The Actor Model, Part Four

CSCI 5828: Foundations of Software Engineering
Lecture 16 — 10/13/2016

Goals

- Introduce OTP
  - GenServer: a module for creating Elixir actors
  - Supervisor: a module for creating actors that manage other actors
- Provide examples throughout

OTP and GenServer (I)

- We've now seen multiple examples of actor-based systems in Elixir
  - In almost every example, our server process
    - had to be spawned (and linked) with an initial state
    - had to implement a "recursive" loop that accepted the current state
    - had to implement message handlers that would update the state as needed
    - had to come up with a way to identify when the server was done, so it could shutdown cleanly
    - had to be able to send messages back to its clients when needed
- None of this code is difficult but it can be tedious and easy to make mistakes

OTP and GenServer (II)

- As a result, a framework called OTP implements a "behavior" called GenServer that can be included in a module to standardize that code
  - Once we include GenServer
    - we get default implementations of six callback methods
    - we override these methods to specify application-specific behavior
- Those methods are:
  - `init(initial_state)`: provides opportunity to initialize server
  - `handle_call(message, {client, tag}, current_state)`: handle a message that needs a reply; typically return { :reply, response, new_state}
  - `handle_cast(message, current_state)`: handle a message that does not need a reply; typically return {:noreply, new_state}
  - `handle_info`; `terminate`; `code_change`; `format_status`: **see textbook**

OTP and GenServer (III)

- There are a two shared responses for `handle_call` and `handle_cast`
  - `{:noreply, new_state}`: update the state without replying to the client
  - `{:stop, reason, new_state}`: signal the server should stop
- There are two additional responses for `handle_call`:
  - `{:reply, response, new_state}`: update state and reply to the client
  - `{:stop, reason, reply, new_state}`: return a reply and then signal that the server should stop
- The `:reply` and `:noreply` responses can be augmented with options:
  - `:hibernate`: Tell the server to store state to disk and go to "sleep" until next message arrives
  - `timeout`: Tell the server to send itself a timeout message if it doesn't receive a message in the specified number of milliseconds

Simple Example (I)

```
defmodule Sequence.Server do
  use GenServer

  def handle_call(:next_number, _from, current_number) do
    { :reply, current_number, current_number+1 }
  end

  def handle_cast({:increment_number, delta}, current_number) do
    { :noreply, current_number + delta}
  end
end
```

Simple Example (II)

```
iex> { :ok, pid } = GenServer.start_link(Sequence.Server, 100)
{:ok,#PID<0.60.0>}
iex> GenServer.call(pid, :next_number)
100
iex> GenServer.call(pid, :next_number)
101
iex> GenServer.cast(pid, {:increment_number, 200})
:ok
iex> GenServer.call(pid, :next_number)
302
```

- Use `GenServer.start_link` to create a new instance of a `GenServer` actor
- Use `GenServer.call` for a blocking call to a `GenServer` actor
- User `GenServer.cast` for a non-blocking call to a `GenServer` actor

Simple Example (III)

```
iex> {:ok,pid} = GenServer.start_link(Sequence.Server, 100, [debug: [:trace]])
{:ok,#PID<0.68.0>}
iex> GenServer.call(pid, :next_number)
*DBG* <0.68.0> got call next_number from <0.25.0>
*DBG* <0.68.0> sent 100 to <0.25.0>, new state 101
100
iex> GenServer.call(pid, :next_number)
*DBG* <0.68.0> got call next_number from <0.25.0>
*DBG* <0.68.0> sent 101 to <0.25.0>, new state 102
101
```

- Pass `[debug: [:trace]]` to generate tracing information for all calls

Simple Example (IV)

```
iex> {:ok,pid} = GenServer.start_link(Sequence.Server, 100, [debug: [:statistics]])
{:ok,#PID<0.69.0>}
iex> GenServer.call(pid, :next_number)
100
iex> GenServer.call(pid, :next_number)
101
iex> :sys.statistics pid, :get
{:ok,[start_time: {{2013,4,26},{18,17,16}}, current_time: {{2013,4,26},{18,17,28}},
      reductions: 50, messages_in: 2, messages_out: 0]}
```

- Pass `[debug: [:statistics]]` to generate tracking of common server stats

Simple Example (V)

```
#####
# External API

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

def next_number do
  GenServer.call __MODULE__, :next_number
end

def increment_number(delta) do
  GenServer.cast __MODULE__, {:increment_number, delta}
end
```

- Add a simple client interface to the server module; clients can now use this interface, rather than dealing with start_link and pids themselves

Exercise for Reader: Implement a Stack

- In this chapter, the book asks the reader to implement an actor that acts like a stack
  - we start with implementing the "pop" operation
  - and then we'll add an operation to push something onto the stack
- First, we create a new mix project and create a file for our code
  - `mix new stack`
  - `cd stack`
  - `mkdir lib/stack`
  - `vi lib/stack/server.ex`

Implementing Stack (I)

- Start with the basic template

```
defmodule Stack.Server do

  use GenServer

  def init(state) do
    IO.puts("Starting Stack actor with state: #{inspect state}")
    {:ok, state}
  end

end
```

- We can now create an instance of our stack actor (it won't do anything yet)
  - `iex -S mix`
  - `{:ok, pid} = GenServer.start_link(Stack.Server, [1, 2, 3, "four"])`
  - `=> Prints "Starting Stack actor with state: [1, 2, 3, "four"]"`

Implementing Stack (II)

- Now, we implement the "pop" operation using handle_call

```
def handle_call(:pop, _from, []) do
  {:reply, nil, []}
end

def handle_call(:pop, _from, [head | tail]) do
  {:reply, head, tail}
end
```

- We use `handle_call` since we want a caller to block until a reply is received
  - `iex -S mix`
  - `{:ok, pid} = GenServer.start_link(Stack.Server, [1, 2, 3, "four"])`
  - `GenServer.call(pid, :pop) # call this five times`
  - `=> Returns 1, 2, 3, "four", and nil`

Implementing Stack (III)

- Now, we implement the "push" operation using handle_cast

```
def handle_cast({:push, value}, state) do
  IO.puts("New state: #{inspect [value|state]}")
  {:noreply, [value|state]}
end
```

- We use `handle_cast` since we don't need a reply when pushing onto a stack
  - `iex -S mix`
  - `{:ok, pid} = GenServer.start_link(Stack.Server, [1, 2, 3, "four"])`
  - `GenServer.cast(pid, {:push, 5}) # adds 5 to stack`
  - `=> Prints "New state: [5, 1, 2, 3, "four"]"`

Implementing Stack (IV)

- Now, let's implement a "stop" operation to terminate a Stack actor

```
def handle_cast({:push, value}, state) do
  IO.puts("New state: #{inspect [value|state]}")
  {:noreply, [value|state]}
end
```

- We use `handle_cast` since we don't need a reply for this situation
  - `iex -S mix`
  - `{:ok, pid} = GenServer.start_link(Stack.Server, [1, 2, 3, "four"])`
  - `GenServer.cast(pid, :stop)`
  - `=> Prints "New state: [5, 1, 2, 3, "four"]"`

Supervisors

- OTP provides a behavior (i.e. module) that makes it easy to
  - create actors that supervise other actors (workers)
  - restart workers according to a defined policy
  - handle restart loops (launch => crash => restart => crash => …)
- mix is able to generate a template for us when an application involves one type supervisor handling one type of worker
  - mix new --sup sequence2
  - Open up lib/sequence2.ex

Supervisor Template

```
defmodule Sequence do
  use Application

  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    children = [
      # Define workers and child supervisors to be supervised
      # worker(Sequence.Worker, [arg1, arg2, arg3])
    ]

    opts = [strategy: :one_for_one, name: Sequence.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Template creates a single supervisor that will relaunch its child workers "one for one"; uses Supervisor.start_link to create the supervisor

Fill in the template (I)

- To prepare, we take the Sequence.Server app from earlier in the lecture, and
  - copy it to lib/sequence2
  - rename it to be called Sequence2.Server
  - update the template to reference this new worker type
- Note: this particular GenServer is set-up to only ever have one instance
  - For now, we ask the Supervisor to only create one child worker

Fill in the template (II)

```
def start(_type, _args) do
  import Supervisor.Spec, warn: false

  children = [
    worker(Sequence.Server, [123])
  ]

  opts = [strategy: :one_for_one, name: Sequence.Supervisor]
  {:ok, _pid} = Supervisor.start_link(children, opts)
end
```

- Here we update the template to contain a value in the children array
  - We would add one entry to the array for each child we want the supervisor to manage; for now, we just create one child
    - Note: our actual code will read Sequence2.Server

Using the supervisor

- With this set-up, iex will call "start" for us automatically
  - That in turn creates a supervisor which then creates the Sequence server.
- If we do something to cause the Sequence server to fail, then it gets restarted
  - Such as `Sequence2.Server.increment_number "ken"`
- When this happens, however, we lose our state. If we had incremented the number, we're no longer where we were; instead, the state goes back to its original value
  - To fix that, we need to create another actor to store the state for us and a second supervisor that manages this new actor (called the stash) and a new supervisor for the current supervisor
    - We will look at that design in a moment

Supervisors: What have we learned so far?

- Supervisors are just actors; when we create them, we get back a process id
  - e.g. `{:ok, _pid} = Supervisor.start_link(children, opts)`
- We launch Supervisors with a call to `Supervisor.start_link`
  - Documentation for Supervisor
- Supervisors manage child processes which can be either GenServer actors or other Supervisors; this allows supervision hierarchies to be created
- There are several ways to attach children to a supervisor; but no matter which method you use, you need to provide a "spec" for that child
  - How do you do that? With methods in Supervisor.Spec
  - Let's see what these look like

Supervision Specifications (I)

- We can gain insight into these specs using iex (as always)
  - `iex -S mix` # within the sequence3 mix project
  - `import Supervisor.Spec, warn: false` # bring spec functions into scope
  - `worker(Sequence.Server, [500])` # create a worker spec
    - => Returns a tuple about the worker with a number of default options
    - `{Sequence.Server, {Sequence.Server, :start_link, [500]}, :permanent, 5000, :worker, [Sequence.Server]}`
  - `worker(Sequence.Server, [500], restart: :temporary, shutdown: :infinity)`
    - `{Sequence.Server, {Sequence.Server, :start_link, [500]}, :temporary, :infinity, :worker, [Sequence.Server]}`
    - In this case, we passed in some arguments about how we want the worker to behave when restarting and shutting down

Supervision Specifications (II)

- We can also add a supervisor as a child. Instead of calling `worker`, we instead call `supervisor`
  - `iex -S mix` # within the sequence3 mix project
  - `import Supervisor.Spec, warn: false` # bring spec functions into scope
  - `supervisor(Sequence3.SubSupervisor, [self])`
    - => Returns a tuple about the supervisor with default options
  - `{Sequence3.SubSupervisor, {Sequence3.SubSupervisor, :start_link, [#PID<0.112.0>]}, :permanent, :infinity, :supervisor, [Sequence3.SubSupervisor]}`

Returning to the code

```
def start(_type, _args) do
  import Supervisor.Spec, warn: false

  children = [
    worker(Sequence.Server, [123])
  ]

  opts = [strategy: :one_for_one, name: Sequence.Supervisor]
  {:ok, _pid} = Supervisor.start_link(children, opts)
end
```

- This approach to launching a supervisor is just one way of doing it
  - We define the children specs in an array; create an array of options; and pass both arrays to Supervisor.start_link
- Another approach is to define the supervisor as a module

Module-based Supervisors

```
defmodule Sequence.SubSupervisor do
  use Supervisor

  def start_link(stash_pid) do
    {:ok, _pid} = Supervisor.start_link(__MODULE__, stash_pid)
  end
  def init(stash_pid) do
    child_processes = [ worker(Sequence.Server, [stash_pid]) ]
    supervise child_processes, strategy: :one_for_one
  end
end
```

- Here we create a module and import the Supervisor behavior; we call a version of start link that defines the name of the supervisor and its initial state; this causes the init function to be called;
  - In init, we define our child workers and options and call "supervise"

Module-based Supervisors with Dynamic Children

```
defmodule Sequence.Supervisor do
  use Supervisor
  def start_link(initial_number) do
    result = {:ok, sup } = Supervisor.start_link(__MODULE__, [initial_number])
    start_workers(sup, initial_number)
    result
  end
  def start_workers(sup, initial_number) do
    # Start the stash worker
    {:ok, stash} =
       Supervisor.start_child(sup, worker(Sequence.Stash, [initial_number]))
    # and then the subsupervisor for the actual sequence server
    Supervisor.start_child(sup, supervisor(Sequence.SubSupervisor, [stash]))
  end
  def init(_) do
    supervise [], strategy: :one_for_one
  end
end
```

- Here, we create a supervisor with no children and then add them later

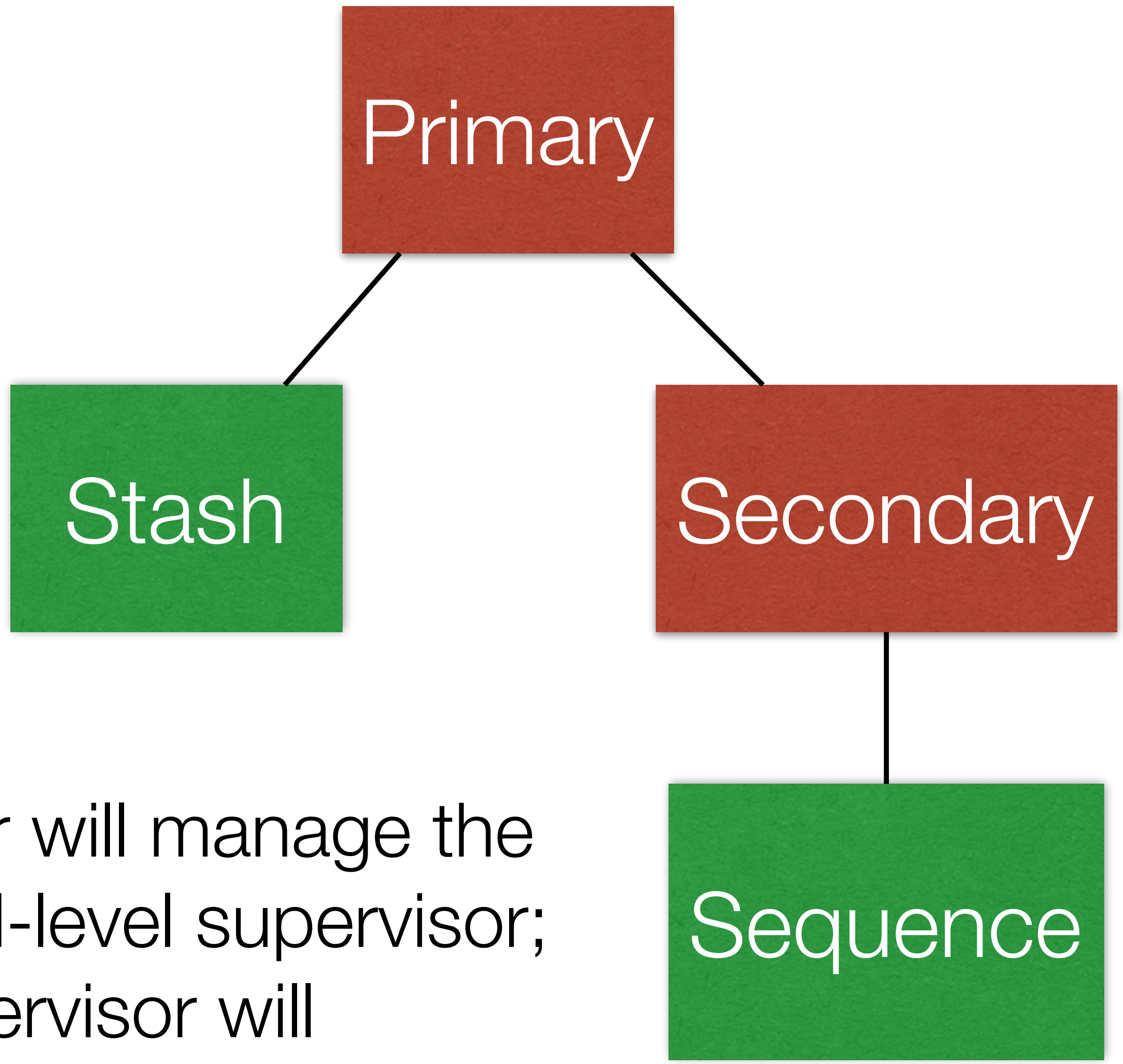
What are Strategies?

- The most common argument specified for a supervisor is its "strategy"
  - This refers to what does a supervisor do when one of its workers dies?
- These are the options
  - :one_for_one => if a child dies, only that process is restarted
  - :one_for_all => if a child dies
    - all other child processes are terminated
    - all children are then restarted
  - :rest_for_one => if a child dies
    - all child processes that started after it are terminated
    - then the child plus these other children are restarted
  - :simple_one_for_one => used in the situation where children are dynamically added and removed over the course of the application

Sequence program "with memory"

- We now return to our example where we would like to make sure that
  - when a sequence server dies, it doesn't forget its state
- We will see examples of both types of module-based supervisors in this program
  - Recall that the idea is that the sequence server will store its state in another actor and can then retrieve it
  - Let's look at the supervision hierarchy that we will use

Supervision Tree



A top-level supervisor will manage the "stash" and a second-level supervisor; the second-level supervisor will manage the Sequence server as in the previous example

Making the Change

- We'll review the code next but, at a high level, we do the following
  - We move the creation of a supervisor out of the main start method
    - Instead, it invokes a special purpose start_link method in a new module
  - That new module creates the primary supervisor
    - and adds two children to it
      - the stash (it receives our initial sequence number)
      - and the secondary supervisor (it receives the pid of the stash)
  - The secondary supervisor then creates an instance of the sequence server and passes it the pid of the stash
  - The sequence server is modified to keep track of its state, plus the pid of the stash; it reads its starting value from the stash and it writes its current number to the stash whenever it is terminated

Wrapping Up

- We've scratched the surface of OTP and seen the basic support for
  - creating servers with GenServer
    - learned about the various callback methods: init, handle_call, handle_cast, terminate, etc.
  - creating supervisors with Supervisor
    - saw two different ways of creating a supervisor
      - one in which the children are specified ahead of time
      - the second in which a supervisor is started with zero children and then child actors are added to it dynamically.
- We will next learn about OTP applications and conclude our review of using Elixir for building distributed concurrent applications