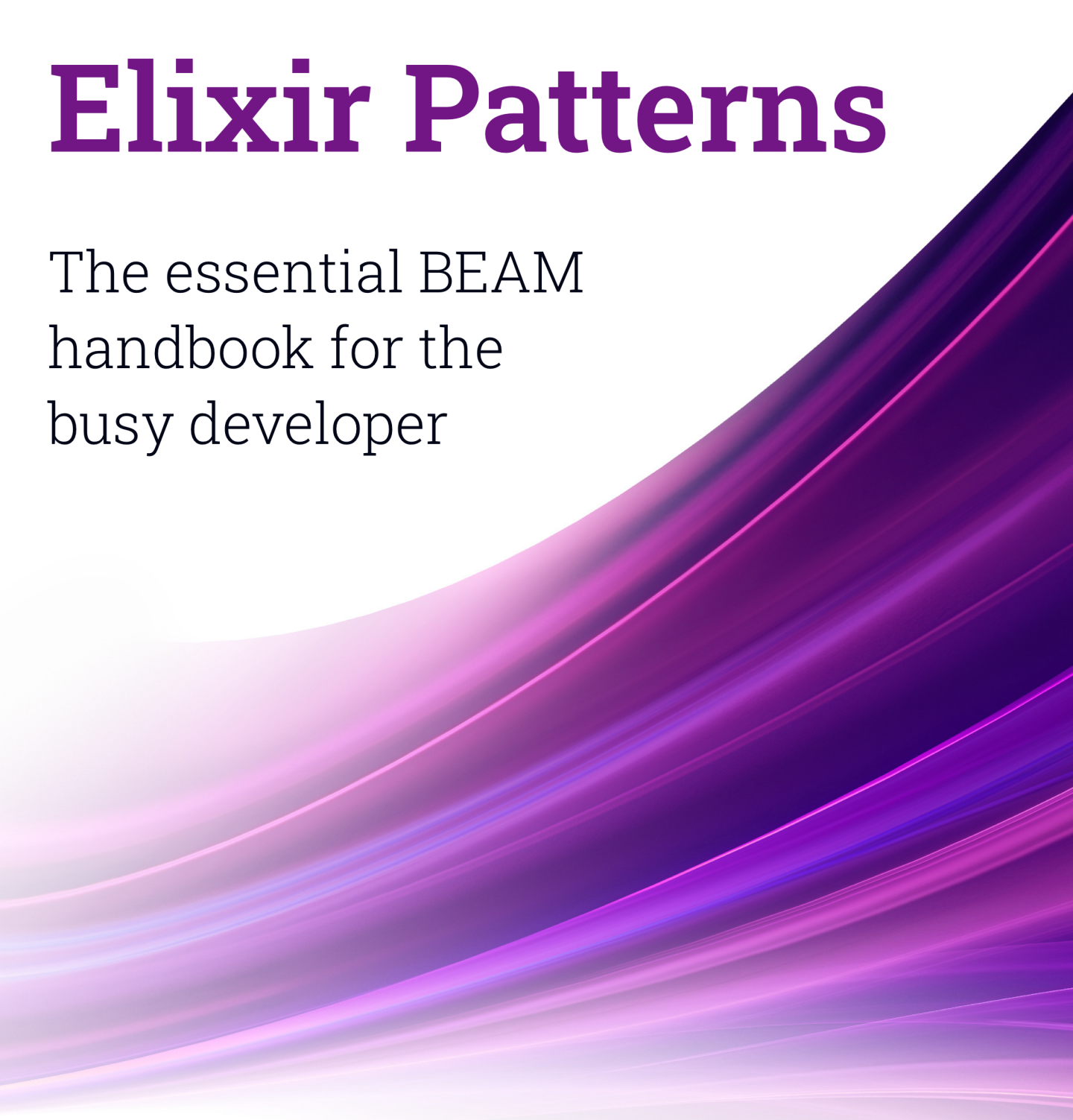
Elixir Patterns

The essential BEAM handbook for the busy developer



Alexander Koutmos
Edited by Hugo Baraúna

Contact team@elixirpatterns.dev for regarding errata and support

Alexander Koutmos
Visit my website at https://akoutmos.com

Hugo Baraúna
Visit my website at https://elixir-radar.com

Printed in the United States of America

First Printing: March 2025
Printed by KDP

ISBN 979-83-07689-76-9

6. Supervisor Initialization Patterns

6.1. Introduction

Now that you have had the chance to experiment with the `Supervisor` and `PartitionSupervisor` modules, it is time to dive into some of the more advanced Elixir process orchestration tools available to you in Elixir. To start, you'll learn how you can use GenServers in order to perform initialization tasks for your application and supervision trees. You will learn how you can perform these initialization tasks in both a synchronous and asynchronous fashion and where each technique is applicable. Finally, we will cover how you can conditionally start your GenServer processes depending on the run-time configuration so you can have fine-grained control over what processes are started.

Before diving into the code, let's first discuss in what situations these techniques are useful and the various ways that we can go about solving these problems using the tools available to us in OTP.

6.2. When Can I Use Initialization Processes?

Supervision tree initialization processes are useful in a wide array of scenarios when you need to perform some unit of work as part of the supervision tree startup. If you recall from previous chapters, it is important to think of supervision trees as a way to bundle related components of your application together so that they can act as a cohesive unit.

To that end, it is often useful to initialize resources, or perform some set up prior to

certain processes starting in your supervision tree. This can include hydrating state inside of :persistent_term, ETS, making HTTP calls, firing off :telemetry events, etc. Whatever your specific need may be, it is often necessary that you perform some unit of work prior to your supervision tree continuing its start-up procedure.

Luckily, Elixir and the BEAM provide a few options for performing startup operations for your supervision trees and applications. We'll cover how you can asynchronously emit a telemetry event using the `Task` module as part of the startup of a supervision tree. This pattern is useful when the initialization work you need to perform is independent from any other processes in your supervision tree. In other words, the other processes in your supervision tree do not depend on the side effects produced by this initialization job.

Let's jump into the code and see what an asynchronous initialization job looks like.

6.3. Running Asynchronous Initialization Jobs

In this example, we will be creating a simple supervision tree, and as part of the initialization process of the supervision tree, we will emit a telemetry event. This telemetry event will contain the information passed to the supervision tree, which can then be used by telemetry consumers to log important information.

As an aside for those unfamiliar with the Telemetry library, it is a simple (and widely used) solution for dynamically emitting/subscribing to application and library events. For example, Ecto and Phoenix leverage the Telemetry library in order to emit metrics and metadata related to database queries and HTTP requests. These metrics and metadata contain things such as timings for requests, the database table that was acted upon, the Phoenix route that was requested and other pieces of information related to the action performed by the libraries.

While we are using the Telemetry library for emitting events in this example, the work that you perform can vary depending on the problem that you are trying to solve. We have seen this technique used for notifying service orchestration platforms that the application is up and running, sending an event to a monitoring system that

the application has successfully started, and several other instances where we need to perform some sort of side effect asynchronously.

That said, let's put together a simple supervision tree that starts a couple of processes and finally fires off an asynchronous Task to let event subscribers know that the supervision tree initialization is complete. We'll start by defining the modules associated with the supervision tree and then define the supervision tree.

Listing 97. Simple GenServer implementation

```
defmodule SimpleGenServer do
  use GenServer

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

  def child_spec(init_arg) do ①
    Supervisor.child_spec(
      %{
        id: init_arg,
        start: {__MODULE__, :start_link, [init_arg]}
      },
      []
    )
  end

  @impl true
  def init(state) do ②
    IO.puts("Starting GenServer: #{state}")

    {:ok, state}
  end
end
```

① We override the autogenerated GenServer `child_spec/1` implementation to control

the `:id` value for the GenServer process.

② We output the name of the GenServer to see when and what processes start.

This simple GenServer provides us a basic process that we can add to our example supervision tree without doing much work beyond that. The internals of the GenServer are not necessarily important here, as we simply need a process to start. Next, let's create a Task module responsible for emitting our init event via `:telemetry`.

Listing 98. Telemetry Task module implementation

```
defmodule InitTelemetryTask do
  use Task

  def start_link(metadata) do
    Task.start_link(__MODULE__, :run, [metadata]) ①
  end

  def run(metadata) do ②
    IO.puts("Running Telemetry Task")

    metrics = %{system_time: System.system_time()}
    :telemetry.execute(init_event(), metrics, metadata) ③
  end

  def init_event do ④
    [:simple, :supervisor, :init]
  end

  def simple_handler(_event_name, measurements, metadata, _config) do ⑤
    IO.puts("Measurements: #{inspect(measurements)}")
    IO.puts("Metadata: #{inspect(metadata)}")
  end
end
```

① We specify that we want to execute the `run/1` function in this module when the

Task is started.

② Output that the function has been invoked and emits the configured telemetry event.

③ Using the `:telemetry.execute/3` function, we can emit an event with the signature defined in `init_event/0`.

④ We put the signature of the Telemetry event in a function so that the source of truth for the event signature is the module that emits the event.

⑤ Our Telemetry event handler simply output the metadata and measurements associated with the event.

We begin the `InitTelemetryTask` module by making use of the use `Task` macro. This macro will automatically generate a `child_spec/1` function in the module so that we can easily add it to our supervision tree. We then define a `start_link/1` function that starts the process and passes through all the provided metadata we want as part of the telemetry event. We also pass to `Task.start_link/3` the name of the function that we want to execute (in this case :run). In the `run/1` function, we then call `:telemetry.execute/3`, which will broadcast the event defined in the `init_event/0` function.

Best practices for using `:telemetry`

As an aside, it makes it easier to refactor and change your telemetry event signatures if they are defined in a function and co-located with the modules that broadcast the event. This way, telemetry event consumers can call your function in order to get the correct signature, and you do not have hard-coded event signatures scattered throughout your codebase.

With that in place, it is time to put together our supervision tree that will start three GenServer processes and also invoke our telemetry Task module.

Listing 99. Supervisor implementation with async initialization process

```
defmodule SimpleSupervisor do
```

```
  use Supervisor

  def start_link(init_arg) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  @impl true
  def init(init_arg) do
    IO.puts("Starting SimpleSupervisor")

    init_task_meta = %{ ①
      init_args: init_arg,
      pid: self()
    }

    children = [ ②
      {SimpleGenServer, :gen_server_one},
      {SimpleGenServer, :gen_server_two},
      {SimpleGenServer, :gen_server_three},
      {InitTelemetryTask, init_task_meta}
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

① Here, we collect all of the metadata associated with the initialization of the `SimpleSupervisor` supervision tree for use by our Telemetry event handler.

② We start three instances of the `SimpleGenServer` process and then asynchronously start the `InitTelemetryTask` process , which emits the Telemetry event containing the metadata around how the supervision tree was initialized.

The `SimpleSupervisor` module we defined here is fairly basic and should look similar to the supervision trees defined in the previous chapters. The only thing that is slightly different this time around is that we create a map of metadata related to the

starting of the supervision tree and then pass it to the `InitTelemetryTask` child process. We can add our `InitTelemetryTask` module to the child process list, since the use Task macro automatically generates a `child_spec/1` function.

With that in place, we can start up the supervision tree and see what our output looks like. You'll notice that the various modules that we defined have print statements. This will help you see the order of how/when things are started.

Listing 100. Starting the `SimpleSupervisor`

```
iex(1) > :telemetry.attach( ①
...(1) >   :init_handler,
...(1) >   InitTelemetryTask.init_event(),
...(1) >   &InitTelemetryTask.simple_handler/4,
...(1) >   nil
...(1) > )
:ok

iex (2) > SimpleSupervisor.start_link([]) ②
Starting SimpleSupervisor
Starting GenServer: gen_server_one
Starting GenServer: gen_server_two
Starting GenServer: gen_server_three
Running Telemetry Task
{:ok, #PID<0.834.0>}
Measurements: %{system_time: 1704236516451483038}
Metadata: %{pid: #PID<0.834.0>, init_args: []}
```

① Attach a telemetry handler to the init event. Note that we are using the `InitTelemetryTask.init_event/0` function to get the signature of the event as opposed to typing it out.

② Start the `SimpleSupervisor` supervision tree to start the three GenServer processes and the single Task process.

You may have expected the `start_link/1` result of `{:ok, #PID<0.834.0>}` to be returned

after to the printing of the Telemetry event data, given that the Telemetry events are dispatched synchronously. The reason that the supervisor PID is returned prior to the telemetry handler being invoked is that we are using the Task module to execute the Telemetry event, and whenever we run Tasks this way, they are executed asynchronously.

Another subtle thing that should be pointed out with this async init Task pattern is the order in which the Task module appeared in the child process list. In this particular case, we want to emit the Telemetry event only if all the other processes in the supervision tree successfully started. If any of the child processes in the supervision tree failed to start, then the Task process (which is last in the list) will not be started. This means that you need to carefully consider where your Task process is slotted into your supervision tree so that you do not falsely report things that may not be true if subsequent processes fail to start and block the supervision tree from starting.

To be clear, if the Task you are running does not have this type of dependency or constraint on the other processes in the supervision tree, you can run it at any point and execute asynchronously as the other processes are starting up. Like many things in software engineering, your specific structure will depend on your specific problem. Luckily, Elixir and OTP are flexible enough to accommodate these different use cases.

6.3.1. Visualizing the Supervision Tree Startup Process

In order to visualize how this supervision tree is started and when the various processes under the supervisor perform their work, it is best to visualize the messages between the processes using a sequence diagram.

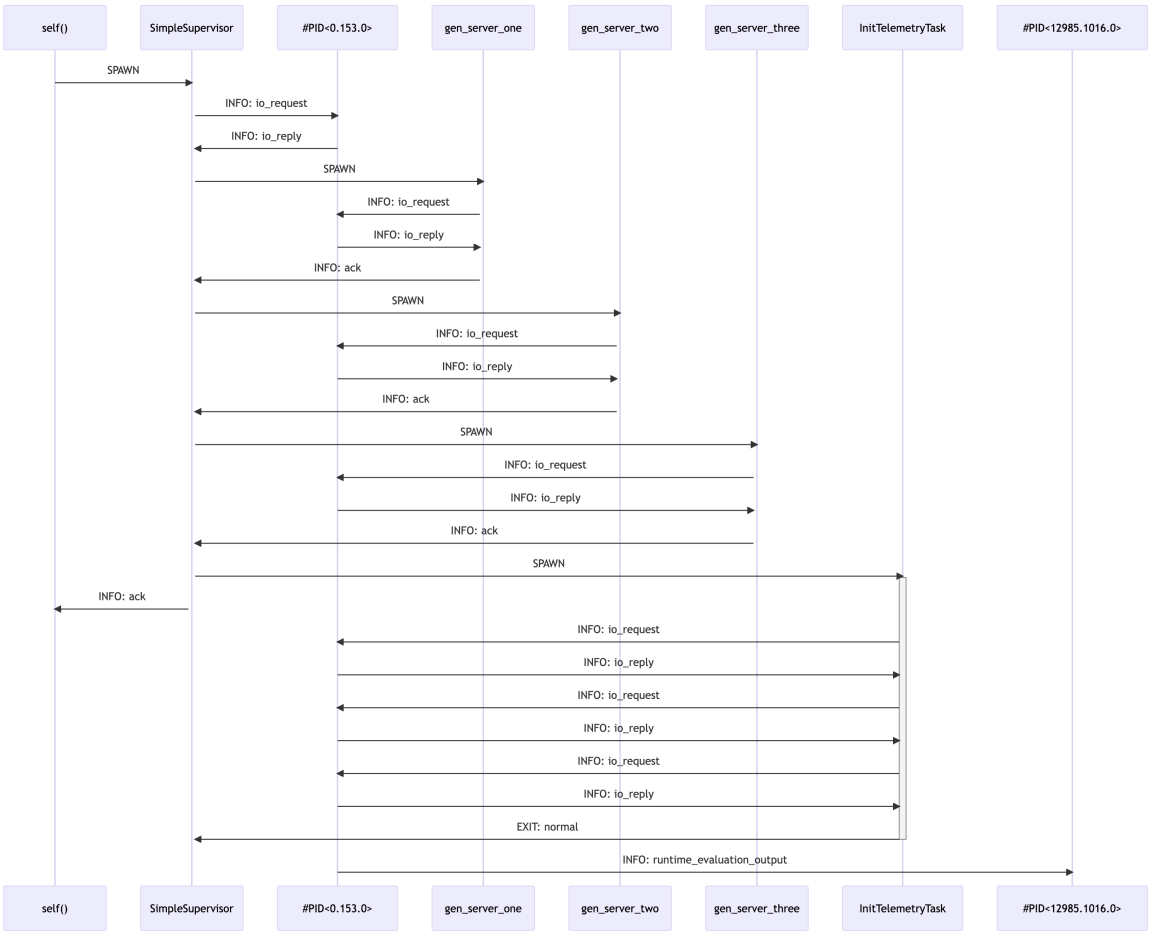


Figure 8. Sequence diagram from starting `SimpleSupervisor`

As you can see, the three GenServer processes (`gen_server_one`, `gen_server_two`, and `gen_server_three`) are started, and then the `InitTelemetryTask` module is spawned before the `SimpleSupervisor` returns an `ack` to the process that started the `SimpleSupervisor` supervisor. After the `InitTelemetryTask` process is started, it then proceeds to output three things (as seen by three `io_request`/`io_reply` pairs). After that, the `InitTelemetryTask` process terminates all the while the `SimpleSupervisor` continues running.

As you can see, with the Task initialization process pattern it is pretty easy to execute some amount of work out of band with the rest of the supervision tree initialization