Extracted from:

Programmer Passport: OTP

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

grox.io
Learning

Programmer Passport

OTP

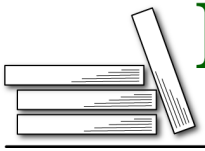
Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: OTP

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: Vanya Wong
Founders: Andy Hunt and Dave Thomas

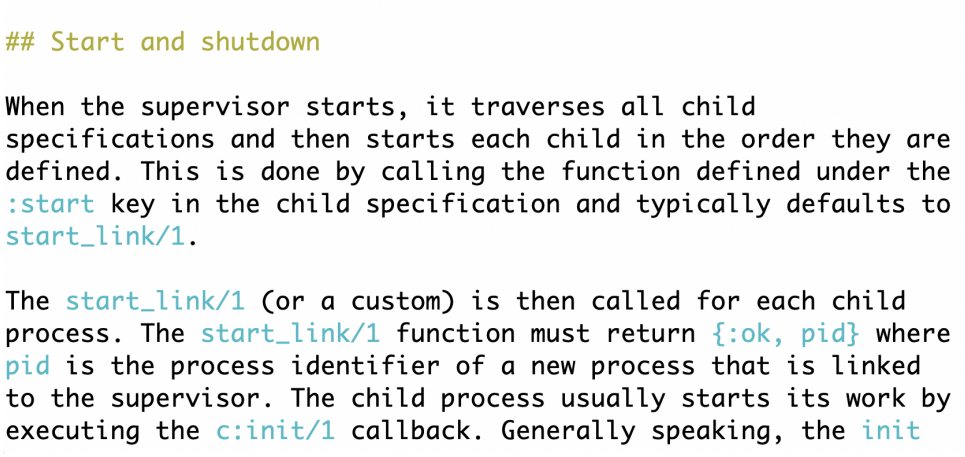
For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-968-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2022

As we explore OTP together, we've been slowly working through the API. First, we built a tiny calculator service *without* OTP. In the last chapter, we focused on the *GenServer* API, and the communication between processes with messages, calls, and casts. In this chapter, we're going to shift gears to the second major part of OTP, the *supervisor*. You may be asking yourself, "Why name this chapter after lifecycles if we're describing a supervisor?"

Let's answer this question in a roundabout way. Open up an IEx session. Next, type `h Supervisor`, and look at the names of concepts in the API. Your system may vary, but mine has headings in gold and API names emphasized in blue, as in the following figure.



```
## Start and shutdown

When the supervisor starts, it traverses all child
specifications and then starts each child in the order they are
defined. This is done by calling the function defined under the
:start key in the child specification and typically defaults to
start_link/1.

The start_link/1 (or a custom) is then called for each child
process. The start_link/1 function must return {:ok, pid} where
pid is the process identifier of a new process that is linked
to the supervisor. The child process usually starts its work by
executing the c:init/1 callback. Generally speaking, the init
```

A quick browse will show you all you need to know. The headings in the documentation tell the story. The heading shown in the figure is "Start and shutdown". That one clearly has lifecycle terms, but other major headings are too. Among them are the following:

- Shutdown
- Child spec
- start_link/2, init/2, and strategies
- Exit reasons and restarts

Also, look at the blue words. On one page, you might find start, init, and start_link. On other pages, you might find terminate, kill, and shutdown. There are terms for children, restarts, and policies describing those things. If you want to understand OTP, think of a supervisor as a *process server* that manages a list of processes we call *children*. That term is yet another lifecycle word.

In this chapter, we're going to build a mix project from scratch. We'll dig into the tools you need to build your own supervisor, and we'll plug in some children.

Along the way, notice that everything we do is related to a few main concepts. A supervisor must *start* a child, and *shut down* children. Supervisors also *detect and respond to failure*. There are plenty of knobs and levers you can manipulate to control this process, but in the end, you'll find that everything we do boils down to these basic ideas.

As usual, the best way to understand what's happening is to dig into some code, so let's get busy!

The Primitive Mechanisms

We'll build our intuition for what's happening within a supervisor by playing with the underlying primitives it's built on. Let's begin our exploration inside IEx, before moving into our own program. First, we'll create some unreliable code:

```
iex(1)> problem = fn -> raise "oh snap" end
#Function<20.128620087/0 in :erl_eval.expr/5>
iex(2)> problem.()
** (RuntimeError) oh snap
```

This code is reliably unreliable, which is perfect for our purposes. Let's simulate a failure by firing up problem in its own process:

```
iex(2)> spawn problem
#PID<0.107.0>
iex(3)>
11:11:48.957 [error] Process #PID<0.107.0> raised an exception
** (RuntimeError) oh snap
    (stdlib) erl_eval.erl:678: :erl_eval.do_apply/6
```

That code did exactly what we expect. The process failed and we saw the results in the IEx console. Now, we can simulate a failure whenever we need one.

Linked Processes Maintain Consistency

Now, let's do the same thing, but we'll *link* the spawned process to our own:

```
iex(4)> spawn_link problem
** (EXIT from #PID<0.103.0>) shell process exited with reason:
an exception was raised:
    ** (RuntimeError) oh snap
        (stdlib) erl_eval.erl:678: :erl_eval.do_apply/6
```

```
11:12:03.188 [error] Process #PID<0.110.0> raised an exception
** (RuntimeError) oh snap
    (stdlib) erl_eval.erl:678: :erl_eval.do_apply/6

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

The function spawn_link starts a process, and links the new process to the one that spawns it. When our unreliable process fails, the IEx console also crashes because it's linked! Sometimes, linking processes in this way helps us preserve consistency by letting us bring down two related processes at once in the event of a failure.

Notice the line number for IEx. It crashed and restarted! What's happening under the covers is that IEx is actually running in OTP. It has a supervisor that detects failure. When the supervisor sees that our IEx session has crashed, it diligently restarts IEx.

That means our supervisor can't be using spawn_link. It's actually using another version of spawn, called spawn_monitor.

Spawn with Monitor Allows Control

Sometimes one process wants to know about the fate of others. Elixir uses Process.monitor/1 for that purpose. The spawn_monitor is a convenience method that lets us spawn and monitor a process at the same time. Let's use it now:

```
iex(1)> problem = fn -> raise "oh snap" end
#Function<20.128620087/0 in :erl_eval.expr/5>
iex(2)> {pid, ref} = spawn_monitor problem
{#PID<0.127.0>, #Reference<0.4055542344.2876506120.76641>}
```

Since our IEx session crashed, we need to create the problem code again. Then we create a monitored process. Notice we get a tuple back. The first element of the tuple is a process ID. The second is a reference. Elixir references are globally unique, and this one will uniquely identify our process when Elixir returns an error.

Now, we can see that our process is no longer alive:

```
iex(3)> Process.alive? pid
false
```

We also got notified that the process is down! There's a message waiting for us in the process mailbox. Let's get it:

```
iex(4)> receive do m -> m end
{:DOWN, #Reference<0.4055542344.2876506120.76641>, :process,
 #PID<0.127.0>,
```

```
{%RuntimeError{message: "oh snap"},
 [{:erl_eval, :do_apply, 6, [file: 'erl_eval.erl', line: 678]}]}}
```

That's the message! We get back a tuple describing the crash. You can read more about monitors and the resulting tuples in the hex monitor documentation[1].

This is the mechanism that supervisors use to detect failure. With firmly established knowledge for what's happening under the hood, let's move on to a project that uses OTP to do the hard work of managing the lifecycles of our programs.

OTP Supervisors Manage GenServer Lifecycles

Let's create a new project, one with a supervisor. We'll call this app super_duper:

```
[otp] ➔ mix new super_duper --sup
...
* creating lib/super_duper/application.ex
...

[otp] ➔ cd super_duper/
```

Notice the list of files mix created for you. One of them is application.ex. That's your supervisor.

Application is a Supervisor Template

Just as the GenServer module is a template for a generic server, the Application module is a template for a supervisor. The documentation says, "Applications are the idiomatic way to package software in Erlang/OTP." When you start an application, you're really starting a *supervisor*, and that supervisor is starting the GenServers that make up the rest of your code base.

Your application might have other projects it depends on, and you can start these within this SuperDuper.Application module.

This is what it looks like, without the comments:

```
defmodule SuperDuper.Application do
  use Application

  def start(_type, _args) do
    children = []

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

1. https://hexdocs.pm/elixir/Process.html#monitor/1

```
end
```

We get the usual `use Application` ceremony. That command executes `Application.__using__`, which establishes `SuperDuper.Application` as a module that implements the `Application` behaviour. As you might expect, this behaviour has various callback functions[2] for starting and stopping applications.

The main callback is `start`. Ours establishes an empty list of children. This is where we'll add dependent services later on. Then, we start the server with `Supervisor.start`, passing our children and options including a name and a policy for restarting the children in our list. We'll talk about these policies later.

2. https://hexdocs.pm/elixir/Application.html#callbacks