Finding Processes in Elixir

Peter C. Marks - 17 May 2016
@PeterCMarks
github.com/pcmarks

Overview

- Finding Processes the Elixir way
- Finding Processes the Jini way

Processes

- Elixir and Erlang are all about processes.

Processes

- Elixir and Erlang are all about processes.
- The more the merrier!

Processes

- Elixir and Erlang are all about processes.
- The more the merrier!
- This is Good Thing, but ...

Processes

- Elixir and Erlang are all about processes.
- The more the merrier!
- This is Good Thing, but …
- How does one keep track of them?

Processes

- Elixir and Erlang are all about processes.
- The more the merrier!
- This is Good Thing, but …
- How does one keep track of them? Maybe millions!!

Finding Processes

Let me count the ways…

Finding Processes

Let me count the ways…

1. By PID

Finding Processes

Let me count the ways…

1. By PID
2. By name

Finding Processes

Let me count the ways…

1. By PID
2. By name
3. By tuple

Finding Processes

Let me count the ways…

1. By PID
2. By name
3. By tuple
4. Using gproc library

Finding Processes

Let me count the ways…

1. By PID
2. By name
3. By tuple
4. Using gproc library
5. “Resource” discovery

Finding Processes - Example

An Example

- From alphasights.com blog:
  “Process registry in Elixir: A practical example”
- A Chat Server - initially just add and retrieve messages.
- Also: ElixirConfEU talk by Sasa Juric on 12 May

```
defmodule Chat.Server do
 use GenServer
 # API
 def start_link do
   GenServer.start_link(__MODULE__, [])
 end
 def add_message(pid, message) do
   GenServer.cast(pid, {:add_message, message})
 end
 def get_messages(pid) do
   GenServer.call(pid, :get_messages)
 end
 # SERVER
 def init(messages) do
   {:ok, messages}
 end
 def handle_cast({:add_message, new_message}, messages) do
   {:noreply, [new_message | messages]}
 end
 def handle_call(:get_messages, _from, messages) do
   {:reply, messages, messages}
 end
end
```

Finding Processes - PID

By PID -

```
iex> {:ok, pid) = Chat.Server.start_link

    ...

iex> Chat.Server.add_message(pid, “foo”)
iex> Chat.Server.get_messages(pid)
 [“foo”]
```

Finding Processes - Name

By name (atom) -

```
        ...
def start_link do
    GenServer.start_link(__MODULE__, [], name: :chat_room)
end
        ...
```

Finding Processes - Name

By name (atom) -

```
        ...
def start_link do
    GenServer.start_link(__MODULE__, [], name: :chat_room)
end
        ...
def add_message(message) do
   GenServer.cast(:chat_room, {:add_message: message})
end
        ...
```

Finding Processes - Name

By name (atom) -

```
        ...
def start_link do
    GenServer.start_link(__MODULE__, [], name: :chat_room)
end
        ...
def add_message(message) do
   GenServer.cast(:chat_room, {:add_message: message})
end
        ...
iex> Chat.Server.start_link
iex> Chat.Server.add_message(“foo”)
```

Finding Processes - Name

By name (atom) -

- But, ...

Finding Processes - Name

By name (atom) -

- But, suppose we wanted to support multiple chat rooms?

Finding Processes - Name

By name (atom) -

- But, suppose we wanted to support multiple chat rooms?
- We could use :chatroom1, :chatroom2, etc.

Finding Processes - Name

By name (atom) -

- But, suppose we wanted to support multiple chat rooms?
- We could use :chatroom1, :chatroom2, etc.
- Or, :joesroom but what if there are two Joe’s who want to use their name?

Finding Processes - Name

By name -

- But, suppose we wanted to support multiple chat rooms?
- We could use :chatroom1, :chatroom2, etc.
- Or, :joesroom but what if there are two Joe's who want to use their name?

Servers (GenServer) allow names to be one of three types: term (atom), and ...

Finding Processes - Tuples

By two forms of a tuple -

1. {:global, term} - across multiple nodes, uses ETS table, synchronized
2. {:via, module, term} - implement yourself

e.g.,

```
def start_link do
    GenServer.start_link(__MODULE__, [],  name: {:global, :chat_room} )
end
```

Finding Processes - Tuples

By two forms of a tuple -

1. {:global, term} - across multiple nodes, uses ETS table, synchronized
2. {:via, module, term} - implement yourself

e.g.,

```
def start_link do
    GenServer.start_link(__MODULE__, [],  name: {:global, :chat_room} )
end
```

equivalent to:

```
    GenServer.start_link(__MODULE__, [],  name: {via, :global, :chat_room} )
```

Finding Processes - Tuples

If you do choose to implement your own registration mechanism, e.g. {:via, module, term} then ...

Finding Processes - Tuples

If you do choose to implement your own registration mechanism, e.g. {:via, module, term} then …

module must implement:

- register_name/2
- unregister_name/1
- whereis_name/1
- send/2

Finding Processes - gproc

The gproc library -

- See Sasa Juric's ElixirConf.eu talk
- gproc is an erlang library

Finding Processes - gproc

The gproc library -

- See Sasa Juric's ElixirConf.eu 2016 talk
- gproc is an erlang library
- It can generate a unique name

Finding Processes - gproc

Using the gproc library -

- See Sasa Juric's ElixirConf.eu 2016 talk
- gproc is an erlang library
- It can generate a unique name
- register the name locally or globally

Finding Processes - gproc

Using the gproc library -

- See Sasa Juric's ElixirConf.eu 2016 talk
- gproc is an erlang library
- It can generate a unique name
- register the name locally or globally
- Associate the process with aliases

```
GenServer.start_link(__MODULE__, [],
    name: {:via, :gproc, {:n, :l, {:session, some_id}}})
```

Finding Processes - Resource

“Resource” discovery -

- Chapter 8 of “Erlang and OTP In Action”

Finding Processes - Resource

“Resource” discovery -

- Chapter 8 of “Erlang and OTP In Action”
- Google “resource discovery in Erlang”

Finding Processes - Resource

“Resource” discovery -

- Chapter 8 of “Erlang and OTP In Action”
- Google “resource discovery in Erlang”
- A little more complicated than gproc

Finding Processes - Resource

“Resource” discovery -

- Chapter 8 of “Erlang and OTP In Action”
- Google “resource discovery in Erlang”
- A little more complicated than gproc
    - Register processes for discovery

Finding Processes - Resource

“Resource” discovery -

- Chapter 8 of “Erlang and OTP In Action”
- Google “resource discovery in Erlang”
- A little more complicated than gproc
  - Register processes for discovery
  - Lookup - “Resource Discovery” - by type, a string

Limited Solutions

These are all good methods/solutions, but they are limited ...

Limited Solutions

These are all good methods/solutions, but they are limited …

For example:

- Processes are tied to some sort of naming convention: arbitrary

Limited Solutions

These are all good methods/solutions, but they are limited …

For example:

- Processes are tied to some sort of naming convention: arbitrary
- Does a “PrintService” or :print_service really print?

Limited Solutions

These are all good methods/solutions, but they are limited …

For example:

- Processes are tied to some sort of naming convention: arbitrary
- Does a “PrintService” or :print_service really print?
- Most importantly: What is the service’s API?

Jini

Very briefly.

- Jini was released by Sun Microsystems in 1998
- Now an Apache project: River

Jini

Very briefly.

- Jini was released by Sun Microsystems in 1998
- Now an Apache project: River
- A service-object-oriented architecture

Jini

Very briefly.

- Jini was released by Sun Microsystems in 1998
- Now an Apache project: River
- A service-object-oriented architecture
  - Usually services are written in Java

Jini

Very briefly.

- Jini was released by Sun Microsystems in 1998
- Now an Apache project: River
- A service-object-oriented architecture
  - Usually services are written in Java
  - Can use “proxies”

Jini

Very briefly.

- Jini was released by Sun Microsystems in 1998
- Now an Apache project: River
- A service-object-oriented architecture
  - Usually services are written in Java
  - Can use “proxies”
- One of its unique ideas: Identify and find services by what they do, ***not*** by how they are named

Jini

Very briefly.

- Jini was released by Sun Microsystems in 1998
- Now an Apache project: River
- A service-object-oriented architecture
  - Usually services are written in Java
  - Can use “proxies”
- One of its unique ideas: Identify and find services by what they do, ***not*** by how they are named
- Use a service’s (object’s) classes and/or its implemented interfaces.

Finding Processes: Behaviourly

Elixir has something comparable to Java interfaces

Finding Processes: Behaviourly

Elixir has something comparable to Java interfaces: behaviours

Finding Processes: Behaviourly

Elixir has something comparable to Java interfaces: behaviours

Refactoring our Chat Server to use behaviours ...

```elixir
defmodule Chat.Server do
 use GenServer
 # API
 def start_link do
   GenServer.start_link(__MODULE__, [])
 end
 def add_message(pid, message) do
   GenServer.cast(pid, {:add_message, message})
 end
 def get_messages(pid) do
   GenServer.call(pid, :get_messages)
 end
 # SERVER
 def init(messages) do
   {:ok, messages}
 end
 def handle_cast({:add_message, new_message}, messages) do
   {:noreply, [new_message | messages]}
 end
 def handle_call(:get_messages, _from, messages) do
   {:reply, messages, messages}
 end
end
```

Finding Processes: Behaviourly

Elixir has something comparable to Java interfaces: behaviours
First:

```
defmodule Chat.Behaviour do

  @callback add_message(pid(), String.t) :: none()
  @callback get_messages(pid()) :: [String.t]

end
```

Finding Processes: Behaviourly

Elixir has something comparable to Java interfaces: behaviours

First:

```
defmodule Chat.Behaviour do

  @callback add_message(pid(), String.t) :: none()
  @callback get_messages(pid()) :: [String.t]

end
```

Next:

```
defmodule Chat.Server do
  use GenServer

  @behaviour Chat.Behaviour

      ...
```

Finding Processes: Behaviourly

How it could work:

Finding Processes: Behaviourly

How it could work:
Given a module, one can get info about it using the __info__/1 function:

```
iex > Chat.Server.__info__(:attributes)
[vsn: [143187402902677653999420462058756613063], behaviour: [:gen_server],
 behaviour: [Chat.Behaviour]]
```

Finding Processes: Behaviourly

How it could work:
Given a module, one can get info about it using the __info__/1 function:

```
iex > Chat.Server.__info__(:attributes)
[vsn: [143187402902677653999420462058756613063], behaviour: [:gen_server],
 behaviour: [Chat.Behaviour]]
```

or to get a list of behaviours:

```
iex > for {:behaviour, x} <- Chat.Server.__info__(:attributes), do: hd(x)
[:gen_server, Chat.Behaviour]
```

Finding Processes: Behaviourly

How it could work:
Given a module, one can get info about it using the __info__/1 function:

```
iex > Chat.Server.__info__(:attributes)
[vsn: [143187402902677653999420462058756613063], behaviour: [:gen_server],
 behaviour: [Chat.Behaviour]]
```

or to get a list of behaviours:

```
iex > for {:behaviour, x} <- Chat.Server.__info__(:attributes), do: hd(x)
[:gen_server, Chat.Behaviour]
```

So, if this module implements the Chat.Behaviour then we know its API

Finding Processes: Behaviourly

How do we use this feature?

Finding Processes: Behaviourly

How do we use this feature?

- Extend GenServer to
  - accept a new name type:

Finding Processes: Behaviourly

How do we use this feature?

- Extend GenServer to
  - accept a new name type: `{:via_behaviour, term}`
  - or `{:via_behaviour, module, term}`
- Extend :gproc, similarly

Finding Processes: Behaviourly

How do we use this feature?

- Extend GenServer to
  - accept a new name type: `{:via_behaviour, term}`
  - or `{:via_behaviour, module, term}`
- Extend :gproc, similarly
- Ideas?