Registry

Topics

- Old / naive approach
  - Illustrating the issues
- Possible manual solution
- The actual decent solution
- Global approach (simplified)
- Conclusion

- *Note: we'll see global registry after distributed Elixir*

Sample use case

- Chat room
- One general chat room
  - Can be name registered
- Can be done with name: :some_atom
  - *Note: "__MODULE__" compiles to a module name*
  - *A module name is an atom...*

Naive approach – code part 1/3

```elixir
defmodule Demo3.Part1.GeneralChatRoom do
  use GenServer
  @me __MODULE__

  defstruct messages: [], participants: %{}

  # API

  def start_link(args \\ []), do: GenServer.start_link(@me, args, name: @me)
  def send_msg(msg), do: GenServer.cast(@me, {:send_msg, self(), msg})
  def participate(name), do: GenServer.cast(@me, {:participate, name, self()})
```

Naive approach – code part 2/3

```elixir
# CALLBACKS

@impl true
def init(_args), do: {:ok, %@me{}}

@impl true
def handle_call({:participate, _, pid}, _, %@me{} = state)
    when is_map_key(state.participants, pid) do
  {:reply, {:error, :already_participating}, state}
end

@impl true
def handle_call({:participate, name, pid}, _, %@me{} = state) do
  new_state = %{state | participants: Map.put_new(state.participants, pid, name)}
  {:reply, :ok, new_state}
end
```

Naive approach – code part 3/3

```
  @impl true
  def handle_call({:send_msg, sender_pid, msg}, %@me{} = state)
      when not is_map_key(state.participants, sender_pid) do
    {:reply, {:error, :not_a_participating_member}, state}
  end

  @impl true
  def handle_call({:send_msg, sender_pid, msg}, %@me{} = state) do
    sender_readable_name = Map.fetch!(state.participants, sender_pid)
    timestamp = DateTime.utc_now()
    message_entry = {timestamp, sender_readable_name, msg}
    # You could send a message to all participants that a new message has been sent
    new_state = %{state | messages: [message_entry | state.messages]}
  end
end
```

Issues

What are the issues with the previous code?

Issues

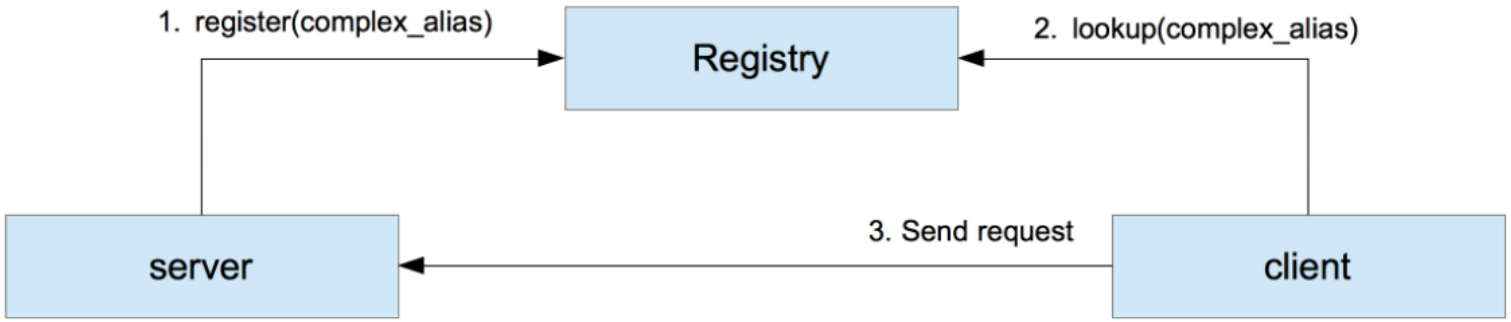
- Not fault tolerant
- Manual PID management
- Only one chat room is possible
  - Due to default name registration limitations
- PubSub (Publish / Subscribe) logic is a manual implementation

Possible manual solution

- Generate atoms when name registering
  - *NOTE: very bad idea*
- Periodic check whether PID is alive
  - *NOTE: very bad patch solution*
- Monitor your processes
- Manual PID management
- For PubSub, work with tasks

- More administrative logic than business logic

Actual solution: registry

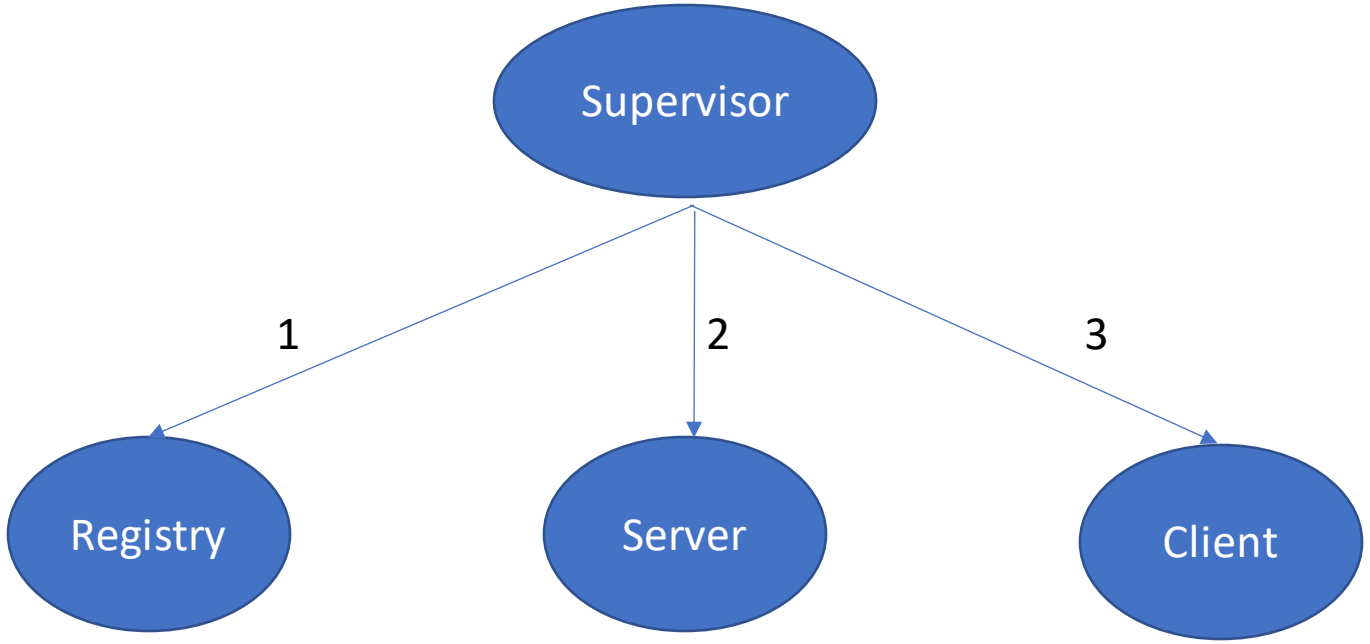


Note: supervision tree

- Look at the dependencies in the previous slide
  - How should you sort the children under your supervisor?

Note: supervision tree

- Look at the dependencies in the previous slide
  - How should you sort the children under your supervisor?



Conclusion

- Registry does the administrative part
- Your GenServer can focus on the business logic
  - And has to do less
- Check the docs!
  - https://hexdocs.pm/elixir/Registry.html