# Serving Generally

A primer into **GenServers**

@gogogarrett

# A **GenServer** in the *wild*

```elixir
defmodule Machina.GameAssigner do
  use GenServer

  alias Machina.GameAssigner.{Timeout, GameFactory}

  def start_link(scope), do: GenServer.start_link(__MODULE__, scope, [name: context(scope)])
  def add_player(pid, player_id), do: GenServer.call(pid, {:add_player, player_id})

  def init(context) do
    :timer.send_interval(2_000, :assign_games)

    {:ok, %{context: context, waiting_players: [], had_players: false}}
  end

  def handle_info(:assign_games, state) do
    with {:ok, :waiting} <- Timeout.check(state.waiting_players),
         {:ok, new_players, game} <- GameFactory.build(state)
    do
      StadiumWeb.Endpoint.broadcast("lobby:game_assigner", "game_offer", game)
      {:noreply, Map.update!(state, :waiting_players, fn (_x) -> new_players end)}
    else
      {:error, :no_players} ->
        {:stop, :normal, state}
      {:error, :timeout_expired} ->
        new_players = case Enum.count(state.waiting_players) do
          count when count in [2, 3] ->
            {:ok, new_players, game} = GameFactory.build_with_droids(state)
            StadiumWeb.Endpoint.broadcast("lobby:game_assigner", "game_offer", game)
            new_players
          _ ->
            StadiumWeb.Endpoint.broadcast("lobby:game_assigner", "game_reject", %{})
            []
        end
        {:stop, :normal, Map.update!(state, :waiting_players, fn (_x) -> new_players end)}
      _ ->
        {:noreply, state}
    end
  end

  def handle_call({:add_player, player_id}, _from, state) do
    new_state =
      %{state |
        waiting_players: Enum.uniq(state.waiting_players ++ [%{join_time: current_time, player_id: player_id}]),
        had_players: true
      }

    {:reply, new_state, new_state}
  end

  defp context(scope), do: :"context:#{scope}"
  def current_time, do: DateTime.utc_now() |> DateTime.to_unix()
end
```

Before we get to **GenServers**...

Let's learn some ***Elixir***

*elixir* much **wow**

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

- Scalability
- Fault-tolerance
- Functional programming
- Extensibility and DSLs
- *Does your taxes*
- A growing ecosystem
- Interactive development
- Erlang compatible

# The aim

- What are processes?
- What is message passing?
- What is state?
- Where am I?
- What are GenServers?

# Processes

# Processes

- All elixir code runs inside processes
- Processes are isolated from each other
- Run concurrent to one another and communicate via message passing
- Building blocks for distributed and fault-tolerant systems

# Basics: processes

## A slow function

```
long_process = fn(something_slow) ->
  :timer.sleep(2000)
  "#{something_slow} result"
end

long_process.("Garrett's")
# 2 secs later
# => "Garrett's result"
```

# Basics: processes

## A really slow function

```
1..5
|> Enum.map(&long_process.("Query #{&1}"))
# 10 secs later
# => ["Query 1 result", "Query 2 result", "Query 3 result", "Query 4 result", "Query 5 result"]
```

That sucks
- someone smart

# spawn/1

# `spawn/1`

- takes a zero-arity lambda
- creates a process and returns, ie: **non-blocking**
- the provided lambda is executed in the new progress, ie: **concurrent**
- after lambda is finished, process exists and memory is released

# Basics: processes

## A non-blocking function

```
spawn(fn -> IO.puts(long_process.("Query 1")) end)
# => #PID<0.70.0>
IO.inspect("howdy")
# => "howdy"
# 2 sec later
# => Query 1 result
```

# Basics: processes

The building blocks:

```
async_long_process = fn(something_slow) ->
  spawn(fn -> IO.puts(long_process.(something_slow)) end)
end

async_long_process.("Yo dawg")
# => #PID<0.77.0>
# 2 sec later
# => Yo dawg result
```

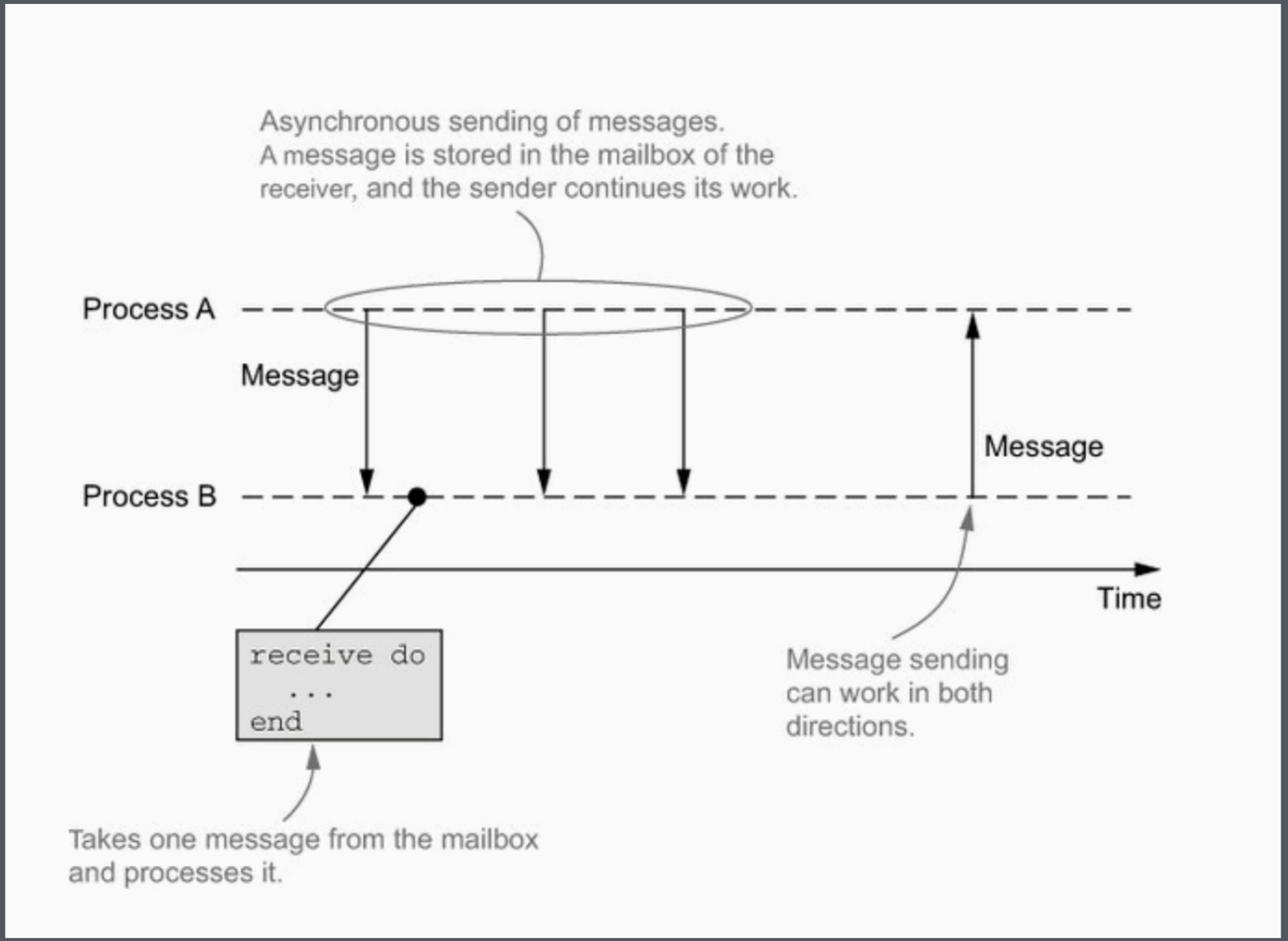
# Basics: processes

## A really slow function, not slow.

```
1..5
|> Enum.map(&async_long_process.("Query #{&1}"))
# => [#PID<0.81.0>, #PID<0.82.0>, #PID<0.83.0>, #PID<0.84.0>, #PID<0.85.0>]
# 2 sec later
# => Query 1 result
# => Query 2 result
# => Query 3 result
# => Query 4 result
# => Query 5 result
```

# Message Passing

# Basics: message passing



send/2

receive/1

# `send/2`

- sends a `message` to the given `dest` and returns the message
- `dest` may be a remote or local pid, a (local) port, a locally registered name, or a tuple {registered_name, node} for a registered name at another node
- `message` can be any elixir data structure
- The consequence of `send` is that a message is placed in the mailbox of the receiver. The caller then continues doing something else.

# `receive/1`

- Take the first message from the mailbox
- Try to match it against the receive patterns, top to bottom
- If a pattern matches the message, run the corresponding code
- ...

# `receive/1`

- If no pattern matches, put the message back into the mailbox at the same position it originally occupied. Then try the next message
- If there are no more messages in the queue, wait for a new one to arrive. When a new message arrives, start from step 1, inspecting the first message in the mailbox
- If the after clause is specified and no message arrives in the given amount of time, run the code from the after block

# Basics: message passing

## Sending a message

```
# sender
pid = self
send(pid, {:anything, :you, :want, 2})
# => {:anything, :you, :want, 2}

# receiver
receive do
  {:anything, :you, :want, number} ->
    IO.inspect("You wanted, #{number}")
  {:greet, person} ->
    IO.inspect(person)
  _ ->
    IO.inspect("un_handled_message")
  after 5000 ->
    IO.inspect("no message received in 5 secs")
end
# => "You wanted, 2"

send(pid, {:greet, "Garrett"})
# => {:greet, "Garrett"}
# What happens here??
# => ?????
```

# Basics: message passing

```
async_long_process = fn(something_slow) ->
  caller = self

  spawn(fn ->
    send(caller, {:long_process, long_process.(something_slow)})
  end)
end

1..5 |> Enum.each(&async_long_process.("query #{&1}"))
# => :ok

get_result = fn ->
  receive do
    {:long_process, result} -> result
  end
end

results = 1..5 |> Enum.map(fn(_) -> get_result.() end)
# => ["query 1 result", "query 2 result", "query 3 result", "query 4 result", "query 5 result"]
```

# Basics: message passing

## parallel map

```
1..5
|> Enum.map(&async_long_process.("query #{&1}"))
|> Enum.map(fn(_) -> get_result.() end)
# => ["query 1 result", "query 2 result", "query 3 result", "query 4 result", "query 5 result"]
```

# State Management

# State Management

- long-running processes that can respond to various messages
- processes can keep their internal state, which other processes can query or even manipulate
- stateful server processes resemble objects
- they maintain state and can interact with other processes via messages
- multiple server processes may run in parallel

# Basics: state management

## The wiring

```
defmodule DatabaseServer do
  def start do # client process
    spawn(&loop/0)
  end

  defp loop do # spawns in a new process
    receive do
      _ ->
        IO.inspect("howdy")
    end

    loop # tail-recursive
  end
end

pid = DatabaseServer.start
# => #PID<0.92.0>
send(pid, :hi)
# => "howdy"
```

# Basics: state management

```
defmodule DatabaseServer do
  def start do
    spawn(&loop/0)
  end

  def run_async(server_pid, something_slow) do # calculates the message and stores in clients mailbox
    # send message to server process
    # pass in "self" to store the message in _our_ mailbox
    send(server_pid, {:run_long_process, self, something_slow})
  end

  defp loop do # listens to messages and responds to them
    receive do
      {:run_long_process, caller, something_slow} ->
        send(caller, {:query_result, long_process(something_slow)})
      _ -> IO.inspect("howdy")
    end

    loop
  end

  defp long_process(something_slow) do # actually performing the work
    :timer.sleep(2000)
    "#{something_slow} result"
  end
end
```

# Basics: state management

```
defmodule DatabaseServer do
  # ...
  def get_result do
    # in the client, receive the messages back from the server
    # pattern match the message and return the `long_process` result
    receive do
      {:query_result, result} ->
        result
      after 5000 ->
        {:error, :timeout}
    end
  end

  defp loop do # listens to messages and responds to them
    receive do
      {:run_long_process, caller, something_slow} ->
        send(caller, {:query_result, long_process(something_slow)})
      _ -> IO.inspect("howdy")
    end

    loop
  end
  # ...
end

server_pid = DatabaseServer.start
# => #PID<0.146.0>
DatabaseServer.run_async(server_pid, "query 1")
# => {:run_long_process, #PID<0.61.0>, "query 1"}
DatabaseServer.get_result
# => "query 1 result"
DatabaseServer.get_result
# 5 sec later
# => {:error, :timeout}
```

# Basics: state management - Adding state

```
defmodule CounterServer do
  def start do
    spawn(fn ->
      initial_state = %{count: 0}
      loop(initial_state)
    end)
  end

  defp loop(state) do
    new_state = receive do
      {:fetch_state, caller} ->
        send(caller, state)
        state
      :add ->
        %{state | count: state.count + 1}
      _ ->
        state
    end

    loop(new_state) # tail-recursive, passing back the state
  end
end

c_pid = CounterServer.start           # => #PID<0.268.0>
send(c_pid, :add)                     # => :add
send(c_pid, :add)                     # => :add
send(c_pid, {:fetch_state, self})     # => {:fetch_state, #PID<0.61.0>}
flush                                 # => %{count: 2}
```

# Basics: state management - Adding client helpers

```elixir
defmodule CounterServer do
  def start do
    spawn(fn -> loop(%{count: 0}) end)
  end

  def add(server_pid) do
    send(server_pid, :add)
  end

  def fetch_state(server_pid) do
    send(server_pid, {:fetch_state, self})
  end

  def get_result do
    receive do
      {:query_result, result} ->
        result
      _ -> {:error, :no_match}
    end
  end

  defp loop(state) do
    new_state = receive do
      {:fetch_state, caller} ->
        send(caller, {:query_result, state})
        state
      :add ->
        %{state | count: state.count + 1}
      _ ->
        state
    end

    loop(new_state)
  end
end

pid = CounterServer.start          # => #PID<0.117.0>
CounterServer.add(pid)             # => :add
CounterServer.add(pid)             # => :add
CounterServer.fetch_state(pid)     # => {:fetch_state, #PID<0.72.0>}
CounterServer.get_result           # => %{count: 2}
```

We've learned *Elixir*.

# GenServers

# Behaviours

Design patterns for processes
- Benjamin Tan Wei Hao

A behaviour is a way to say: give me a module as argument and I will invoke the following callbacks on it, with these argument and so on.
- Jose Valim

A behaviour module for implementing the server of a client-server

# Genserver

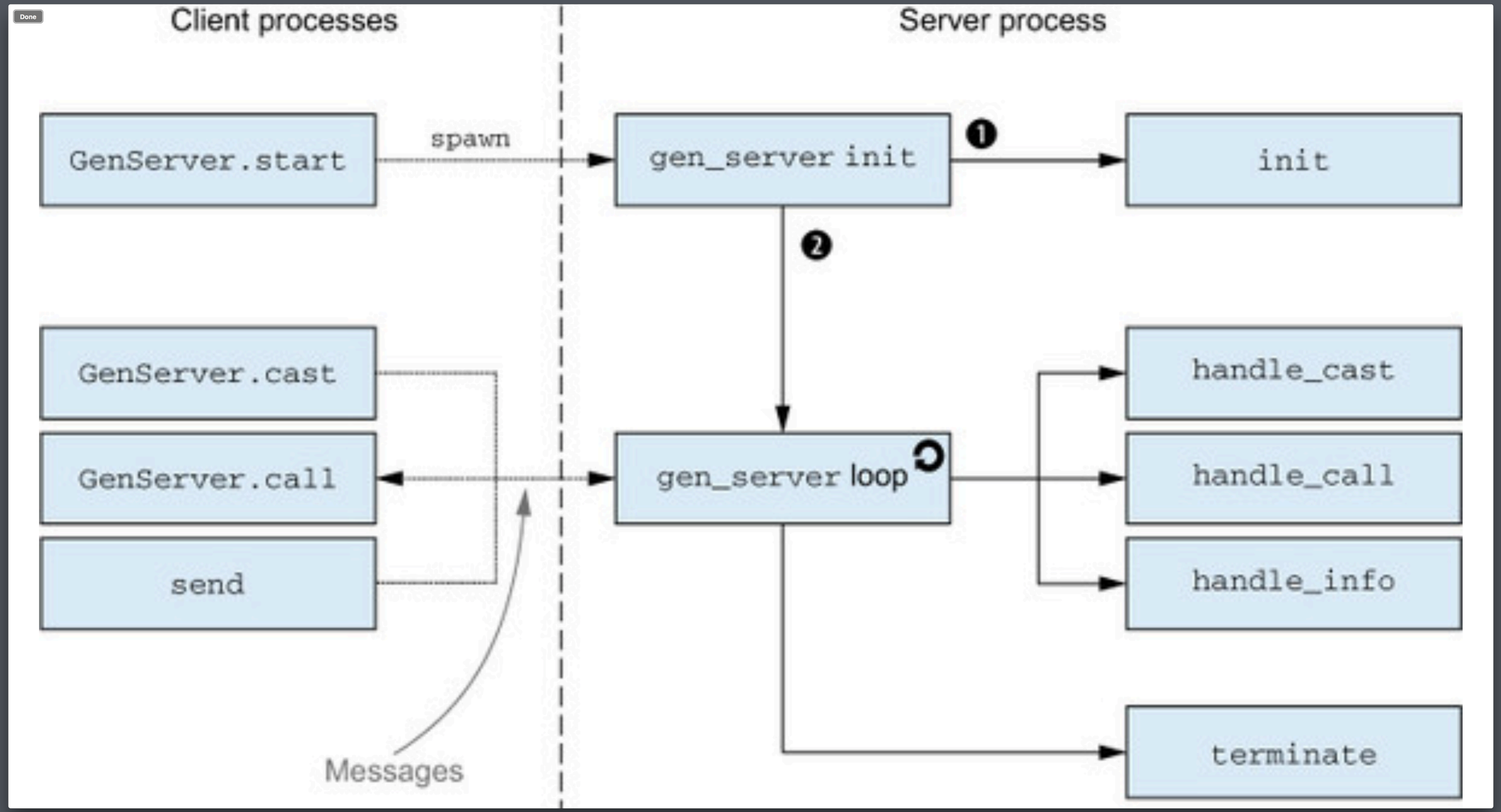
- wraps a regular module in a set of special functions
- that abstract the way processes communicate

## Genserver: functions

- `start` - spawn a new server
- `start_link` - spawn a new server, and links to the current process
- `call` - send a sync message to the server and wait for a reply
- `cast` - send an async message to the server and continue executing
- `reply` - used to reply directly to a client

## Genserver: callbacks

- `init(args)` - setup inital state of process
- `handle_call(request, from, state)` - handle sync messages
- `handle_cast(request, state)` - handle async messages
- `handle_info(msg, state)` - handle general messages
- `terminate(reason, state)` - hook to handle shutdown
- `code_change(old_vsn, state, extra)` - witchcraft to migrate state between otp releases


Done
Client processes
Server process
GenServer.start
spawn
gen_server init
1
init
2
GenServer.cast
GenServer.call
send
gen_server loop
handle_cast
handle_call
handle_info
terminate
Messages

## GenServer

- A GenServer is implemented in *two parts*:
  - the client API and,
  - the server callbacks.
- The client and server run in *separate processes*, with the client passing messages back and forth to the server as its functions are called.

# GenServer: basic setup

```
defmodule Basic do
  use GenServer

  def start_link do
    # GenServer.start_link(Basic, [])
    # __MODULE__ == Basic || this_module
    # [] == initial state
    GenServer.start_link(__MODULE__, [])
  end
end

{:ok, pid} Basic.start_link
# => {:ok, #PID<0.101.0>}
```

# GenServer: basic setup

```
defmodule Basic do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, [])
  end

  def init(initial_data) do
    # initial_data == []
    {:ok, initial_data} # {:ok, some_state}
  end
end
```

# GenServer: basic setup

```
defmodule Basic do
  use GenServer

  def start_link, do: GenServer.start_link(__MODULE__, "hello")

  def get_my_state(pid) do
    GenServer.call(pid, :get_the_state)
  end

  def init(initial_data) do # server api
    greetings = %{greeting: initial_data}
    {:ok, greetings}
  end

  def handle_call(:get_the_state, _from, state) do
    {:reply, state, state}
  end
end

{:ok, pid} = Basic.start_link
Basic.get_my_state(pid)              # => %{greeting: "Hello"}
GenServer.call(pid, :get_the_state) # => %{greeting: "Hello"}
```

# `handle_call`

- is synchronous
- blocking until it returns a reply
- first arg is the pattern to match
- second arg is a reference from the caller
- third arg is the `current_state`

# GenServer: get and set state

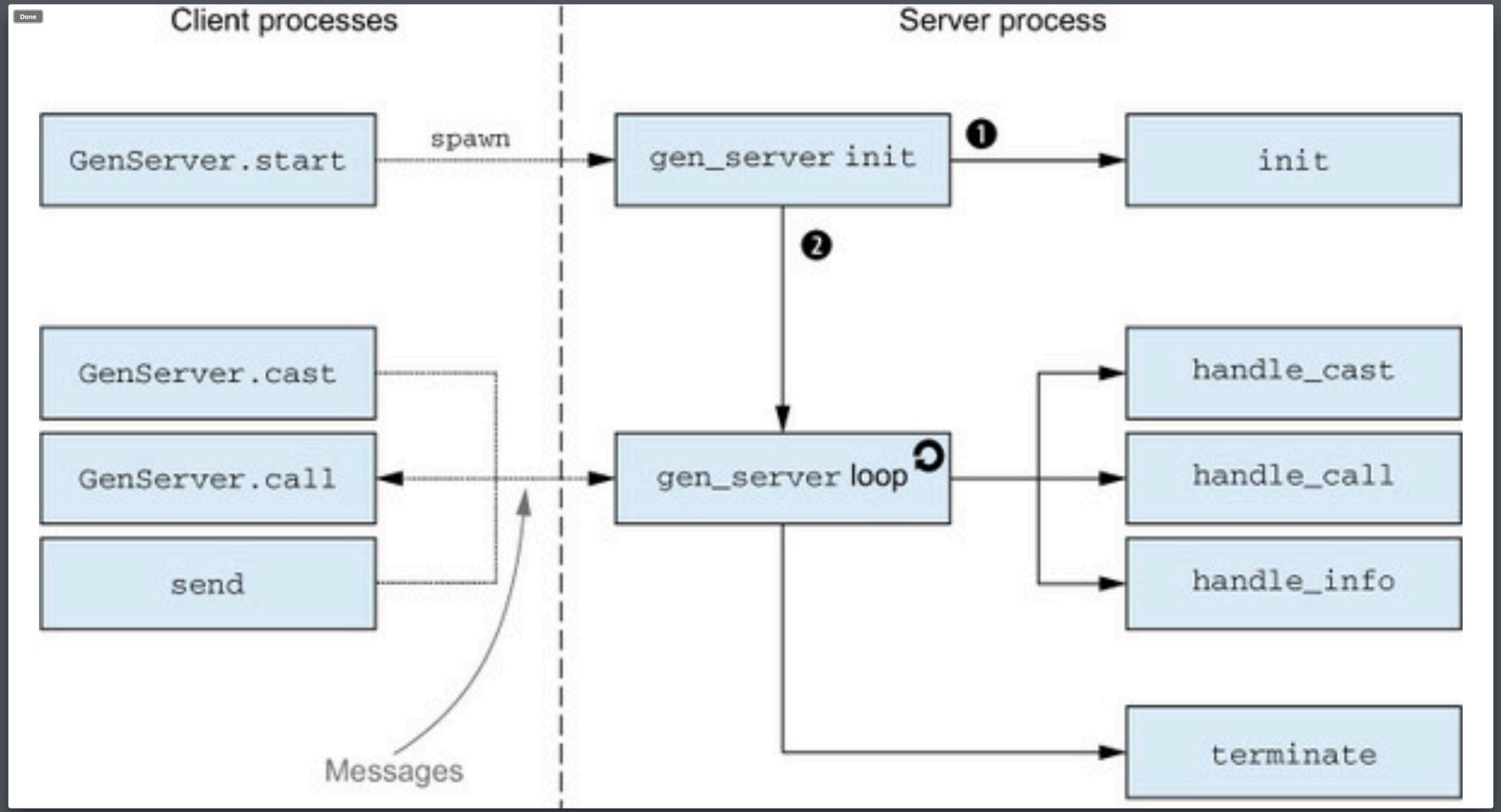
```
defmodule Basic do
  use GenServer

  def start_link, do: GenServer.start_link(__MODULE__, "hello")

  def get_my_greeting(pid), do: GenServer.call(pid, :get_the_greeting)

  def set_my_greeting(pid, new_greeting), do: GenServer.call(pid, {:set_the_greeting, new_greeting})

  def init(initial_data) do # server api
    greetings = %{greeting: initial_data}
    {:ok, greetings}
  end

  def handle_call(:get_the_greeting, _from, state) do
    current_greeting = Map.get(state, :greeting)
    {:reply, current_greeting, state}
  end

  def handle_call({:set_the_greeting, new_greeting}, _from, state) do
    new_state = Map.put(state, :greeting, new_greeting)
    {:reply, new_state, new_state}
  end
end

{:ok, pid} = Basic.start_link
greeting = Basic.get_my_greeting(pid)   # => "hello"
Basic.set_my_greeting(pid, "howdy")     # => %{greeting: "howdy"}
greeting = Basic.get_my_greeting(pid)   # => "howdy"
```

# `handle_cast`

- is asynchronous
- continues executing in client process
- may or may not change server state

# GenServer: get and set (async) state

```elixir
defmodule Basic do
  use GenServer

  def start_link, do: GenServer.start_link(__MODULE__, "hello")

  def init(initial_data) do
    greetings = %{greeting: initial_data}
    {:ok, greetings}
  end

  def get_my_greeting(pid), do: GenServer.call(pid, :get_the_greeting)

  def set_my_greeting(pid, new_greeting), do: GenServer.cast(pid, {:set_the_greeting, new_greeting})

  def handle_call(:get_the_greeting, _from, state) do
    current_greeting = Map.get(state, :greeting)
    {:reply, current_greeting, state}
  end

  def handle_cast({:set_the_greeting, new_greeting}, state) do
    new_state = Map.put(state, :greeting, new_greeting)
    {:noreply, new_state}
  end
end

{:ok, pid} = Basic.start_link
greeting = Basic.get_my_greeting(pid)   # => "hello"
Basic.set_my_greeting(pid, "howdy")     # => %{greeting: "howdy"}
greeting = Basic.get_my_greeting(pid)   # => "howdy"
```


Done
Client processes
Server process
GenServer.start
spawn
gen_server init
1
init
2
GenServer.cast
GenServer.call
send
gen_server loop
handle_cast
handle_call
handle_info
terminate
Messages

# Resources

- Nice 8min vid over GenServers
- Elixir in Action - the best elixir book
- The Little Elixir & OTP Guidebook

{:reply, **"Thanks"**, **"Fin"**}

**@gogogarrett**