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.

Building Your First Projection

The bank account ledger example is one of the easiest to build and understand in event sourcing, so let's start there. In this exercise, you'll build a projection for a bank account balance. In more formal terms, you'll build a *projector* that takes data from events in the event log and stores that data in a read model projection, which can then be queried by consumers.

A projector is a piece of code responsible for producing read model data for consumption by one or more parties. For this sample, the projector is `AccountBalance` and the projection is just a piece of data containing the most up to date balance for a given account. As with aggregates, projections have *keys* that allow you to differentiate one from another. The bank ledger example is easy because it has an obvious key: the account number.

Let's start with the client API for the projector and then move on to the implementation details. Assume that you want to be able to perform the following tasks:

```
iex(1)> Projectors.AccountBalance.apply_event(%{event_type: :amount_deposited,
...(1)> account_number: "NEWACCOUNT", value: 12})
:ok
iex(2)> Projectors.AccountBalance.lookup_balance("NEWACCOUNT")
{:ok, 12}
```

In this code, an event (`:amount_deposited`) is applied to a projector, and then the projected balance is available for query. A subtle point here is that the `apply_event` function doesn't accept a key separate from the event. This is to avoid accidentally applying an event to the wrong projection. You don't want to apply the deposit event to `OLDACCOUNT` if the value of `account_number` on the event is `NEWACCOUNT`. This brings up another law of event sourcing:

All Data Required for a Projection Must Be on the Events

The event is the only source of truth. If code allows a different piece of information to be supplied as a parameter that contradicts information on the event, you can corrupt an entire event stream. As such, all keys, metadata, and payload data must come from events and nowhere else.

Implicit information can also be used by projectors. For example, in many cases, especially when building things like rolling averages, the projector may need to make use of event counts or previously calculated data. If you're thinking that previously calculated data is data not on events, and so it violates the rule you just learned, that's natural. The distinction is subtle and

pedantic, but projectors can maintain their own internal state, also derived from the event log. The key is that this behavior is still referentially transparent, and given the same event log, a projector will always produce the same output.

In this chapter, you'll be creating projectors as GenServers that manage their projections as in-memory data. Later in Chapter 12, Scaling Out the Event Sourcing Building Blocks, on page ?, you'll get to dive deep into techniques for dealing with distributed, persistent, highly-available projections.

But for now, let's focus on simple projections and create an empty scaffold for the GenServer:

```
defmodule Projectors.AccountBalance do
  use GenServer
  require Logger

  def start_link(account_number) do
    GenServer.start_link(
      __MODULE__,
      account_number,
      name: via(account_number))
  end

  @impl true
  def init(account_number) do
    {:ok, %{balance: 0, account_number: account_number}}
  end

  defp via(account_number) do
    {:via, Registry,
      {Registry.AccountProjectors, account_number}}
  end
end
```

This is enough to call Projectors.AccountBalance.start_link("my_account") and get back a pid. This server maintains a projection that has the account number and the current balance. In order to trigger this server to update the balance, let's add some functions for application of events.

```
def apply_event(%{account_number: account} = event)
    when is_binary(account) do
  case Registry.lookup(Registry.AccountProjectors, account) do
    [{pid, _}] ->
      apply_event(pid, event)
    _ ->
      Logger.debug(
        "Attempt to apply event to non-existent account, starting projector")
      {:ok, pid} = start_link(account)
      apply_event(pid, event)
  end
```

```
end

def apply_event(pid, event) when is_pid(pid) do
  GenServer.cast(pid, {:handle_event, event})
end

@impl true
def handle_cast({:handle_event, evt}, state) do
  {:noreply, handle_event(state, evt)}
end

def handle_event(%{balance: bal} = s,
                 %{event_type: :amount_withdrawn, value: v}) do
  %{s | balance: bal - v}
end

def handle_event(%{balance: bal} = s,
                 %{event_type: :amount_deposited, value: v}) do
  %{s | balance: bal + v}
end

def handle_event(%{balance: bal} = s,
                 %{event_type: :fee_applied, value: v}) do
  %{s | balance: bal - v}
end
```

A couple of interesting things are going on here. The first is that if an event arrives for an account that doesn't yet have a running projector, the code starts one with an initial balance of `0`. There will always be one server per account, each maintaining its own projection. An Elixir process registry is used to manage the list of all projectors, keyed on the account number.

Next, let's add some code to allow consumers to query the balance of any given account:

```
def lookup_balance(account_number) when is_binary(account_number) do
  with [{pid, _}] <-
    Registry.lookup(Registry.AccountProjectors, account_number) do
    {:ok, get_balance(pid) }
  else
    _ ->
      {:error, :unknown_account}
  end
end

@impl true
def handle_call(:get_balance, _from, state) do
  {:reply, state.balance, state}
end
```

The registry[1] (note you'll have to start this explicitly via start_link as shown in the next example, or through a supervision tree) makes for an ideal way of

1. https://hexdocs.pm/elixir/1.12/Registry.html

locating the process responsible for the account balance projection for a given account. If there's no running projector, then the balance query will return {:error, :unknown_account}.

With all of this in the module, exercise what you've built so far:

```
iex(1)> {:ok, _} = Registry.start_link(keys: :unique,
...(1)> name: Registry.AccountProjectors)
{:ok, #PID<0.304.0>}
iex(2)> c("balance_projector.exs")
...(3)> Projectors.AccountBalance.apply_event(
...(4)>   %{event_type: :amount_deposited,
...(5)>   account_number: "NEWACCOUNT", value: 12})
:ok
iex(6)> Projectors.AccountBalance.apply_event(
...(7)>   %{event_type: :amount_deposited,
...(8)>   account_number: "NEWACCOUNT", value: 30})
:ok
iex(9)> Projectors.AccountBalance.lookup_balance("NEWACCOUNT")
{:ok, 42}
iex(10)> Projectors.AccountBalance.lookup_balance("DOESNOTEXIST")
{:error, :unknown_account}
```

That's all there is to it! The state updating aspect of projectors is easy. It doesn't take much effort to perform some basic business logic and store the result. In real applications, instead of manually calling apply_event, you would likely see a dispatch or routing system that funnels events from a stream through projectors.

The more challenging part of projectors, as you'll see as you go through more advanced exercises, is designing the events and projections and working within the many constraints enforced by event sourcing.

Aggregate State vs Projections

Aggregates maintain their own state. This state must be considered private and is only ever used to validate commands sent to the aggregate and populate outgoing events. This differs from projections, which are designed to be shared, communal views accessible to any consumer that knows how to find and interpret them. In simple cases, aggregate state and projection data may be indistinguishable, but resist the temptation to combine them or violate role boundaries because doing so can remove the predictable and repeatable aspects of an event sourced system.

Let's move on to a more sophisticated, and realistic, example.