represent handlers as a JavaScript object whose keys are the message types we handle.
3. A globally unique subscriberId. This is how this aggregator identifies itself to the message store.

Merely creating a subscription doesn't actually start the flow of messages, however—that's what the start function at line 14 is for. We're taking the convention that every autonomous component must expose a start function to actually begin its polling cycle. We don't want that cycle to start in test, for example.

This start function has one piece of work to do before releasing the message hounds. queries.incrementVideosWatched, which you wrote on page 5, assumes that the row it's going to update exists. This assumption is a lot easier than checking to see if it exists every time we process a message, but it does mean we need to put that row in place. So, start calls init, which in turn calls queries.ensureHomePage:

video-tutorials/src/aggregators/home-page.js

```
function ensureHomePage () {
  const initialData = {
    pageData: { lastViewProcessed: 0, videosWatched: 0 }
  }

  const queryString = `
    INSERT INTO
      read_model_pages(page_name, page_data)
    VALUES
```

```
      ('home', :pageData)
    ON CONFLICT DO NOTHING
  `

  return db.then(client => client.raw(queryString, initialData))
}
```

This function sets up what this row looks like before we've seen any messages and then inserts it into the data base using ON CONFLICT DO NOTHING. We'll insert this row exactly once, no matter how many times we start this aggregator.

Congrats! You just wrote your first aggregator. You took stream of VideoViewed events and turned them into a Read Model that the home page application can use. You could make up additional aggregations, and in fact, the exercises at the end of this chapter will include a challenge to do that. This is where message-based architectures get so interesting. You can slice and dice the same source data into whatever shape is required. And since you were saving all that source data, you can do this going all the way back to when you first turned the system on.

Okay, break time is over, we still have a little work to do to connect this the running system.

Configuring the Aggregator

We need to pull this aggregator into config.js and modify src/index.js so that it calls the aggregator's start function. config.js first, then start:

video-tutorials/src/config.js

```
// ...
const createHomePageAggregator = require('./aggregators/home-page')
function createConfig ({ env }) {
  // ...
  const homePageAggregator = createHomePageAggregator({ db, messageStore })
  const aggregators = [
    homePageAggregator,
  ]
  const services = [
  ]
  return {
    // ...
    homePageAggregator,
    aggregators,
    services,
  }
}
```

We start by requireing the aggregator. Then inside of createConfig we instantiate it by passing it the db and messageStore reference that were instantiated in code

represented by the ellipses—we won't keep reprinting the configuration from previous chapters. The we set up an array named aggregators and put homePageAggregator in it. We'll use this array to start all of our aggregators. Since we're here in the file, we also make a similar array for services. It's empty for now because we won't write our first service until Chapter 6, Registering Users, on page ?. Lastly, we add homePageAggregator, aggregators, and services to config's return object.

Now that these pieces are configured, we can start this aggregator in src/index.js:

video-tutorials/src/index.js
```
function start () {
  config.aggregators.forEach(a => a.start())
  config.services.forEach(s => s.start())
  app.listen(env.port, signalAppStart)
}
```

To the app.listen call that starts the Express application, we added a couple of lines to start all the aggregators and services. config.aggregators.forEach loops over the aggregators array we set up in config.js and calls each aggregator's start function. It does the same thing for services, which at this point is empty.

And just like that, you have an aggregator that is configured to connect to the live flow of messages and aggregate a Read Model the home page Application can use to show the global videos watched count.

Having the Home Page Application Use The New Read Model

Speaking of the home page Application, it currently isn't using our aggregator's output. Let's fix that:

video-tutorials/src/app/home/index.js
```
function createQueries ({ db }) {
  function loadHomePage () {
    return db.then(client =>
      client('read_model_pages')
        .where({ page_name: 'home' })
        .limit(1)
        .then(camelCaseKeys)
        .then(rows => rows[0])
    )
  }

  return {
    loadHomePage
  }
}
```

We just need to modify the `loadHomePage` query. Instead of querying the monolithic `videos` table, we're going to query the special-purpose `read_model_pages` table. We want the one where `page_name` is equal to `home`. Notice how there was so summation to do, no joins. We built a laser-focused read model to serve this particular page. If we have other Read Model needs, we can build them using all the same events, and the home page Application won't have to change one wit. That's the power of autonomy.