The Pragmatic Programmers

Your Elixir Source

# Real-World Event Sourcing

## Distribute, Evolve, and Scale Your Elixir Applications

Kevin Hoffman

*edited by Kelly Talbot*

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

## Handling Errors by Modeling Failure

Let's take this example to the next step. The code you've written so far is actually fairly brittle. For example, the code can easily be broken with a divide by zero error:

```
iex(4)> evt2 = EventSourcedCalculator.V2.handle_command(%{value: -1},
...(4)>  %{cmd: :div, value: 0})
%{event_type: :value_divided, value: 0}
iex(5)> state2 = EventSourcedCalculator.V2.handle_event(%{value: 9}, evt2)
** (ArithmeticError) bad argument in arithmetic expression
    es_calc_v2.exs:31: EventSourcedCalculator.V2.handle_event/2
```

Modeling errors and failure conditions is an enormous topic covered in detail later in the book, so for now you'll only handle a couple of easily managed cases. Create a third version of the calculator with the following code:

esintro/es_calc_v3.exs

```
defmodule EventSourcedCalculator.V3 do
  @max_state_value 10_000
  @min_state_value 0

  def handle_command(%{value: val}, %{cmd: :add, value: v}) do
    %{event_type: :value_added,
      value: min(@max_state_value - val, v)}
  end

  def handle_command(%{value: val}, %{cmd: :sub, value: v}) do
    %{event_type: :value_subtracted,
      value: max(@min_state_value, val - v)}
  end

  def handle_command(
        %{value: val},
        %{cmd: :mul, value: v}
      )
      when val * v > @max_state_value do
    {:error, :mul_failed}
  end

  def handle_command(%{value: _val}, %{cmd: :mul, value: v}) do
    %{event_type: :value_multiplied, value: v}
  end

  def handle_command(
        %{value: _val},
        %{cmd: :div, value: 0}
      ) do
    {:error, :divide_failed}
  end

  def handle_command(%{value: _val}, %{cmd: :div, value: v}) do
    %{event_type: :value_divided, value: v}
```

```
  end

  def handle_event(
        %{value: val},
        %{event_type: :value_added, value: v}
      ) do
    %{value: val + v}
  end

  def handle_event(
        %{value: val},
        %{event_type: :value_subtracted, value: v}
      ) do
    %{value: val - v}
  end

  def handle_event(
        %{value: val},
        %{event_type: :value_multiplied, value: v}
      ) do
    %{value: val * v}
  end

  def handle_event(
        %{value: val},
        %{event_type: :value_divided, value: v}
      ) do
    %{value: val / v}
  end

  def handle_event(
        %{value: _val},
        %{}
      )

  def handle_event(%{value: _val} = state, _) do
    state
  end
end
```

A couple highlights here are that this code enforces an upper and lower bound on the calculator state. Code like this will accept commands that could overflow values by modifying the changed amounts. In the preceding code, the @max_state_value module constant limits the value on the event.

Let's see how this new code behaves:

```
iex(2)> EventSourcedCalculator.V3.handle_command(
...(2)> %{value: 9500}, %{cmd: :add, value: 650})
%{event_type: :value_added, value: 500}
```

This is the first new feature: the command requested that the value 650 be added, but due to the ceiling implementation, only 500 was added. This may

seem like a small detail now, but this kind of behavior will be critically important later. Command handlers should never be considered as merely "blindly converting a command into an event that occurred in the past." Put another way, a command represents a request for something to happen and an event represents what *actually* happened and the two will only ever look identical in the simplest of cases.

Let's see what happens when you try to divide by zero:

```
iex(3)> EventSourcedCalculator.V3.handle_command(
...(3)> %{value: 9500}, %{cmd: :div, value: 0})
{:error, :divide_failed}
```

This is another subtle but important detail. The command was rejected outright, so the code returned an error type of :divide_failed. Since nothing happened as a result of this command, *no event was emitted*. Sometimes it may seem useful to emit an "error event", but that has many consequences. When to emit error events and when to simply reject commands is something that will be covered throughout the book as you model different domains.

This brings up the first event sourcing law:

**All Events are Immutable and Past Tense**

Every event represents something that actually happened. These events cannot be modified and always refer to an event that took place. Modeling the absence of a thing or a thing that didn't actually occur may often seem like a good idea, but it can confuse both developers and event processors.

In order to not break this rule, you need to not store an event for command validation failures. While guidelines for when to emit events for failures are hotly debated, my personal opinion distilled from much suffering is that validation failures are not events.

The critical aspect here is that the event handler is not where validation happens, that happens in a command handler. After all, if the thing already occurred, how can we go back in time to validate it?

Chapter 11, Modeling and Discovering Application Domains, on page ? goes into more detail and provides tips and tricks that will help with modeling failure and other edge cases.

Of course, repeatedly handling commands and applying the resulting events while still maintaining an intermediate result looks an awful lot like a *fold*. In functional programming, a fold is a higher-order function that performs a recursive operation on some list of data, progressively building up a single

return value. Another form of a fold is a *reduce*, which, as the name implies, *reduces* a list of elements into a single element. Operations like summing the numbers in a list or producing a final balance from a list of ledger items can all be represented as a fold.

Indeed, you could say that the state of any event sourced component can be produced by *folding* a function over the event stream. Let's look at how to do that with the sample you've been building so far. Make sure the es_calc_v3.exs file has been compiled and EventSourcedCalculator.V3 imported.

```
iex(3)> cmds = [%{cmd: :add, value: 10},
...(3)> %{cmd: :add, value: 50}, %{cmd: :div, value: 0},
...(3)> %{cmd: :add, value: 2}]
[
  %{cmd: :add, value: 10},
  %{cmd: :add, value: 50},
  %{cmd: :div, value: 0},
  %{cmd: :add, value: 2}
]
iex(4)> initial = %{value: 0}
%{value: 0}
iex(5)> cmds |> List.foldl(initial,
...(5)>  fn cmd, acc -> handle_event(acc, handle_command(acc,cmd)) end)
%{value: 62}
iex(6)>
```

Now, having worked through the exercises in this chapter on your own, hopefully you received the payoff of a nice fresh hit of dopamine. At this point you can declare a sequence of commands that you wish to send to a calculator. You can then fold that list of commands by passing the event produced by the handle_command function (if any) to the handle_event function. The preceding code can be refactored to use more of the pipeline operator, but this syntax is hopefully a bit more easily read.

It is very important that you know what happened to the event produced by %{cmd: :div, value: 0}. As you know, this produces a failure event. Recall the following code from V3 of the calculator:

```
def handle_event(%{value: _val} = state, _) do
  state
end
```

This fallthrough matches any event that the calculator doesn't handle. The calculator doesn't explicitly handle :divide_failed error, so this function picks that up. This brings up the second law of event sourcing:

**Applying a Failure Event Must Always Return the Previous State**

Any attempt to apply a bad, unexpected, or explicitly modeled failure event to an existing state must always return the existing state. Failure events should only indicate that a failed thing occurred in the past, not command rejections.

No matter what happens during the processing of an event, a failure during processing or an event that indicated a failure must always return the state current at the time of processing. This rule is actually what makes an event *stream* possible. If processing an event could break the stream, event sourcing wouldn't work.