Phoenix and RabbitMQ

Feb 20, 2017

Last post I talked about building a high throughput, highly available analytics backend. I'm not going to do a thorough tutorial here, just identify some steps in getting set up and a few things I learned.

Getting set up

I installed RabbitMQ through `brew`, and Phoenix / Elixir from the Phoenix homepage. Starting the RabbitMQ server was as simple as

```
/usr/local/sbin/rabbitmq
```

and getting a Phoenix app going was easy too (I called my app `frequency`; it's an Artsy tradition to name applications after physics terms):

```
mix phoenix.new frequency --no-brunch --no-ecto --no-html
```

We just want a pure phoenix API, so we'll leave out `brunch` (the build tool), `ecto` (the ActiveRecord equivalent), and all of the `html` support and templating we would need if we wanted our app to have a front-end.

The `mix.exs` file defines the application's dependencies as follows (also add these to your `applications` method):

```
  defp deps do
    [{:phoenix, "~> 1.2.1"},
     {:phoenix_pubsub, "~> 1.0"},
     {:gettext, "~> 0.11"},
     {:cowboy, "~> 1.0"},
     {:amqp, "~> 0.2.0-pre.1"}, # https://github.com/pma/amqp/issues/28
     {:briefly, "~> 0.3"},
     {:ex_aws, "~> 1.0"},
     {:hackney, "~> 1.6"}
    ]
  end
```

Receiving calls and publishing messages

Under `web/router.ex` we add a single `POST` route:

```
  scope "/api", Frequency do
    pipe_through :api

    post "/t", TracksController, :index
  end
```

In that route we reference the `TracksController` which doesn’t exist yet, so under `web/controllers` let’s create `tracks_controller.ex` with the following body:

```
defmodule Frequency.TracksController do
  use Frequency.Web, :controller

  def index(conn, params) do
    {:ok, message} = Poison.encode(params)
    Frequency.Worker.publish(message)
    conn
     |> text("200")
  end
end
```

And you’ll see that in turn defers to a `Frequency.Worker` that we’ll have to make. In `lib` we’ll make `worker.ex` which looks like

```
defmodule Frequency.Worker do
  use GenServer

  ## Client API

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

  def publish(message) do
```

```
    IO.puts "handling cast.. "
    GenServer.cast(:publisher, {:publish, message})
  end

  ## Server Callbacks

  def init(:ok) do
    {:ok, connection} = AMQP.Connection.open
    {:ok, channel} = AMQP.Channel.open(connection)
    AMQP.Queue.declare(channel, "tracks")
    {:ok, %{channel: channel, connection: connection} }
  end

  def handle_cast({:publish, message}, state) do
    AMQP.Basic.publish(state.channel, "", "tracks", message)
    {:noreply, state}
  end

  def terminate(_reason, state) do
    AMQP.Connection.close(state.connection)
  end
end
```

This worker publishes all messages to a RabbitMQ channel: It defines a single GenServer with the name `publisher` which we'll set up to start under the same supervisor as our Frequency application (we'll do this in a minute). The GenServer exposes a single method, `:publish`, which drops the message into a channel defined by the `:init` method. Finally, in `lib/frequency.ex`, update the children of our process to include our new worker.

```
children = [
      # Start the endpoint when the application starts
      supervisor(Frequency.Endpoint, []),
      worker(Frequency.Worker, []),
    ]
```

Halfway there.

Receiving messages from RabbitMQ and posting them to S3

Under `lib`, we'll create a `receiver.ex` which reads messages off the RabbitMQ channel, adds them to a list, and then every 1,000 messages will encode those messages as a JSON file and upload them to S3 using ExAWS (you'll need to add the variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to your environment).

```
alias ExAws.S3

defmodule Receiver do
  def wait_for_messages do
    channel_name = "tracks"
    {:ok, connection} = AMQP.Connection.open
    {:ok, channel} = AMQP.Channel.open(connection)
    AMQP.Queue.declare(channel, channel_name)
    AMQP.Basic.consume(channel, channel_name, nil, no_ack: true)
    Agent.start_link(fn -> [] end, name: :batcher)
    _wait_for_messages()
  end

  defp push(value) do
    Agent.update(:batcher, fn list -> [value|list] end)
    flush_if_full()
  end

  defp flush do
    Agent.update(:batcher, fn _ -> [] end)
  end

  defp full? do
    Agent.get(:batcher, fn list -> length(list) > 1000 end)
  end

  defp make_key do
    rand = :crypto.strong_rand_bytes(6) |> Base.url_encode64
    now = DateTime.utc_now |> DateTime.to_string
    "batch_#{now}_#{rand}.json"
  end

  defp write_and_upload(path, json) do
    File.write!(path, json)
    S3.put_object("<your-bucket>", "frequency/#{make_key()}", File.read!(path)) |
  end

  defp flush_if_full do
    if full?() do
```

```
      l = Agent.get(:batcher, fn list -> list end)
      {:ok, path} = Briefly.create
      {:ok, json} = Poison.encode(l)
      write_and_upload(path, json)
      flush()
    end
  end

  defp _wait_for_messages do
    receive do
      {:basic_deliver, payload, _meta} ->
        push(payload)
        IO.puts "received a message!"
        _wait_for_messages()
    end
  end
end
```

Finally, we can string it all together with `mix phoenix.server` in one terminal window, and `iex -S mix` in another, and in the `iex` pane run

```
Receiver.wait_for_messages
```

And all that's left is hammering our API with `POST` requests, which I elected to do in Ruby:

```
require 'net/http'
uri = URI('127.0.0.1:4000/api/t')
30.times do
  1000.times do
    Thread.new {Net::HTTP.post_form(uri, 'event' => 'sent_a_message', 'user_id' =
  end
  sleep(.5) # ruby can only spawn so many threads
end
```

Sit back and watch your API soak up thousands of concurrent requests without a sweat.

github
twitter