Understanding GenServers in Elixir.

Why does this article exist?

A common mistake when first entering Elixir is to use GenServer's for everything. Wrap most of your code with potential for failure in a GenServer because if it does fail, it's isolated. However, GenServer's have the potential of being extremely dangerous to your application and can also create bottlenecks when used incorrectly.

One of GenServer's most important principals is that it can only process one message at a time; which means that in order to handle multiple messages happening in parallel, you need to have multiple instances/processes of that GenServer running.

Elixir's parallelism is achieved through processes

What are the costs of GenServer's?

Let's say you're creating a mailer (using swoosh) and you wrap it in a GenServer because if it fails you don't want it to crash the server. (This is actual code I've seen in a production app)

Elixir

```elixir
defmodule MyApp.Mailer do
  use Swoosh.Mailer, otp_app: :sample
end

defmodule MyApp.MailSender do
  use GenServer

  import Swoosh.Email

  @from_info {"Bill Nye", "BillNye@TheScienceGuy.org"}

  def start_link(_), do: :ok
  def init(_), do: :ok

  def send_mail(email, name, subject, body) do
    GenServer.call(:mailer, {:send_email, {email, name, subject, body}})
  end

  def send_mail_async(email, name, subject, body) do
    GenServer.cast(:mailer, {:send_email, {email, name, subject, body}})
  end

  def handle_call({:send_mail, {email, name, subect, body}}, _, _) do
     email
       |> create_mail(name, subject, body)
       |> Mailer.deliver
  end

  def handle_call({:send_mail, {email, name, subect, body}}, _, _) do
     email
        |> create_mail(name, subject, body)
        |> Mailer.deliver
  end

  defp create_mail(email, name, subject, body) do
    new()
      |> to({name, email})
      |> from(@from_info)
      |> subject(subject)
      |> html_body(body)
  end
end
```

Here we've created a GenServer with two functions, they both send mail, but one will return the results of the

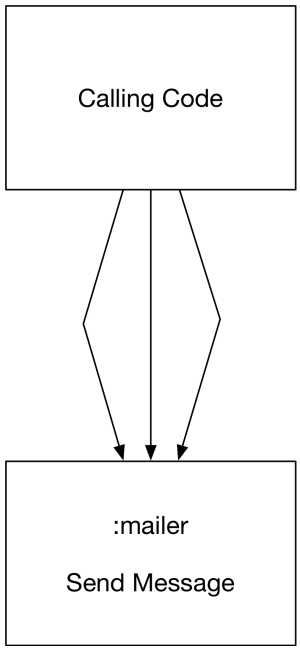
send request while the other will return right away and execute the request shortly after. This module allows mail sends to fail and not crash the rest of your system!

What do you lose by following this?

By doing this one of the (maybe) unintended side-effects is you are limited to only being able to pass one message at a time, this can be leveraged at as a back-pressure system to some extent, but there are much better solutions like GenStage. The reason for this is due to timeouts, when things start queing up and taking longer than there timeout, it's going to drop the messages and they will just be lost!

On the flip side if passing one message at a time is an unintended side effect, for example in the case that you want the sytem to send multiple messages in parallel you've created a bottle neck.

Example of bottleneck:



What can be harmful about this?

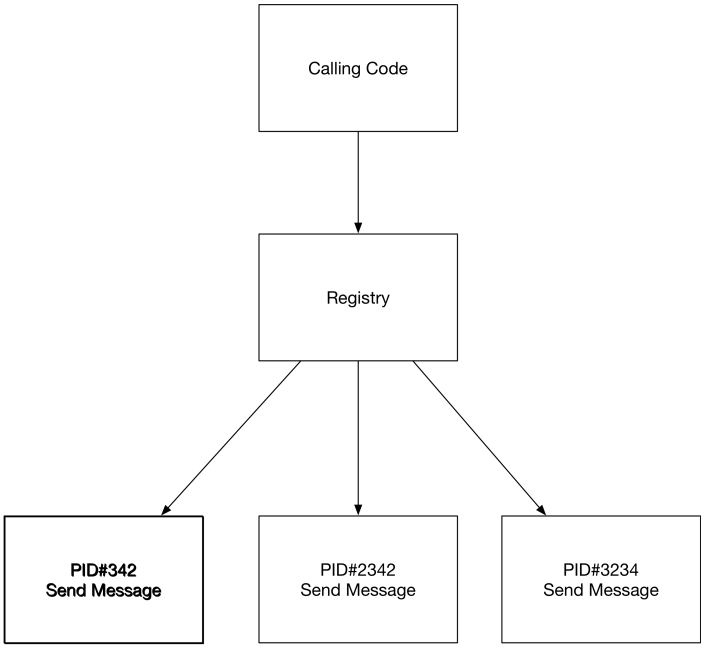
First off it's dangerous to loose messages in the queue due to timeouts. That's a sign your application isn't able to scale to demand, why start with bottlenecks when you can eliminate the obvious ones? That leads to my second point, they can keep a lot in memory, memory isn't free, be careful with what you store in the state. Also when it comes to distribution how will the state work? If you have more nodes does the state need to be shared? These are all questions I'll answer in my next article: *Distributed GenServers*.

Another note about GenServers, is message passing isn't cheap, and things that are large need to be passed a different way otherwise they will get copied in memory, doing this multiple times can lead to your system taking

up more memory than expected or neccessary to pass along messages. Discord created fastglobal as a solution to this problem.

Parallelism: Registries to the rescue

Registries provide you with a way to store set of PID's for GenServers. This allows you to ask the registry to lookup the PID and then call the PID directly.



Registries provide a great solution to possible bottlenecks with GenServer's but this all assumes you're running on one machine, what happens when you start to go distributed!

Some of the solutions on that are:

- horde
- gproc
- pg2

Next time in *Distributed GenServers* we'll discuss how to use some of these tools to create a distributed registry, as well as some of the pains and issues around GenServer's at a distributed level.

Have you fallen trap to the GenServer bottleneck pattern? Or have more questions about GenServers?

Let me know in the comments below!