The Pragmatic Programmers

Your Elixir Source

Real-World Event Sourcing

Distribute, Evolve, and Scale Your Elixir Applications

Kevin Hoffman

edited by Kelly Talbot

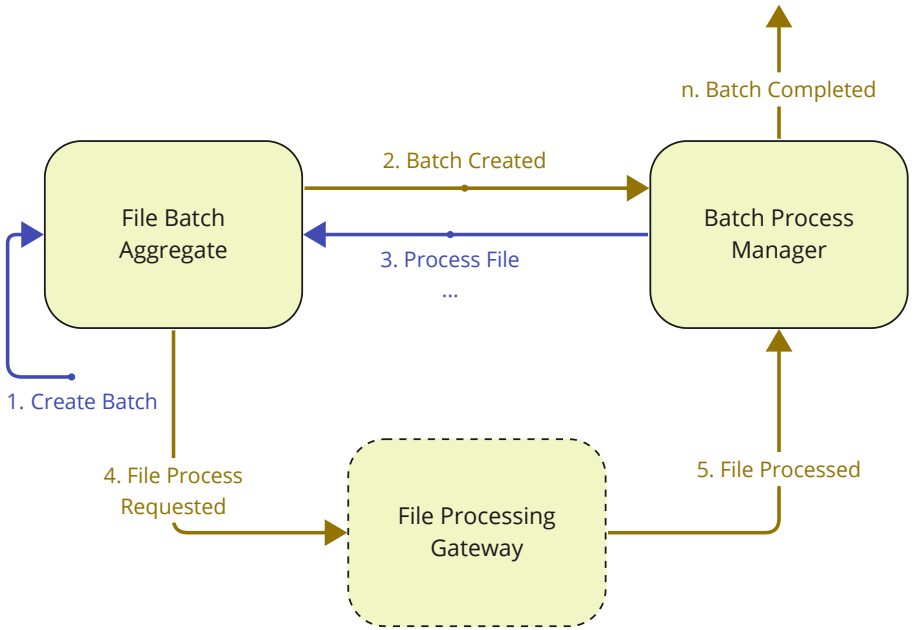
This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

Creating a Simple Process Manager

The canonical "hello world" sample of process managers is usually a batch processor. You're going to build a process manager that advances the handling of a batch of files. The nature of these files or what needs to be done to them isn't really relevant for this exercise.

The process starts when an initiating event, batch_created, is received. From there, the process manager issues a single command requesting work from an aggregate for each of the files in the batch. Thereafter, the process manager keeps tabs on the processing status of each of the files, as shown in the following diagram:



The flow through the system in an error-free path might be:

1. A batch is created via the create_batch command.
2. The file batch aggregate then emits the batch_created event.
3. *The process manager creates and dispatches one command per file on the batch.*
4. The file batch aggregate (not coded for this sample) emits a file processing requested event, which is picked up by a notifier/gateway.

5. When the file processing (external) is completed, the injector/gateway dispatches the file_processed event.
6. *The process manager updates its internal state upon receipt of file processed events.*

While this might seem like a lot of busywork, you'll only be creating the process manager for this flow and assuming that the rest of the system has already been taken care of. Before creating the next bit of code, make sure to take note of the next law of event sourcing:

Work is a Side Effect

A frequently asked question in new event sourcing projects is "where does the work happen? Aggregates are not allowed to perform side effects or read from external data. Process managers are not allowed to perform side effects or read from external data. Projectors can create external data, but they can't perform "work" either.

If you follow the rule that work is a side-effect, things may be easier to understand. If work is a mutation of the world outside the event sourced system, then it's a side effect, and side effects are only allowed through gateways. The core primitives of aggregates, projectors, and process managers must never do work.

For this sample, create a GenServer that reacts to the batch_created and file_processed events. You can create an application to hold this GenServer by typing mix new batch --sup in your favorite shell. Add a process_manager.ex file to lib/batch:

chap4/batch/lib/batch/process_manager.ex

```
defmodule Batch.ProcessManager do
  use GenServer

  def start_link(%{id: _id} = state) do
    GenServer.start_link(__MODULE__, state)
  end

  def init(%{id: id}) do
    {:ok,
     %{
       id: id,
       files: %{},
       status: :idle
     }}
  end

  def handle_call({:process_event, evt}, _from, state) do
    handle_event(state, evt)
  end
```

```
defp handle_event(
       state,
       %{
         event_type: :batch_created,
         files: files
       }
     ) do
  f = Enum.map(files, fn f -> {f, :pending} end) |> Map.new()

  state = %{
    state
    | files: f,
      status: :created
  }

  reply =
    Enum.map(files, fn f ->
      %{
        command_type: :process_file,
        file: f
      }
    end)

  {:reply, reply, state}
end

defp handle_event(state, %{
       event_type: :file_processed,
       file: %{id: file_id, status: file_status}
     }) do
  files = Map.put(state.files, file_id, file_status)

  state = %{
    state
    | files: files,
      status: determine_status(files)
  }

  # To add functionality we could send retry commands for those
  # files that have failed

  {:reply, [], state}
end

defp determine_status(file_map) do
  cond do
    Enum.all?(
      file_map,
      fn {_f, status} -> status == :success end
    ) ->
      :success

    Enum.any?(
      file_map,
      fn {_f, status} -> status == :error end
```

```
      ) ->
        :error

      true ->
        :pending
    end
  end
end
```

The code in this process manager implements the process flow outlined in the preceding diagram. The process manager GenServer is created by calling start_link. Note that in this example the starting or provisioning of an OTP process is done out of band from the event handling. This is just to keep things simple for the example. How loosely or tightly coupled the lifetime of a process manager is to the lifetime of an OTP process (or other kind of process/thread) depends entirely on which frameworks and tools you're using. You'll encounter examples of multiple types of server lifetimes throughout the book.

The process manager begins its life with the status of :idle and, after receiving a batch created event, quickly switches to :created. In response to this batch created event, the process manager returns an array of commands: one for each file that needs to be processed.

It then listens for :file_processed events coming back from the file processing gateway and continues to adjust its internal state accordingly. The biggest piece of business logic contained in this process manager is the determination of the status of the batch process. The process is considered successfully completed if all files were processed successfully, it's in error if one or more files failed processing, or otherwise the status is :pending.

Now use iex to interact with the sample process manager (line feeds in commands are for human-friendly output only and not part of the command):

```
$  iex -S mix
iex(1)> {:ok, pid} = Batch.ProcessManager.start_link(%{id: "batch1"})
{:ok, #PID<0.194.0>}
iex(2)> GenServer.call(pid,
  {:process_event, %{event_type: :batch_created, files: ["f1", "f2", "f3"]}})
[
  %{command_type: :process_file, file: "f1"},
  %{command_type: :process_file, file: "f2"},
  %{command_type: :process_file, file: "f3"}
]
iex(3)> GenServer.call(pid, {:process_event,
  %{event_type: :file_processed, file: %{id: "f1", status: :success}}})
[]
iex(4)> GenServer.call(pid, {:process_event,
  %{event_type: :file_processed, file: %{id: "f2", status: :success}}})
```

```
[]
iex(5)> GenServer.call(pid, {:process_event,
  %{event_type: :file_processed, file: %{id: "f3", status: :success}}})
[]
iex(6)> :sys.get_state(pid)
%{
  files: %{"f1" => :success, "f2" => :success, "f3" => :success},
  id: "batch1",
  status: :success
}
```

In the preceding iex session, you took on the role of a "meatspace aggregate" and manually converted the commands emitted by the process manager into events. In a complete system, the aggregate would automatically have processed those commands. Responding to each :file_processed event advances the overall state of the process manager. When all files have been processed, the manager's status moves to :success.

Depending on your application's needs, the completed process manager state might be marked for deletion or cleanup, or you might want to leave the state around for historical queries. If you want to periodically "sweep" your process manager state to clean it up, but you want summary data to remain, then you will need to create a projector that emits the summary data. Another law of event sourcing has appeared!

All Projections Must Stem from Events

Every piece of data produced by any projector must stem from at least one event. You cannot ever create projection data out of band of the event stream. Doing so would violate other event sourcing rules and ruin your system's ability to participate in replays.

This law also means never producing projections at timed intervals unless you're injecting stimulus events.

So far, so good. This should give you an idea of the kind of code that typically goes into a process manager. At this point, it would be easy to start adding more state and managing other semi-related activities within this one module, but that would violate the next law of event sourcing.

Never Manage More Than One Flow per Process Manager

Each process manager is responsible for a single, isolated process. Its internal state represents that of an instance of that managed flow, e.g. "Order 421" or "Batch 73" or "New User Provisioning for User ABC". As tempting as it may be to create a process manager for "orders" or "users", never lump multiple process flows into a

Never Manage More Than One Flow per Process Manager
single manager. Doing this generally means the failure of one flow can cascade out throughout the system. Keeping flows separate also avoids accidentally corrupting one process state with that of another.

Next, let's go through the process of modeling another sample domain where you'll build a slightly more complex process manager.