Eco GenStage

Exercises

Setup

In this section of the exercises, the working directory is `eco` folder within the `beam2024` directory.

The project was generated as an umbrella application. Feel free to explore folder structure and file contents. The command used for project generation was:

```
$ mix phx.new eco --no-mailer --no-ecto --umbrella --no-gettext
```

A key point to note is that this project was created as a Phoenix application rather than a pure Elixir application. The command accepts the same parameters as `mix new`, along with additional options for managing components such as Gettext for internationalization or mailer functionality.

To ensure you have Hex and Phoenix, execute the following commands:

```
$ mix local.hex
$ mix archive.install hex phx_new
```

Next, set up the application (using a predefined alias in the `mix.exs` file):

```
$ mix setup
```

To verify that everything is running smoothly run the below command and navigate to `localhost:4000` in the browser:

```
$ iex -S mix phx.server
```

Goals

The application revolves around submitting sample data to process it with GenStage and finally assigning a star to a respective bin depending on the string contents.

Data processing pipeline

Inputs The application expects data to be submitted via a form, located in the template in the `index_live.ex`. The form contains only one field and a submit button. The data is accepted as a string of any length and is case-insensitive.

Outputs The pipeline should publish an event on the `recyclables` topic with a `{:recycled, sign}` payload using Phoenix.PubSub.

Producing events - Producer / Source In the `apps/eco/lib/eco/` folder, create a `source.ex` file. This will be the GenStage producer. The general structure should be something like this:

```elixir
defmodule Eco.Source do
  use GenStage

  def start_link(initial_state), do GenStage.start_link(__MODULE__, initial_state, name: __M

  def init(initial_state) do
    # your code here
  end

  def handle_demand(demand, state) do
    # your code here
  end
end
```

To correctly initialize the producer, return `{:producer, initial_state, dispatcher: GenStage.BroadcastDispatcher}` from the `init/1` function.

Since we want to leverage the abilities of GenStage to dispatch events automatically as they come in, we don't need a complex `handle_demand/2` implementation. Set the return value to `{:noreply, [], state}`.

To dispatch events down the pipeline, using the form input from the user, we will add `handle_call/3`. Add the following functions:

```elixir
def start(event, timeout \\ 5000) do
  GenStage.call(__MODULE__, {:start, sequence}, timeout)
end

def handle_call({:start, sequence}, _from, state) do
  events = [],
  {:reply, :ok, events, state}
end
```

As mentioned in the Inputs section, our data at this point should be the submitted string. To simulate batch processing, split the string into a list of single letters.

For example, if the input is "ACTGMGPBNHDACT", the output value for the events variable should be `["A", "C", "T", ...]`. Note that at this stage, we are only generating events; no additional data processing is taking place here.

Event data processing In the `apps/eco/lib/eco/` directory, create a `producer_consumer.ex` file. This will correspond to the GenStage `producer_consumer`, a stage where our data will undergo transformation. Here, the data will be converted to upper case and some values will be discarded. The general structure should be something like this:

```elixir
defmodule Eco.ProducerConsumer do
  use GenStage
```

```
  def start_link(_initial) do
    GenStage.start_link(__MODULE__, [], name: __MODULE__)
  end

  @impl true
  def init(init_state), do: {:producer_consumer, init_state, subscribe_to: [Eco.Source]}

  @impl true
  def handle_events(events, _from, state) do
    processed_events = []
    {:noreply, processed_events, state}
  end
end
```

The `start_link/1` is implemented to allow us to incorporate the `ProducerConsumer` module into the supervision tree.

The `init/1` function marks the process as a `:producer_consumer` and subscribes it to the `Eco.Source` module.

In `handle_events/3`, implement the event processing logic - these events will be published to the consumer. Since the event content is case-insensitive, we can freely change its case to facilitate comparison with exemplary data. Apply `String.upcase/1` to the event content and discard any events whose content is not included in the list `["A", "C", "T", "G"]`.

Consume events In the `apps/eco/lib/eco/` directory, create a `sink.ex` file. This module will consume events produced by the previous stages. Here's the general module implementation:

```
defmodule Eco.Sink do
  use GenStage
  alias Phoenix.PubSub

  def start_link(_initial) do
    GenStage.start_link(__MODULE__, [])
  end

  def init(state) do
    {:consumer, state, subscribe_to: [Eco.ProducerConsumer]}
  end

  def handle_events(events, _from, state) do
    # your coude here
    {:noreply, [], state}
  end
```

```
end
```

The `start_link/1` is implemented to allow us to incorporate the `Sink` module into the supervision tree.

The `init/1` function marks the process as a `:consumer` and subscribes it to the `Eco.ProducerConsumer` module.

In `handle_events/3`, implement the logic to consume the list of events. At this stage, iterate over the `events` variable (similar to what we did in the `ProducerConsumer` module) and publish messages on a specific topic using `PubSub`.

To publish messages, use `Phoenix.PubSub.broadcast/3`. The topic to publish information to is `"recyclables"` and the message format should be `{:recycled, sign}`, where `sign` is a value from the event.

Tying it all up Before our pipeline can accept the request, we need to include it in our application's children. To do so, modify the list of children in the `Eco.Application` module by adding the following:

```
children = [
  # ...
  {Eco.Source, []},
  {Eco.ProducerConsumer, []},
  {Eco.Sink, []}
  # ...
]
```

Lastly, in `index_live.ex`, modify the form submit event (remember to add an `alias` to `Source`):

```
def handle_event("save", %{"sequence" => sequence}, socket) do
    Source.start(sequence)
    {:noreply, socket}
  end
```

Additional Start multiple `Sink` instances, modify `handle_event/3` to sleep for a random time between 0.5s and 2s with `Process.sleep(Enum.random(500..2000))`, and observe what happens.