Course Content

udemy.com/elixir-scalable-and-efficient-application-development/learn/v4/t/lecture/7102958

Outside of OTP, application is a very general term and we toss it around without much care. "There's an app for that," comes to mind often. In this less specific and constrained perspective, applications are general, not necessarily single purpose, programs that perform things for us. We have seen many different forms of applications take form over the years—desktop applications, web applications, and now mobile applications. For desktop applications, we can think of our word processors, web browsers, text editors, IDEs, games, and so on. For web applications, we can think of our social media sites, task tracking applications, games, office suites, and so on. The mobile app space contains similar applications to those already listed. The applications are mostly the same, differentiated only by platform.

With respect to OTP, applications are self-contained process trees that serve some purpose, or may even wrap several (OTP) applications together as a new (super) singular application. The question of how applications are defined, started, and managed is typically accomplished via process supervision trees, but this detail isn't necessary to understanding OTP applications in general.

The difference between applications and OTP applications is mostly contextual. However, the word can be scary to newcomers.

A great example of an OTP application that we have already used numerous times is `iex`. From all our examples, so far, running `Process.list/0` returns the current list of processes running:

```
iex(1)> Process.list
[#PID<0.0.0>, #PID<0.3.0>, #PID<0.6.0>, #PID<0.7.0>, #PID<0.9.0>, #PID<0.10.0>,
 #PID<0.11.0>, #PID<0.12.0>, #PID<0.14.0>, #PID<0.15.0>, #PID<0.16.0>,
 #PID<0.17.0>, #PID<0.18.0>, #PID<0.20.0>, #PID<0.21.0>, #PID<0.22.0>,
 #PID<0.23.0>, #PID<0.24.0>, #PID<0.25.0>, #PID<0.26.0>, #PID<0.27.0>,
 #PID<0.28.0>, #PID<0.29.0>, #PID<0.30.0>, #PID<0.38.0>, #PID<0.39.0>,
 #PID<0.40.0>, #PID<0.41.0>, #PID<0.42.0>, #PID<0.43.0>, #PID<0.45.0>,
 #PID<0.46.0>, #PID<0.47.0>, #PID<0.48.0>, #PID<0.51.0>, #PID<0.52.0>,
 #PID<0.53.0>, #PID<0.54.0>, #PID<0.55.0>, #PID<0.56.0>, #PID<0.57.0>,
 #PID<0.58.0>, #PID<0.60.0>]
```

These are all the different processes bundled up by the `iex` application, many of which are themselves OTP applications. For example, we can see the started, named applications when we launch an interactive session:

```
iex(2)> :application.which_applications
[{:stdlib, 'ERTS  CXC 138 10', '2.5'}, {:elixir, 'elixir', '1.0.5'},
 {:kernel, 'ERTS  CXC 138 10', '4.0'}, {:iex, 'iex', '1.0.5'},
 {:logger, 'logger', '1.0.5'}, {:compiler, 'ERTS  CXC 138 10', '6.0'},
 {:syntax_tools, 'Syntax tools', '1.7'}, {:crypto, 'CRYPTO', '3.6'}]
```

The tuple returned by `:application.which_applications/0` is defined as `{Application, Description, Vsn}`, where `Application` is the name of the application, `Description` is either the application name in string form or an explanatory text of the application, and `Vsn` is the loaded version of the application `(1)`.

What does it mean for an application to be loaded? The Erlang VM has a method for hot-swapping code, and therefore, being able to keep track of the loaded module version becomes more important. So, the "loaded" version is the currently in-memory version of the application (which is not necessarily the latest version).

Overall, the word application means what we would expect, a single entity for working with and manipulating a unit of code. The unit of code itself could be many different things—an API library for querying an HTTP endpoint, a distributed key-value store, `iex` itself, or whatever else you can imagine and develop.

Gen(eric) behaviours

OTP defines several generic behaviours we can use when creating Elixir applications. There is the `GenServer` behaviour, the `GenEvent` behaviour, and the `:gen_fsm` behaviour. All of these behaviours have their foundation in an even more general behaviour of OTP processes.

These behaviours remove some of the tedious work we had to do for handling messages and performing work that we encountered in the previous chapter.

We will start with our discussion on `GenServer`, and then move onto more specialized variants.

Gen(eric) servers

OTP gives us the basic blueprint for a process that receives messages, processes messages and sends a result back, like any server would.

Gen in `GenServer` really stands for generic or general because it provides the general details of such a process without constraining its users too much into an inflexible solution. For example, we saw that the main event loop of the processes we wrote in the previous chapter were all very similar in nature; the only real differences between each were the messages the process responded to and its handling of those messages. Most, if not all, other details were the same. This is what the `GenServer` behaviour provides for us.

For a quick example, let's recreate our key-value store from the previous chapter.

We will start with the skeleton code of any `GenServer` module:

```
1. defmodule KV do
2.   use GenServer
3.   def start_link(opts \\ []) do
4.     GenServer.start_link(__MODULE__, [], opts)
5.   end
6.   def init(_) do
7.     {:ok, HashDict.new}
8.   end
9. end
```

To create a `GenServer` process, we define a regular Elixir module. However, unlike other modules we've so far created, we have `use GenServer` at the beginning. This instructs Elixir that the module we are defining will use the `GenServer` behaviour.

Next, we define a few functions usually required to get things off the ground:

```
1. def start_link(opts \\[]) do
2.   GenServer.start_link(__MODULE__, [], opts)
3. end
```

Instead of calling `Process.spawn` or `Process.spawn_link`, we use the helper function from the `GenServer` module to start up our process. This helper function will call our second function, `init/1`, as shown in the following code:

```
1. def init(_) do
2.   {:ok, HashDictnew}
3. end
```

The `init/1` function will be called to set up the state of the process and make sure the initial state is good. Most often, it will be the single line, `{:ok, state}`, where `state` is the internal state object of the process, initialized to the proper state. Since we are creating a simple key-value store, we can use the *HashDict* structure for our process state.

If the first atom of the tuple is anything other than `:ok`, the process will refuse to start. This is useful when upstream dependencies or other guarantees are unsatisfied and it is not desired to start under such circumstances.

This is enough code to get the server started and off the ground; however, it won't be very useful since we haven't defined a few other functions.

The `GenServer` processes have a few functions for handling `messages–handle_call/3` and `handle_cast/3`. These functions serve similar purposes, but have different behaviours. Moreover, these functions are never called directly by client code; they are called internally by the OTP framework. The OTP framework takes care of the internal main loop for each process, dispatching messages to our versions of `handle_call/3` and `handle_cast/3`.

Continuing our key-value example, let's define our `handle_call/3` function:

```
def handle_call({:put, key, value}, _from, dictionary) do
  {:reply, :ok, HashDict.put(dictionary, key, value)}
end
def handle_call({:get, key}, _from , dictionary) do
  {:reply, HashDict.get(dictionary, key), dictionary}
end
```

The first parameter of `handle_call/3` is the message tuple. Typically, we have an atom specifying the message type or action the server should perform, the second parameter (which we don't use in this example) is the calling process, and the third parameter is the internal data of the process, in this case, the dictionary we are storing data in.

The first pattern for `handle_call/3` matches against a triple of `{:put, key, value}`, where `key` is the key or identifier of the data being stored and `value` is the actual data being stored. The second pattern matches `{:get, key}`, where `key` is the identifier of the data to retrieve.

The returned triple is consumed by the OTP framework. The first element of the triple informs the OTP framework how to handle the response. The second element is the value to return to the calling process. The third is the internal state, modified or not, to be used the next time a message is received.

This is fairly similar to how our processes from the previous chapter worked, except the OTP framework takes care of a lot of the tedious plumbing required to get two processes communicating.

For completeness, here is the `KV` module so far:

```
defmodule KV do
  use GenServer
  def start_link(opts \\ []) do
    GenServer.start_link(__MODULE__, [], opts)
  end
  def init(_) do
    {:ok, HashDict.new}
  end
  def handle_call({:put, key, value}, _from, dictionary) do
    {:reply, :ok, HashDict.put(dictionary, key, value)}
end
  def handle_call({:get, key}, _from, dictionary) do
    {:reply, HashDict.get(dictionary, key), dictionary}
  end
end
```

From here, we can actually try out the process in an interactive session:

```
iex(1)> import_file "kv.exs"
{:module, KV,
 <<70, 79, 82, 49, 0, 0, 12, 136, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 243,
131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
 {:handle_call, 3}}
iex(2)> {:ok, kv_pid} = KV.start_link
{:ok, #PID<0.69.0>}
```

Notice, the `GenServer.start_link/3` function actually returns a tuple, `{:ok, pid}`. In the previous chapter, we only returned the PID, but here, we are returning a tag for the PID. This let's the calling or parent process know that the starting or child process started successfully.

Now, using the PID and the `GenServer.call/2` function, we can send our message to the key-value process:

```
1. iex(3)> GenServer.call(kv_pid, {:put, :a, 42})
2. :ok
```

We can also use the `GenServer.call/2` function to retrieve data from the key-value process as well:

```
1. iex(4)> GenServer.call(kv_pid, {:get, :a})
2. 42
```

But, why should the client process depend on using the `GenServer.call/2` function? This should be something the `KV` module handles for the client. That is, we can add some helper functions to the `KV` module that will call and send the correct messages for clients:

```
1. def put(server, key, value) do
2.   GenServer.call(server, {:put, key, value})
3. end
4. def get(server, key) do
5.   GenServer.call(server, {:get, key})
6. end
```

These two functions are no different than what we ran previously in `iex`; we are just wrapping them up into the `KV` module.

Reloading and restarting the `KV` process, we can use the new functions:

```
1. iex(1)> import_file "kv.exs"
2. {:module, KV,
3.  <<70, 79, 82, 49, 0, 0, 12, 136, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 243,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.  {:handle_call, 3}}
5. iex(2)> {:ok, kv_pid} = KV.start_link
6. {:ok, #PID<0.67.0>}
7. iex(3)> KV.put(kv_pid, :a, 42)
8. :ok
9. iex(4)> KV.get(kv_pid, :a)
10. 42
```

The helper functions we create can be taken a step further; if we know there is only ever going to be one instance of the process, we can have the process registered as soon as it starts. To do this, we need to add a key-value pair to the `opts` list in `KV.start_link/1`; change the definition of `KV.start_link` to the following code:

```
1. def start_link(opts \\ []) do
2.   GenServer.start_link(__MODULE__, [], [name: __MODULE__] ++ opts)
3. end
```

That is, we pass the pair, `[name: __MODULE__]`, and whatever parent process passes to `GenServer.start_link/3`. By providing the `:name` attribute in `opts`, `GenServer.start_link/3` will register the process for us after it starts. This means we don't need to reference it by the PID returned. In fact, we can change `KV.put/3` and `KV.get/2` to be `KV.put/2` and `KV.get/1` with the following definitions:

```
1. def put(key, value) do
2.   GenServer.call(__MODULE__, {:put, key, value})
3. end
4. def get(key) do
5.   GenServer.call(__MODULE__, {:get, key})
6. end
```

After the process is registered, we can use the `__MODULE__` directive to reference the registered name of the module. That is, the new version of `KV` is the following:

```
1. defmodule KV do
2.   use GenServer
3.   def start_link(opts \\ []) do
4.     GenServer.start_link(__MODULE__, [], [name: __MODULE__] ++ opts)
5.   end
6.   def init(_) do
7.     {:ok, HashDict.new}
8.   end
9.   def put(key, value) do
10.    GenServer.call(__MODULE__, {:put, key, value})
11.  end
12.  def get(key) do
13.    GenServer.call(__MODULE__, {:get, key})
14.  end
15.  def handle_call({:put, key, value}, _from, dictionary) do
16.    {:reply, :ok, HashDict.put(dictionary, key, value)}
17.  end
18.  def handle_call({:get, key}, _from, dictionary) do
19.    {:reply, HashDict.get(dictionary, key), dictionary}
20.  end
21. end
```

Using our new version of this module tends to be the same:

```
1. iex(1)> import_file "kv.exs"
2. {:module, KV,
3.  <<70, 79, 82, 49, 0, 0, 12, 152, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 207,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.  {:handle_call, 3}}
5. iex(2)> KV.start_link
6. {:ok, #PID<0.76.0>}
7. iex(3)> KV.put(:b, 42)
8. :ok
9. iex(4)> KV.get(:b)
10. 42
```

This time around, however, we do not need to know the PID of the server process. This may seem familiar to process registration in the previous chapter. In fact, what `GenServer.start_link/3` and `Process.register/2` are doing is, in effect, the same. To see this, remember from the previous chapter, we discussed an internal Elixir function, `Process.registered/0`, which returns the keys of all the registered, named applications. If we run this function from inside our current `iex` session, we will see that `KV`, our key-value server process, is among the members in the list:

```
1. iex(4)> Process.registered
2. [:init, :file_server_2, :user, :application_controller, KV, :user_drv,
3.  :standard_error, :global_group, :elixir_counter, IEx.Supervisor,
4.  :elixir_config, :elixir_sup, Logger, :elixir_code_server, IEx.Config,
5.  :error_logger, :standard_error_sup, :kernel_safe_sup, :rex, :erl_prim_loader,
6.  Logger.Supervisor, :inet_db, :kernel_sup, :code_server, :global_name_server,
7.  Logger.Watcher]
```

Do not be fooled by the upper cased naming; every element of the list is an atom. Try `Process.registered |> Enum.at(n) |> is_atom`, where `n` is the index of `KV`. The result should be `true`.

We can also try the following command:

```
1. iex(5)> KV in Process.registered
2. true
```

This also definitively gives the desired result.

Asynchronous messaging

It may seem that we have lost asynchronous communication by switching to OTP and using the `GenServer` behaviour. That is, `GenServer.call/2` is a blocking, synchronous invocation, and the calling process must wait for the reply before continuing, effectively halting any form of parallelism that may have been exploited.

The `GenServer.call/2` function is not the end of the discussion though, there is still some hope for asynchrony yet.

There is another function available to us when creating `GenServer` processes — `GenServer.cast/2`. This function, instead of blocking, casts the message to the `GenServer` process and returns, practically, immediately.

Like `GenServer.call/2` requires `handle_call/3` to be defined, `GenServer.cast/2` requires a new function, `handle_cast/2`, to be defined. Luckily for us, this function too isn't much different though. Let's demonstrate with a very simplistic `PingPong` module:

```
defmodule PingPong do
  use GenServer
  def start_link(opts \\ []) do
    GenServer.start_link(__MODULE__, [], [name: __MODULE__] ++ opts)
  end
  def ping() do
    GenServer.cast(__MODULE__, {:ping, self()})
  end
  def handle_cast({:ping, from}, state) do
    send from, :pong
    {:noreply, state}
  end
end
```

First, we define the `PingPong` module and have it use the `GenServer` behaviour. The `start_link` and `ping` functions are helper functions for setting up and making the public API of the process easier. The `init/1` function only needs to be redefined when there is process state that must be initialized; otherwise, the default is fine.

In the `handle_cast/2` function, we match against a pattern or request of `{:ping, from}`, where `from` should be a PID. Upon receiving a `{:ping, from}` message, we send back the `:pong` message and return `{:noreply, state}`.

Unlike `handle_call/3` from earlier, we only return the success status of `:noreply` and the state. There is no direct return to the calling process; we have no channel to do so without using `send/2`, and using `send` requires a reference to the calling process.

Using our new `PingPong` module quite similar to before, we simply start and cast into it:

```
iex(1)> import_file "pingpong.exs"
{:module, PingPong,
 <<70, 79, 82, 49, 0, 0, 11, 48, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 124,
131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
 {:handle_cast, 2}}
iex(2)> PingPong.start_link
{:ok, #PID<0.82.0>}
iex(3)> PingPong.ping
:ok
iex(4)> flush
:pong
:ok
```

We don't receive the `:pong` message when calling `PingPong.pong/0`; we will receive it when we call `flush/0` though. However, it should be noted that `PingPong.ping/0` still returns `:ok` from `GenServer.cast/2`. From the documentation of `GenServer.cast/2`:

"This function returns `:ok` immediately, regardless of whether the destination node or server does exists, unless the server is specified as an atom."

Returning `:ok` is par for the course, but returning it immediately changes things. Since there is now inherent asynchrony happening, immediately calling `flush/0` after casting a message to the server process may not always return results. To demonstrate this, try adding `:timer.sleep 5000` into the `handle_cast/2` function .

Testing `GenServer.cast/2` functions from a public API perspective can be highly frustrating and mildly tedious. We will cover some testing strategies around these functions shortly.

Gen(eric) events

Sending messages to a `GenServer` process can be thought to be similar to sending events. A tagged message to perform some action received and the `GenServer` process will compute some result or perform some action, based on the inputs of the message and current state of the process. However, performing complex logic around forwarding, dropping, filtering, and other such actions in the context of the `GenServer` behaviour is quite daunting.

The `GenEvent` behaviour handles many of the issues of developing an event-based process inside the context of OTP processes.

The `GenEvent` behaviour acts as an event dispatcher; it receives and forwards events to handlers. The handlers are in charge of performing actions on the received events. There are a number of applications of this model and process, there are even a few already in Elixir and OTP. For example, Elixir's `Logging` module is a `GenEvent` process, forwarding the log events to the console or any additional handler.

We can create our first `GenEvent` manager using the `GenEvent.start_link/0` function:

```
1. iex(1)> {:ok, event_manager} = GenEvent.start_link
2. {:ok, #PID<0.62.0>}
```

After we have created our managing process, we can use either `GenEvent.sync_notify/2` or `GenEvent.notify/2` to send events to the manager process:

```
1. iex(2)> GenEvent.sync_notify(event_manager, :foo)
2. :ok
3. iex(3)> GenEvent.notify(event_manager, :bar)
4. :ok
```

However, since the event manager has no handlers, the events are dropped. Let's define and add a basic forward handler. It will receive the event and send it to the parent process:

```
iex(4)> defmodule Forwarder do
...(4)>   use GenEvent
...(4)>   def handle_event(event, parent) do
...(4)>     send parent, event
...(4)>     {:ok, parent}
...(4)>   end
...(4)> end
{:module, Forwarder,
 <<70, 79, 82, 49, 0, 0, 9, 156, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 10,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
 {:handle_event, 2}}
iex(5)> GenEvent.add_handler(event_manager, Forwarder, self())
:ok
iex(6)> GenEvent.sync_notify(event_manager, :ping)
:ok
iex(7)> flush
:ping
:ok
```

In the handler module, `Forwarder`, we define the `handle_event/2` function for receiving events. This function takes the event and the current state variable of the process, sends the event to the process defined by `parent`, and then returns `{:ok, parent}` for the OTP main loop.

After defining the module, we add it to the handler we started earlier and send another event to the manager. This time, though, the event is dispatched to our handler. Our handler (`Forwarder`) receives the message and uses `send/2` to send it to the parent process (the current IEx session). Thus, when we call `flush/0`, we see the `:ping` event message.

Without breaking the single responsibility principle, we can have more handlers do more things with the same events. For example, let's add another handler to the current example that prints the message to the console:

```
iex(8)> defmodule Echoer do
...(8)>   use GenEvent
...(8)>   def handle_event(event, []) do
...(8)>     IO.puts event
...(8)>     {:ok, []}
...(8)>   end
...(8)> end
{:module, Echoer,
 <<70, 79, 82, 49, 0, 0, 9, 156, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 2, 10,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
 {:handle_event, 2}}
```

Here, we define another handler that is very similar to the `Forwarder` from before. Instead, we call `IO.puts event` instead of `send parent, event`. We don't really need to keep track of any internal state, so we pass around an empty list.

After defining the handler, let's add it to the event manager and send an event to the manager:

```
iex(9)> GenEvent.add_handler(event_manager, Echoer, [])
:ok
iex(10)> GenEvent.sync_notify(event_manager, :hello)
hello
:ok
iex(11)> flush
:hello
:ok
```

Immediately, we see the message printed to the output of the console. Furthermore, we can flush the current process mailbox and see the message is there as well.

This also works with the asynchronous variant as well:

```
iex(12)> GenEvent.notify(event_manager, :world)
hello
:ok
iex(13)> flush
:hello
:ok
```

To better demonstrate the capabilities of asynchronously sending events, we will need to either perform more work or artificially pause one of the handlers. Try adding a `:timer.sleep` to the Echoer handler, and try both `GenEvent.sync_notify/2` and `GenEvent.notify/2`.

Special OTP processes

The OTP behaviours we have covered so far are often sufficient for a lot of the applications being developed. However, how do we get the benefits of supervision trees and the near-infinite level of traceability and debugging capabilities of the BEAM, and, most importantly, take control of the main event loop of the OTP process? If the task we are trying to solve does not fit into either of the OTP behaviours we have already covered, there is still a way to get a good number of benefits of OTP without using the given behaviours. These are referred to as special OTP processes in the Erlang world. There are a few requirements on the developer's part and as such, the use of special processes should be avoided if at all possible.

Unlike the processes we wrote in the previous chapter, these processes will conform to the OTP design principles and support the many tracing and debugging facilities inherent with the OTP processes and Erlang VM.

The definition of special process modules is very similar to that of the modules using the regular behaviours. They will have a `start_link` function, an `init` function, and a few system functions for handling specific OTP messages. As a simple example, let's create the `PingPong` module as a special process.

We will start with the `start_link` function:

```
def start_link(opts \\ []) do
  :proc_lib.start_link(__MODULE__, :init, [self(), opts])
end
```

So far, this isn't much different than what might be expected. We start and link the current module using the `init` function, passing the parent and `opts`. However, instead of using `spawn_link/3`, we are using `start_link/3` from the Erlang module, `proc_lib`. The difference between this function and `spawn_link/3` is that the former starts the process synchronously. Furthermore, as we will see, the resulting process must call `:proc_lib.init_ack/2` to signal the start of the process; if `:proc_lib.init_ack/2` is never called, the parent process will hang forever. There is a variation of `:proc_lib.start_link/4` that includes a `timeout` parameter; however, most often this can be handled by the supervisor process.

The `init` function is slightly more involved:

```
1. def init(parent, opts) do
2.   debug = :sys.debug_options([])
3.   Process.link(parent)
4.   :proc_lib.init_ack(parent, {:ok, self()})
5.   Process.register(self(), __MODULE__)
6.   state = HashDict.new
7.   loop(state, parent, debug)
8. end
```

We start by creating the `debug` object from the `:sys` module. This object will be used for retrieving debugging information; most often, the process will not use it directly.

Next, we create a link back to the parent process and call `init_ack`. This tells the parent process that child process is up and running. The tuple passed to `init_ack` is what is returned by the `:proc_lib.start_link` call from the `start_link` function. The call to `init_ack` should not happen until after all of the required dependencies and information for a successful start of the process are complete. That is, this function is how we tell the parent process that the child is successfully started or failed because of a `:badarg` error or missing dependency.

Next, optionally, we can register the process in the local nodes process table. The process can also have some more sophistication around process registration as well, such as, if the `:name` attribute is in the `opts` list, then register the process.

Finally, initialize the state dictionary and invoke the `loop` function passing the `state` dictionary, the parent process reference, and the `debug` object. The OTP design principles require that these are all held onto during the lifetime of the process.

The `loop` function could really be named anything you want, I am just using `loop` for consistency with the previous chapter, and it's not the worst name to use for this context.

Next, we need to define the `loop/3` function for our special process:

```
1. defp loop(state, parent, debug) do
2.   receive do
3.     {:ping, from} ->
4.       send from, :pong
5.     {:system, from, request} ->
6.       :sys.handle_system_msg(request, from, parent, __MODULE__, debug, state)
7.   end
8.   loop(state, parent, debug)
9. end
```

When entering the `loop` function, the process will block, waiting for either a `:ping` message or a `:system` message. The `:ping` message is the message we care about receiving. The `:system` message, on the other hand, is an OTP message requirement. Luckily, we can leverage the Erlang `:sys` module to handle the `:system` messages. However, it also means we must define a few more `functions–system_continue/3`, `system_terminate/4`, and `system_get_state/1`. These functions are trivial to define and are really only a tedium for now:

```
1. def system_continue(parent, debug, state), do: loop(state, parent, debug)
2. def system_terminate(reason, _, _, _): do, exit(reason)
3. def system_get_state(state): do: {:ok, state}
```

These functions should never be called by users or clients, but they still need to be public for the OTP behaviours.

That covers the bare minimum required to get a special OTP process off the ground. Let's take it for a spin:

```
1. iex(1)> import_file "pingpong_sp.exs"
2. {:module, PingPong,
3.  <<70, 79, 82, 49, 0, 0, 11, 192, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 1, 212,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.  {:system_get_state, 1}}
5. iex(2)> PingPong.start_link
6. {:ok, #PID<0.79.0>}
7. iex(3)> send PingPong, {:ping, self()}
8. {:ping, #PID<0.60.0>}
9. iex(4)> flush
10. :pong
11. :ok
```

We are back to the API from the previous chapter; we lost our ability to use the `Gen*` functions for sending messages to our process. We can add a function to abstract away the `send/2` call:

```
1. def ping() do
2.   send __MODULE__, {:ping, self()}
3.   :ok
4. end
```

But, we've had a taste of `GenServer.call`. It would be nice if we had something along those lines.

We can use receive; let's try `ping/0` another way. Add the following function to our special

process version of `PingPong`:

```
1. def ping() do
2.   send __MODULE__, {:ping, self()}
3.   receive do
4.     {:reply, response} ->
5.       response
6.   after 10000 ->
7.     {:error, :timeout}
8.   end
9. end
```

This code is similar to our first idea; however, it gives the result in a synchronous and blocking manner. We send the `:ping` message to the `PingPong` process and then wait for the `:reply` message. Upon receiving the `:reply` message, return the response. Additionally, if no `:reply` is returned in 10 seconds, return the `{:error, :timeout}` tuple.

Now, adjust the handling code for the `:ping` message:

```
1. defp loop(state, parent, debug) do
2.   receive do
3.     {:ping, from} ->
4.       send from, {:reply, :pong}
5.     {:system, from, request} ->
6.       :sys.handle_system_msg(request, from, parent, __MODULE__, debug, state)
7.   end
8.   loop(state, parent, debug)
9. end
```

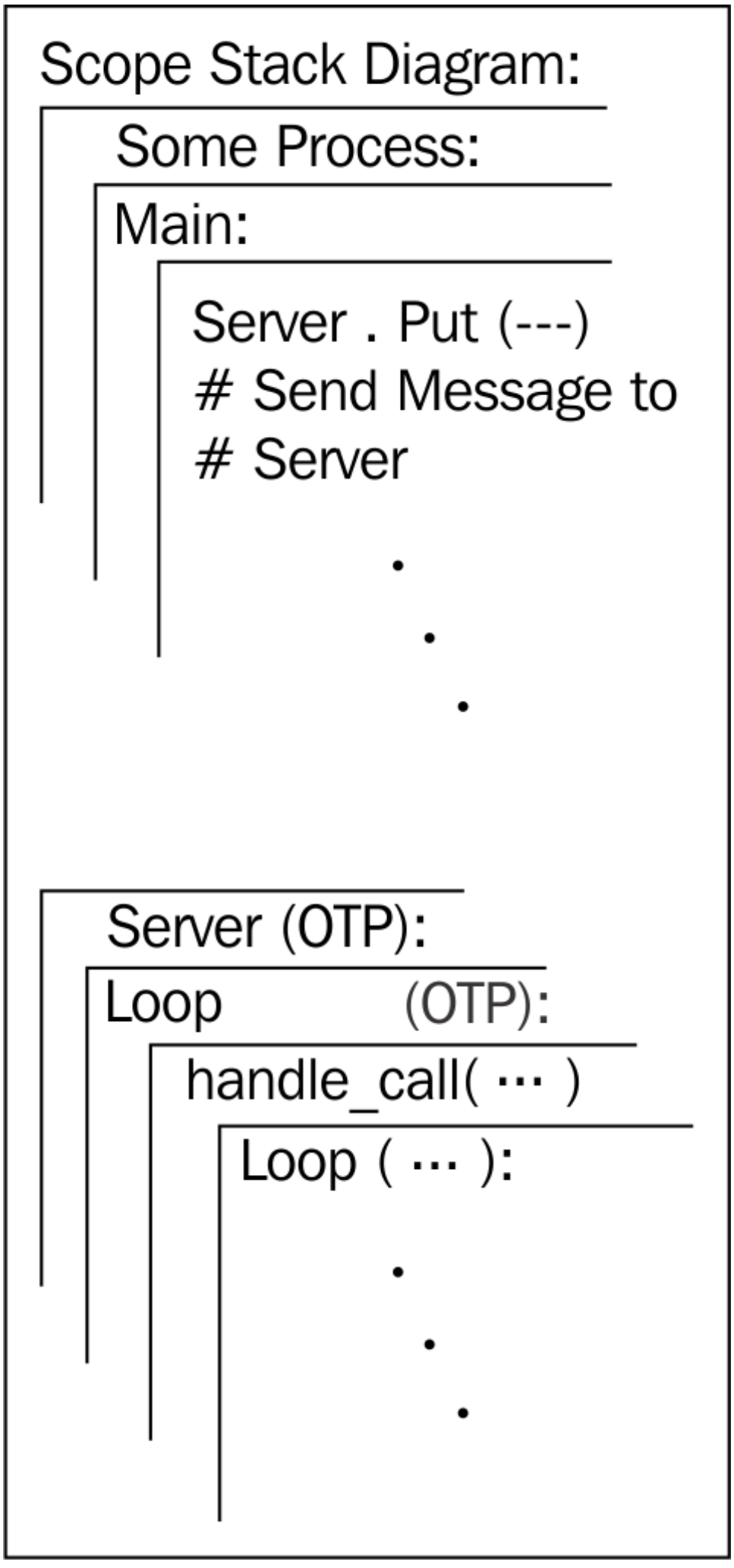
We are only adjusting the one `send/2` call, returning the `{:reply, :pong}` tuple. This is what the `receive` block here is expecting.

Now, loading up and starting the `PingPong` process, we should be able to simply call `ping` and receive the response as a return value:

```
1. iex(3)> PingPong.ping
2. :pong
```

Variable scope in the Gen* processes

When we create helper functions for our `Gen*` processes, we need to keep in mind the scope that these functions are running. Recall the key-value process from earlier. When we added the helper functions `put/2` and `get/1`, these functions are running under the context of the calling process, not the server process that will handle the message:


Scope Stack Diagram:
Some Process:
Main:
Server . Put (---)
Send Message to
Server
Server (OTP):
Loop (OTP):
handle_call(···)
Loop (···):

Scope stack diagram This means that the helper functions themselves will never have access to the current state of the server, and thus, will not be able to do any extra logic with that respect.

Being in the calling scope is how we can, sometimes implicitly, grab the calling process identifier, or other known to exist variables. However, the helper functions should request the most (everything except calls to `self/0`) must-have variables as parameters instead of referencing the variables directly.

Back-pressure and load shedding

Throughout this chapter, we discussed many systems that used synchronous and asynchronous means of transport. The `GenServer` has `call` versus `cast`, and the `GenEvent` has `sync_notify` versus `notify`. In developing applications, it will be tempting to always use the asynchronous versions by default because they are faster, which can be true. Writing files to disk or writing data to the network `can` be very expensive; forcing upstream processes to wait for this process means clients or users may also experience that very same wait.

However, I don't support this temptation. Use the synchronous forms first and profile your system and processes. Then, if it is warranted, decide to use asynchronous methods. By using the synchronous variants, there is inherent back-pressure built into the system. This actually tends to be a good thing because it can keep the system from crashing by applying pressure on the source of input. Doing nothing tends to cause more catastrophic failures, queues spilling, and system resource contention; the system will apply its own back-pressure in the form of completely dropping everything.

Another approach is to use asynchronous behaviours, but have a mechanism to flip the API functions to synchronous behaviours when there are too many messages being sent. This is the exact approach used in Elixir's built-in `Logging` module.

Building back-pressure or load shedding into a system will be a huge benefit to building a truly scalable application. Without it, limits will be found the hard way, users will be angry, stock holders will be equally upset, and business can be lost. Choosing either back-pressure or load balancing (or both) will help mitigate these issues. There will still be upset users, but the number of upset users will be orders of magnitude smaller than **all** of them.

Furthermore, using good end-to-end design and idempotent APIs will largely hide the issues and failures of back-pressure and load-shedding from the users and clients.

Quiz Time!

Quiz 8 | 2 Questions

Start Quiz Skip Quiz >

Question 1:

Identify the generic behaviours defined by OTP from the following options:

- `:gen_fsm`
- `GenHandler`
- `Gen_Server`
- `:gen_server`

Question 2:

Which of the following modules are examples of `GenEvent` process?

- [ ] `Handler`
- [ ] `Logging`
- [ ] `Server`
- [ ] `Callback`