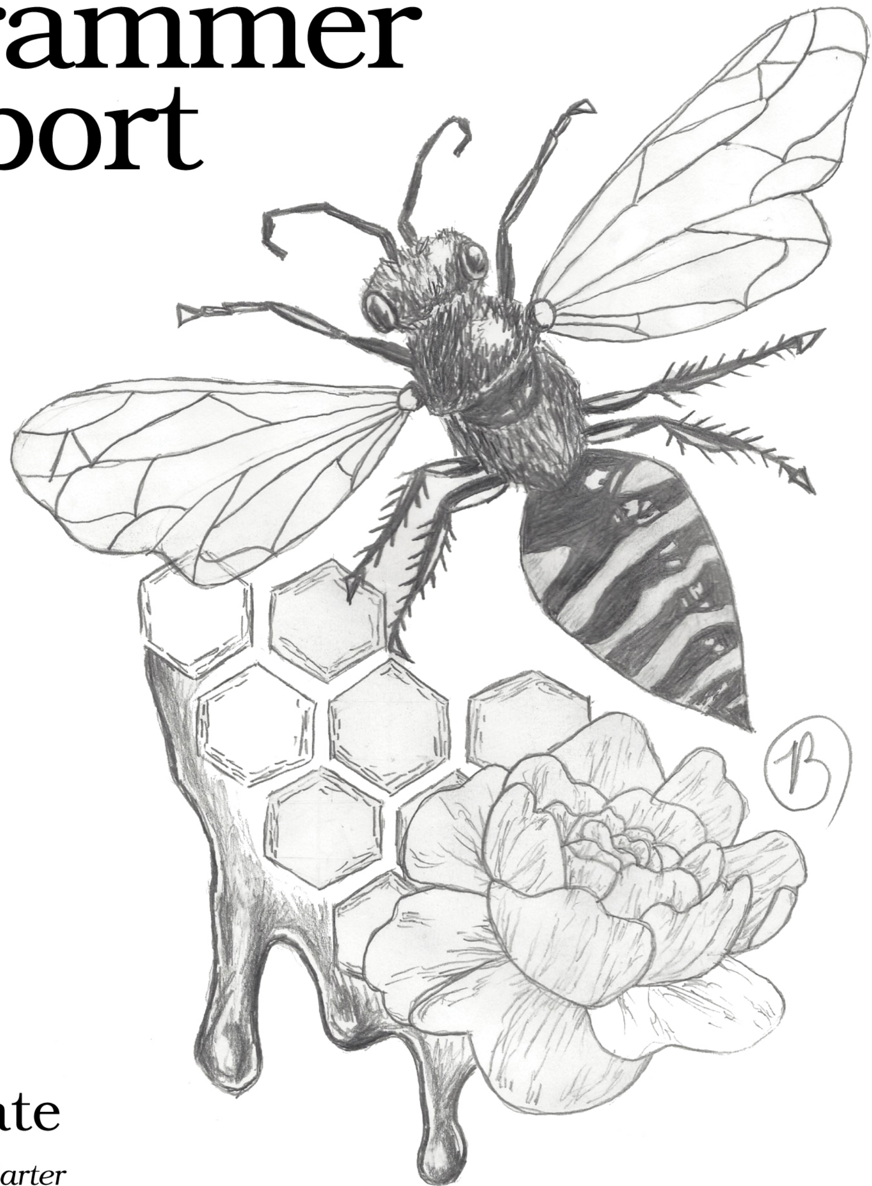
The Pragmatic Programmers

grox.io Learning

Programmer Passport

OTP



Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: OTP

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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*.

Copyright © 2022 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Contents

Preface . . . v

1. A Basic Handmade Server . . . 1
   - Build a Server with a Process 2
   - Build the Boundary Layer 7
   - Build an OTP Server, with Mix 12
   - Your Turn 15

2. Communication Between Servers . . . 17
   - Anatomy of a GenServer 17
   - handle_info Processes Nonstandard Messages 20
   - Schedule an Alarm with handle_cast 25
   - Implement a Status Message with handle_call 29
   - Your Turn 32

3. The Lifecycle and Supervision . . . 35
   - The Primitive Mechanisms 36
   - OTP Supervisors Manage GenServer Lifecycles 38
   - Add Some Children 42
   - Lifecycle Policy 47
   - Your Turn 52

4. The Power of a Name . . . 55
   - Add Dynamic Characters to SuperDuper 55
   - Dynamic Children 57
   - Dynamic Supervisors 63
   - The Process Registry: The Power of a Name 65
   - Your Turn 66

Bibliography . . . 69

Preface

OTP is one of the most consequential inventions in the entire Erlang and Elixir ecosystems. The library provides a common framework for reliably starting, stopping, and running concurrent workers. OTP is the reason many developers learn Erlang or Elixir. Elixir's wonderful management characteristics and seamless concurrency flow out of OTP. If you're looking to take the step from beginner to intermediate Elixir developer, OTP would be a great place to start.

With so much attention on OTP, you might wonder if the world needs yet another OTP book. The wonderful *Designing Elixir Systems with OTP [IT19]* I wrote with James E Gray II and the brilliant *Elixir in Action [Jur15]* by Saša Jurić provide excellent insight into how to design Elixir with OTP. If you're using OTP in conjunction with Phoenix, Lance Halvorsen's treatment in *Functional Web Development with Elixir, OTP, and Phoenix [Hal18]* might be just the ticket.

Still, this short book fills a nice niche. OTP is a behaviour, a program written by someone else. The only way to understand OTP is to know what the behaviour is doing under the surface. After teaching OTP for four years, I've learned that many developers don't understand the underlying behaviour, so they need more than the documentation. At the same time, they don't always have the time or attention for a longer text. This small companion to Groxio's OTP course[1] presents a no-frills treatment for those needing more than stock documentation or blog posts, without reading a more expansive OTP treatment.

If you find yourself in this niche, I'd love to join you in the journey. If you like what you see, who knows? You might decide to join us on Groxio for some video demonstrations of OTP and other Elixir concepts!

1. https://grox.io

CHAPTER 1

A Basic Handmade Server

For dozens of years, the Erlang language has achieved extraordinary reliability. Now, Elixir programmers expect the same. Elixir and Erlang systems can support applications with even hundreds of thousands of processes without skipping a beat. The actor-based message passing architecture is now common across many programming languages. Perhaps the most important feature in either of these languages is OTP.

OTP is a set of libraries and APIs that enable applications with extraordinary reliability and scalability through concurrency. In the four OTP chapters that follow, we'll cover the main two main abstractions for OTP, the boundary and lifecycle layers.

The boundary layers allow Elixir to share state across processes with a combination of concurrency, message passing, and recursion called *GenServers*, short for generic servers. They also allow processes to communicate and control the impact of failures. All of this may seem like a mystery at first, but don't worry. We'll walk you through a good example.

The lifecycle layers are responsible for starting and stopping processes with *supervisors*. These supervisors can detect when a process crashes and execute a policy for responding to failure, perhaps by starting a new process.

This combination of GenServers and supervisors has brilliantly withstood the tests of time. Failures in any part of the system are transient because any but the most catastrophic failures are quickly remedied with a restart.

This book is not designed to replace *Designing Elixir Systems with OTP [IT19]* by James Edward Gray II and myself. That book focuses on design philosophies and considerations for complex Elixir systems, which often include OTP. Instead, it's a companion. Instead of focusing on a single OTP application using Elixir, this one will focus on the basic understanding of the OTP API.

Instead of working with one primary project, we'll solve several smaller problems. Since we won't have to handle deep design questions throughout the release, we'll be able to spend more time on those aspects of OTP that are not as commonly approached. Hopefully, when we're done, you'll have a better understanding of the various knobs and levers that GenServer provides, and where you might use them.

In this first chapter, we're going to build a calculator application without OTP. Along the way, we'll teach you the OTP terminology for the features we're building. We'll use native processes and message passing rather than the OTP library. Then we'll wrap the application in an API, and show how we might implement that API using OTP instead.

At its basic level, a `GenServer` is a running process that sends and receives messages. It has several standardized functions called *callbacks* that communicate with your application so you can customize the process with your own code. Since OTP programs *implement only these callbacks*, beginners sometimes have a tough time understanding what a whole GenServer looks like. We're going to remedy that problem first by building an app without using the GenServer API. Then, we'll make a few tiny tweaks to replace our process machinery with a GenServer.

Rather than walk through a lot of theory, let's write a few dozen lines of simple code that describe how you might build a server—a process that sends and receives messages—without OTP. That will give you a good sense of how an OTP service is built. Then, we'll convert that service to use OTP.

Let's get started!

Build a Server with a Process

In this chapter, we're going to build a calculator service. First, we're going to build the core of our service, the piece that does the actual computation. Then we're going to wrap our core layer in a boundary layer that will hold the calculator value as we add calculations, one at a time. Though we haven't formally mixed in OTP yet, this boundary layer is the realm of the OTP GenServer. It will have a message loop, a means to start, and a means to send messages.

The main building block of OTP is the GenServer. These tiny services are not necessarily network servers. Instead, they are *functions* running in a *process* in a recursive message loop. Rather than explain things with words, let's show some rough code that does the job.

First, we need a recursive loop, one with potentially changing state. We'll capture the state of our server in the execution of a recursive loop:

```
def run(old_state) do

  new_state = ???(old_state)

  run(new_state)
end
```

A small but important bit of our program is undefined. It's the part that actually does the work. When you think about the ???(state) part of the program as a function, we can express that program more simply, like this:

```
def run(state) do
  state
  |> Core.do_work
  |> run
end
```

The two previous listings are the same. This recursive function is really the heart of an OTP GenServer. The function takes some state, transforms it in some way using our core, and then calls itself with the transformed state. As you might expect, this recipe is a bit of an oversimplification. It needs a few additional ingredients, especially some way to communicate with other programs. We're going to wrap this tiny function in a process. To interact with other processes, our run function needs to add some way to send and receive messages. Let's call it listen, and add some process machinery to start our server, like this:

```
def start(initial_state) do
  spawn fn -> run(initial_state) end
end

def run(state) do
  state
  |> listen
  |> run
end

def listen(state) do
  receive do
    :some_message -> Core.do_work(state)
    :another_message -> Core.do_other_work(state)
  end
end
```

The above listing is a more complete look at our overall plan. We start a process with spawn. Then we start with the state, and call listen(state) to do a bit of

work from our core layer, returning the new state. Finally, we recursively call `run` again with the new state.

That's our rough plan. When all is said and done, we'll have:

- A piece of *state* data
- A *core* layer to do the work of our service by transforming our state
- A *start* link to start our service
- A *message loop* that recursively calls itself with transformed state
- A *listen* function to respond to messages on the server

Eventually, our OTP GenServers will also have these pieces. For now, let's follow this template by building a project using primitives rather than OTP.

Create a project with `mix new calculator`, and we'll get to work. Let's start with the part of our program that does the work, the functional core.

Build a Core

Functional cores do the work of GenServers. The functional core should work on data that's validated and safe. It should be predictable, so it avoids side effects.[1] We'll hit the highlights throughout this book. If you are looking for deeper design considerations for building a functional core, check out *Designing Elixir Systems with OTP [IT19]* for more details.

Add the following functions, which make up our core, to `lib/core.ex`:

```
defmodule Calculator.Core do
  def add(acc, number), do: acc + number
  def subtract(acc, number), do: acc - number
  def multiply(acc, number), do: acc * number
  def divide(acc, number), do: acc / number

  def inc(acc), do: acc + 1
  def dec(acc), do: acc - 1

  def fold(list, acc, f) do
    Enum.reduce(list, acc, fn item, acc -> f.(acc, item) end)
  end
end
```

Each of the functions in our core will work on a *state* that consists of a number. Each function will take a number as an argument, perform calculations on that number as a handheld calculator would, and return the resulting number. We have a final function, one called `fold`, that isn't part of our API.

1. https://lispcast.com/what-are-side-effects/

Organizing functions in a predictable core makes testing substantially easier. We'll write a tiny set of test cases here, but it's also important to point out that other more exhaustive forms of testing such as property-based testing are far easier within a functional core. If you want to know more about property-based testing, check out *Property-Based Testing with PropEr, Erlang, and Elixir [Heb19]*.

These tests can stay simple because they don't have to deal with complex testing issues such as processes, unpredictable results, or impure data:

```
defmodule CoreTest do
  use ExUnit.Case
  import Calculator.Core

  test "subtracts" do
    assert subtract(10, 4) == 6
  end

  test "adds" do
    assert add(10, 4) == 14
  end

  test "multiplies" do
    assert multiply(10, 4) == 40
  end

  test "divides" do
    assert divide(10, 2) == 5.0
  end

  test "fold" do
    assert fold([1, 2, 3, 4], 0, &add/2) == 10
  end
end
```

This list of tests isn't complete, but you get the idea. Rather than linger with our tests too long, let's look at the main functions that make up our core in a little more detail.

Reducers do the Heavy Lifting

The main functions in our core, those that do the bulk of the work, are *reducers*. These functions have a first argument of some type and return data of that same type. Take, for example, `&add/2`. The first argument is a number, and it returns a number.

Let's look at why those functions are called reducers. Another name for *reduce* in functional languages is *fold*. Add this `fold` function to `lib/core.ex`, like this:

```
def fold(list, acc, f), do: Enum.reduce(list, acc, &(f.(&2, &1)))
```

Notice that we're just calling the Enum.reduce function underneath. The only difference is that we flip the two arguments in the reducer, called f. Said another way, fold *is* an implementation of reduce with a slightly different API.

The main functions in our core, add, subtract, multiply, and divide are all reducers. These functions work with Enum.reduce, but that explanation is a bit vague. Let's explore reducers in IEx:

```
iex> import Calculator.Core
Calculator.Core
iex> list = [1, 2, 3]
[1, 2, 3]
iex> acc = 10
10
iex> 10 |> subtract(1) |> subtract(2) |> subtract(3)
4
```

We take a list of numbers, [1, 2, 3]. They all work with our reducers. The last piped expression in the previous example is exactly what happens in a reduce: we start with an accumulator and pipe each number in our list through the reducers.

Now, let's execute the same function using our fold:

```
iex> fold(list, acc, &subtract/2)
4
```

So when we say that our reducers work with Enum.reduce/3, that's not strictly true. Our reducers specify the accumulator *first* instead of *second*.

You might notice that the Enum.reduce and our fold take reducers with exactly two arguments. Strictly speaking, reducers don't have to take two arguments. For example, we can run inc/1 through our reducers by ignoring the second reducer argument, like this:

```
iex> 10 |> inc |> inc |> inc
13
iex> fold([1, 2, 3], 10, fn acc, _ -> inc(acc) end)
13
```

We fold over a list. The list could actually contain any data; the result would be the same. We ignore each item in the list and call inc on the accumulator each time.

If you think of our service as a robot, these reducers form the CPU, or the brain. We'll add in the machinery to do the rest of the work in a handmade boundary layer.

Build the Boundary Layer

A core by itself is a *library*. Boundaries, whether we build them ourselves or with a GenServer, are process machinery. Remember, our boundary will spawn a recursive function called run that has a listen function to interact with other processes.

Since the concerns of the boundary are different from the concerns of the core, we'll put the code in a separate module. Open up the file lib/boundary.ex and we'll get to work.

Establish the Boundary Module

Now, let's establish our boundary. Our boundary layer will use our core functions in a message loop. The boundary will track state over time using recursion and message passing. The boundary will use our core to transform the state. If this all seems confusing, just key in the code below into boundary.ex. It will eventually make sense:

```
defmodule Calculator.Boundary do
  alias Calculator.Core
end
```

Let's work from the inside out. We'll first listen for requests to do work. We'll receive messages that do the work for each of our main reducer functions, like this:

```
def listen(state) do
  receive do
    {:add, number} ->
      Core.add(state, number)

    {:subtract, number} ->
      Core.subtract(state, number)

    {:multiply, number} ->
      Core.multiply(state, number)

    {:divide, number} ->
      Core.divide(state, number)
  end
end
```

Each piece of code receives a message and modifies the state within our functional core. We take messages in tuple form so clients can provide both the command and the arguments for the command. We've cleanly separated the concerns of the boundary from the concerns of the core.

These messages are fire-and-forget asynchronous messages. Clients won't have to wait for a response. In GenServer speak, these messages are *casts*.

Before we move on, let's add a way to clear the calculator and a way to return state. Add these two messages before the `end` statement for the `receive do` expression:

```
:clear ->
  0

{:state, pid} ->
  send(pid, {:state, state})
  state
```

The `clear` command is dead simple. It simply returns `0` for the state. Technically speaking, it doesn't use the core. It simply resets the state.

The `:state` message is a little more complicated. This message contains the caller's process id so it can send the state back to the client. We send a `{:state, state}` tuple with an atom and the state back to the client to provide some assurance that the client is receiving the correct state. Then, we return the original `state`, since a query to get the `:state` should not change its value.

Next, we'll provide the `run` loop, like this:

```
def run(state) do
  state
  |> listen
  |> run
end
```

Lovely! It's just like our template. All that remains is a function to start the process:

```
def start(initial_state) do
  spawn(fn -> run(initial_state) end)
end
```

And we're off to the races! We now have a working service. Let's take it for a test drive.

Use the Server with send

We can use our service, but we're going to have to interact with it using process primitives of `send` and `receive`. Open up an IEx console with `iex -S mix`, or recompile your existing IEx session if it's already open with `recompile`. Let's play with our service.

```
iex> import Calculator.Boundary
Calculator.Boundary
```

```
iex> pid = start 0
#PID<0.215.0>
```

We now have a started server. We can verify that it's alive, like this:

```
iex> Process.alive? pid
true
```

It's running! Let's get the state.

```
iex> send pid, {:state, self()}
{:state, #PID<0.140.0>}
iex> flush
{:state, 0}
:ok
```

We use flush to see the results in our mailbox. We get a state of 0 back, so it's working! Now, we can interact with our calculator by sending messages that use the reducers in our core to change the state of the calculator:

```
iex> send pid, {:add, 10}
{:add, 10}
iex> send pid, {:add, 20}
{:add, 20}
iex> send pid, {:subtract, 40}
{:subtract, 40}
iex> send pid, {:state, self()}
{:state, #PID<0.140.0>}
iex> flush
{:state, -10}
:ok
```

We interact with the calculator server. We add 10, add 20, and subtract 40 and then get a state that looks like my college bank balance. The interaction is ugly, though. Using this API is ugly and error prone. There's a better way. Let's add an API layer.

Establish an API

We've coded the core with functions to do the work for each service. We've added the boundary layer that handles process machinery. Now, it's time to code the API. It's the API's job to present a common, convenient interface to other programs, whether they are part of the same application or a remote one.

The Calculator module is the right place for the API. It's the module with the flattest namespace, and the one that should contain documentation about our API. Instead of forcing our users to use process primitives, our API will consist of functions.

Open up lib/calculator.ex so we can get started:

```
defmodule Calculator do
  alias Calculator.Boundary

end
```

We'll start from scratch. We clear out the ceremony from the original mix project. While you're at it, clean out the test, too. Now to start the process:

```
def start(initial_state) do
  Boundary.start(initial_state)
end
```

We call the API to create our process. This function is a *constructor* since it creates the process. Our function returns a pid. If we ever wanted to harden this service, we'd want to return an {:ok, pid} tuple to allow for errors. For our trivial illustration, our simple service will suffice.

Next, we'll build an API version for each of our service's messages. Let's start with the five asynchronous calls:

```
def add(calculator, n), do: send(calculator, {:add, n})
def subtract(calculator, n), do: send(calculator, {:subtract, n})
def multiply(calculator, n), do: send(calculator, {:multiply, n})
def divide(calculator, n), do: send(calculator, {:divide, n})

def clear(calculator), do: send(calculator, :clear)
```

Remember, Elixir's modules should have functions where the first argument is the data type for our module. Since this file wraps with a backend server, the type we'll use is the pid. We can be more descriptive, though. Our process IDs represent processes that wrap our calculator services. We'll name them calculators.

These will become GenServer casts, asynchronous fire-and-forget messages. Let's look at the synchronous counterpart, those with a call and response:

```
def state(calculator) do
  send(calculator, {:state, self()})
  receive do
    {:state, state} ->
      state
  after
    5000 ->
      {:error, :timeout}
  end
end
```

We send the state message, but we also need to receive the response. That means our message also needs the pid for our process, self().

Now that we have an API, let's take it for a spin.

Use the Server through Our API

We'll exercise our GenServer through an API layer. To see how that layer might work, let's go back to IEx. Remember to `recompile` if you haven't already done so.

```
iex> calc = Calculator.start 10
#PID<0.260.0>
iex> Calculator.add calc, 1
{:add, 1}
iex> Calculator.add calc, 5
{:add, 5}
iex> Calculator.state calc
16
```

As expected, it's working perfectly! All is not well, though. Our calculator has problems:

```
iex> Calculator.add calc, :this_will_crash
{:add, :this_will_crash}
iex>
14:09:25.004 [error] Process #PID<0.139.0> raised an exception
** (ArithmeticError) bad argument in arithmetic expression
    :erlang.+(1, :this_will_crash)
    (calc) lib/core.ex:2: Calculator.Core.add/2
    (calc) lib/boundary.ex:10: Calculator.Boundary.run/1

nil
iex> Process.alive? calc
false
```

We crashed our server, and that shows two flaws in our design. The first is a boundary concern. Our boundary doesn't adequately validate our data, and our core doesn't adequately guard against incorrect data. These are generic *programming concerns*.

The second problem is more serious. It's an infrastructure problem. We need to be able to detect failure and take action. These are *lifecycle* concerns.

We need a *process server*, one that we can integrate into our calculator or any other project we might build. This process server should know how to *start up* a service, *shut it down* cleanly, and *detect* when services shut down.

We need OTP.

Build an OTP Server, with Mix

Elixir's OTP has the features we've built so far from the boundary on out called GenServers. OTP also has a way to build and configure *process servers* called supervisors. We'll save the supervisors for later in this series and focus on the GenServers.

Let's build a new boundary. Instead of building our process machinery from scratch, we'll use the OTP library instead.

Build the OTP Boundary

Before we use the GenServer library, let's think about the parts of our program that we might need to customize. Let's bring back the boundary layer, which looks like this:

```
defmodule Calculator.Boundary do
  alias Calculator.Core

  def start(initial_state) do
    spawn(fn -> run(initial_state) end)  # <--- init
  end

  def run(state) do
    state
    |> listen
    |> run
  end

  def listen(state) do
    receive do
      {:add, number} ->
        Core.add(state, number)          # handle_cast

      {:subtract, number} ->
        Core.subtract(state, number)     # handle_cast

      {:multiply, number} ->
        Core.multiply(state, number)      # handle_cast

      {:divide, number} ->
        Core.divide(state, number)        # handle_cast

      {:state, pid} ->
        send(pid, {:state, state})        # handle_call
        state

    end
  end
end
```

These comments point to the lines of code that you'll be writing yourself. We'll need to specify our own starting callback using init. We'll also need a callback

for each of the custom messages our server supports. The handle_cast callback will implement one-way asynchronous functions, and the handle_call callback will support two-way synchronous ones. You can remember these because phone CALLS are two way, and podCASTS are one way.

Here's what our program looks like, piece by piece:

```
defmodule Calculator.Server do
  use GenServer
  alias Calculator.Core

  def start_link(initial) when is_integer(initial) do
    GenServer.start_link(__MODULE__, initial)
  end

  def init(number) do
    {:ok, number}
  end
end
```

We start with the use GenServer, which announces our intention to use the macros in this API. We will keep as much of our custom code as possible within our core, so we alias that. We also provide a start_link to spawn our process. Don't worry about the details just yet. Simply understand that start_link starts a process, linked back to this one, so that Elixir can restart the process in the event of failure.

We provide a name (the module for our program) and an initial value to start_link. Elixir will store our pid in a registry under the name Calculator.Server in case the process crashes and we need to start a new one.

Now, let's add the callbacks:

```
def handle_cast({:add, number}, state) do
  {:noreply, Core.add(state, number)}
end
def handle_cast({:subtract, number}, state) do
  {:noreply, Core.subtract(state, number)}
end
def handle_cast({:multiply, number}, state) do
  {:noreply, Core.multiply(state, number)}
end
def handle_cast({:divide, number}, state) do
  {:noreply, Core.divide(state, number)}
end
def handle_cast(:clear, _state) do
  {:noreply, 0}
end

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

```
end
```

The arguments may not make complete sense, but you can see the general pattern. Each function is a callback that implements a single clause of the receive argument. The handle_cast callbacks don't need to reply, because they're asynchronous, so they respond with a :noreply tuple. The handle_call function needs to specify a :reply tuple. The second tuple element goes to the client and the third goes back to the recursive call in the server.

All that remains is to define an API. Within OTP programs, it's customary to build an API layer into the GenServer, so add these lines to lib/server.ex, like this:

```
def add(pid, number), do: GenServer.cast(pid, {:add, number})
def subtract(pid, number), do: GenServer.cast(pid, {:subtract, number})
def multiply(pid, number), do: GenServer.cast(pid, {:multiply, number})
def divide(pid, number), do: GenServer.cast(pid, {:divide, number})
def clear(pid), do: GenServer.cast(pid, :clear)

def state(pid) do
  GenServer.call(pid, :state)
end
```

These functions use GenServer calls to do calls and casts rather than using native send and receive functions. That's all there is to it!

Let's take it for a spin:

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> alias Calculator.Server
Calculator.Server
iex> {:ok, server} = Server.start_link(0)
{:ok, #PID<0.228.0>}
iex> Server.add server, 10
:ok
iex> Server.state server
10
iex> Server.subtract server, 2
:ok
iex> Server.state server
8
```

It works! In the following chapters, you'll see us use this GenServer to automatically restore a crashing server. We'll leave it to you to decide whether to decide which Calculator to integrate.

Now, it's a good time to wrap up.

Your Turn

We started this chapter with a little basic background, and then we went right into how you might build something *without* OTP first. We just went through a whole long chapter, building a service from scratch that we could have built using OTP. We used the same techniques you might use in other distributed, functional languages. We used processes and a message loop to manage state. By now, you may realize why.

Layers of an OTP App

OTP is like a *template* for an application. Your code fills in the template by using *callbacks*. We wrote the code for a calculator in layers, with a core, a boundary, and an API. Once we finished our calculator, we replaced the boundary with an OTP server.

OTP is a framework for building concurrent, reliable services. You can absolutely build the services in OTP by hand using Elixir's primitives, but you shouldn't. Instead, you should rely on an OTP foundation to take advantage of years of experience.

You'll build your OTP servers application in layers. The *functional core* has the functions that perform each of your services. The *boundary* has process machinery, validations, and the like. The *api* layer wraps up the boundary layer with tiny functions that present the service to your user.

The best way to understand services is to build your own or add on to ours. These exercises will help.

Try It Yourself

In this section, we'll introduce a few simple problems with OTP. Build each of the following services with OTP and GenServer.

- Add a `negate` command to the calculator. It should multiply the calculator value by -1.
- Implement `inc` as a `handle_info` callback instead of a `handle_cast`.

These *medium* problems have you building your own OTP server. Each of your services should have a functional core, a boundary, and an API layer.

- Build a *stack* server. You can find a good example of a service in the OTP documentation within IEx. From IEx, issue the command `h GenServer`, and you'll see an example of a stack application. You'll need to extract the `push`

and `pop` features into a functional core, and wrap your service in an API. It should support the commands `push`, `pop`, and `state`.

- Build a Counter server. Handle the commands `inc` and `state`.

This *hard* problem needs to take advantage of the `init` callback to start a periodic timer, and calls a function.

- Implement a *clock* server that prints the time every minute. The server should support one message: `tick`, which prints the time every minute. You must also send that message every minute.

Next Time

In the next chapter, we'll dig into the GenServer callbacks. We'll go beyond the generic servers you find in the documentation. In particular, we'll dive into the response tuples and how they work.

CHAPTER 2

Communication Between Servers

Typical libraries are made up of simple functions. You use them by writing calls to functions, and the library returns a result. The GenServer API is a little different. A GenServer reverses the roles. These generic servers are fully functioning servers with a few pieces missing. Your application implements these missing pieces, called *callbacks* so rather than you calling functions on the GenServer API, often *the GenServer calls your functions* instead.

As you might imagine, over the years, GenServers have evolved to handle many different scenarios, so the various knobs and levers you can use to tailor your applications can bewilder even the hardiest developer. Take heart. When you look closely, several patterns emerge. This chapter is dedicated to helping you understand the communication between a Genserver, your application, and other processes. Let's get started.

Anatomy of a GenServer

In the last chapter, we built a basic calculator, with and without a GenServer. Let's look back at that program. We'll focus on a a couple major messages: add and state:

```
def start(initial_state) do
  spawn(fn -> run(initial_state) end)
end

def run(state) do
  state
  |> listen
  |> run
end

def listen(state) do
  receive do
    {:add, number} ->
      Core.add(state, number)
```

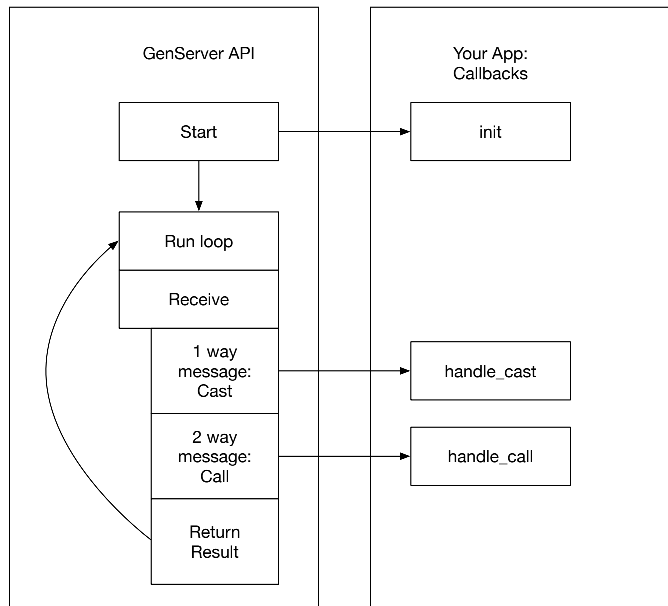
```
    {:state, pid} ->
      send(pid, {:state, state})
      state
  end
end
```

This program has the same basic shape of many other Elixir programs. There are two major parts, the *lifecycle management* and the *message process*. Our lifecycle management is a simple `start` function to start a process. The `run` and `listen` messages represent the message loop.

Our server supports two different kinds of messages. The first kind, the `add` message, simply receives a message and transforms the state. It's a *one-way* message, a simple asyncronous message. It's a *cast* in GenServer terminology. The other kind, the `state` message, is a *two-way* call-response message, a *call* in GenServer terminology.

A GenServer is a Template

You can think of a GenServer as a template for your code. You can fill in the blanks with callbacks, bits of code you'll implement in your own modules as shown in the following figure.



This diagram gives you a good picture of what's happening. When we work with OTP, the `GenServer` library builds the generic lifecycle management and

message loops, leaving the rest to your application. The GenServer calls your application's callback functions at certain specific times.

Notice the structure of the GenServer. It has the same application components as the calculator we built in the last chapter. It implements the lifecycle management by spawning a process. We'll cover the lifecycle in Chapter 3, The Lifecycle and Supervision, on page 35. For now, let's focus on the rest.

The Basic Callbacks

At each possible application integration, OTP will call your app. The most-used callbacks are:

init
: called when a server starts a GenServer

handle_call
: called when a server receives a two-way message

handle_cast
: called when a server receives a one-way message

handle_info
: called when a server process receives a *generic message*, a message not formatted for OTP

This is the general shape your code will have when you're using these callbacks:

```
def init(state) do
  # custom-code-here
  {:ok, initial_server_state}
end

def handle_call(message, from_pid, server_state) do
  # custom-code-here
  {:reply, client_response, new_server_state}
end

def handle_cast(message, server_state) do
  # custom-code-here
  {:noreply, new_server_state}
end

def handle_info(message, server_state) do
  # custom-code-here
  {:noreply, new_server_state}
end
```

You'll use init callback exists to do one or both of these two things:

- Call some code with a side effect before starting your application.
- Transform the inbound `state` into a state that's friendlier for your GenServer.

The `handle_` callbacks are slightly more complicated. Each has its own signature, but there are some common themes:

- the first argument is always the message
- the last argument is always the server's state
- the response is always a standard response tuple

The `handle_info` callback is the simplest. It takes two arguments, the `message` the server is receiving and the state of the server. It usually returns a `:noreply` tuple, one that provides the new state for the GenServer. This state will be passed to the next `handle_` callback.

The `handle_cast` callback works almost exactly like a `handle_info`, but with one major difference. The API gets the pid of the caller in the second argument, the `from` field. Otherwise, it's the same. You provide your custom code and typically return a standard `:noreply` tuple.

The `handle_call` is a two-way synchronous API, so it needs to send a `:reply` tuple. Usually, the `reply` tuple has the atom `:reply`, followed by the message to send to the client, followed by the new state for the GenServer. This new state will flow back into the next `handle_` callback, and the circle of life continues!

This chapter will focus on making the most of those callbacks. You'll learn how Elixir calls them, and how your callbacks should respond. We'll go off the beaten path a bit to explore some of the optional bits of OTP that you might miss if you're not a careful reader. We'll look at response tuples beyond the typical use cases and how to find them.

We're going to start our tour with the simplest message, one that OTP does not create. Let's explore the `handle_info` callback.

handle_info Processes Nonstandard Messages

You've seen a brief introduction of `handle_info`, but now we can fill in some more details. Use the `handle_info` callback to send *generic* Elixir messages to a GenServer. By generic, we mean messages that work with any generic Elixir process, not necessarily a GenServer process.

Elixir uses the actor programming model, meaning each process has its own message queue. You can use process primitives to send messages to any Elixir process, as long as you have a pid.

Sometimes, you may want your GenServer to receive messages from Elixir or Erlang process primitives rather than calls or casts built for OTP. A good example is the Process.send_after/2 function. Let's see how that works.

Send a Timed Message

As you might expect, Elixir and Erlang have several tools for sending messages to any process based on some interval. The tools are easy to use and useful. Let's take a look at a few of them.

```
iex> Process.send_after(self(), :hi, 2000);
receive do m -> m end;
IO.puts("Done!")
Done!
:ok
```

Process is Elixir's module for dealing with any process, including GenServers. The self() function returns the pid for our own process. As is customary, the first argument to functions in Process will represent a process. We chase that argument with the message, :hi, and a duration in milliseconds.

A similar message, :timer.send_interval, sends a message after a specified period of time to a pid, like this:

```
iex(3)> :timer.send_interval(1000, self(), :tick)
{:ok, {:interval, #Reference<0.1649946897.170917896.30200>}}
iex(4)> flush
:tick
:tick
:ok
iex(5)> flush
:tick
:tick
:ok
```

We ask the timer, in another process, to send a message :tick at one-second intervals. Then, we flush the message buffer a couple of times to see what's in the message box. After running this short program, it would be good to exit the console to prevent your mailbox from being flooded with :tick messages!

Both of these tools are interesting to OTP programmers, but neither the :tick nor the :hi message was formatted for OTP. It turns out that handle_info is built especially for retrieving generic messages like these.

The Bones of a Simple Egg Timer

Say we want to build our very own overpowered egg timer using Elixir. We could use OTP to start the timer, and it could send a message to any process that we want when the timer expires.

We'll make use of the send_after message to implement the timer functionality, and we'll use the handle_info callback to receive the message to respond to our timer.

Our functional core will let us create a timer with the beginning time, a duration, a status, and a function to call when it expires. Eventually, we'll store them in our GenServer in a map with a string key.

Type mix new egg_timer. Change into the project directory, and key this into alarm.ex:

```
defmodule EggTimer.Alarm do
  defstruct ~w[duration name time f]a

  def new(name, duration, f \\ &default_fn/0)
      when is_atom(name) and is_integer(duration) and is_function(f) do
    __struct__(
      time: Time.utc_now(),
      name: name,
      duration: duration,
      f: f
    )
  end

  def trigger(alarm) do
    alarm.f.()
    alarm
  end

  def default_fn do
    IO.puts("Alarm triggered!")
  end
end
```

Our functional core is an alarm. We have attributes for a name that we'll use to register an alarm, the current time (this will come in handy later when we want to compute how much time is left on the timer), the total amount of time for the alarm, and a function to call when the alarm triggers.

Keep in mind that while our alarm has the data and functions for managing alarms, it does not know how to send timed messages. Such process machinery belongs in the boundary. Instead of implementing an actual trigger, we use a high-order function to keep our core pure and provide flexibility for

our API. This way, a user can provide a function so any alarm can trigger any activity our users can wrap in a function.

There are only three functions: a constructor to create new alarms, one that provides a function to call when the alarm triggers, and the default function for our alarm. We won't worry about adding alarms to our map of alarms within our core. We'll manage that map in our GenServer.

To kick off our example, let's build the beginnings of our process machinery, code that lives in our boundary. We'll start with a function to start a server.

Build a GenServer

Our initial GenServer will have three pieces. We'll create a start_link to start the server, with an init callback. We'll also include a function to send a message at the prescribed time with Process.send_after. Since that message is generic instead of a GenServer message, we'll need a handle_info callback to process it. Open up server.ex and key this in. We'll cover the file in pieces to explain each one:

```
defmodule EggTimer.Server do
  use GenServer
  alias EggTimer.Alarm

  def start_link(timers) when is_map(timers) do
    GenServer.start_link(__MODULE__, timers)
  end
```

We create our boundary code in a separate file. Our goal is to keep all of the process machinery in this file, but nothing else. This code will handle all of the mechanics of the GenServer, and also do the work of sending timed messages across processes.

Next, we'll look at the code that will schedule our physical Elixir process timer:

```
def schedule(pid, alarm) do
  Process.send_after(pid, {:alarm, alarm.name}, alarm.duration)
  :ok
end
```

That code is simple enough. We take a pid and an alarm. Since this module is a server, we list the server pid as the first argument. Next, we'll look at our first callback:

```
def init(timers) do
  {:ok, timers}
end
```

This init function looks practically useless here, but there's a good reason for it. This callback is a great place to process side effects our GenServer may need, like connecting to an external resource.

init is also a great place to do any transformation on the data. For example, we might fetch a user, given an ID or something similar. Since init may use functions that fail, we *must* return an error tuple so the GenServer can properly handle startup problems.

Next, we'll do the main callback that handles a tripped alarm:

```
  def handle_info({:alarm, name}, timers) do
    Alarm.trigger(timers[name])
    new_timers = Map.delete(timers, name)

    {:noreply, new_timers}
  end
end
```

This is the heart of our server. You'll notice that the message we receive in handle_info matches the format of the message we send exactly. Just as all other handle_ callbacks do, we get the state of the GenServer in the timers argument. We trigger the timer, remove it from our map, and then feed the new list of timers, without the one we just executed, back into our server. Since the user is not expecting a reply, we send a noreply tuple with the new state of the GenServer.

We have a pretty good start, and surprisingly, we can see it in action!

Test Drive It

Let's take our timer for a test drive, even though we haven't integrated the code to schedule a timer. We'll do the work to schedule a timer by hand.

Open up IEx, or recompile, and follow this script:

```
iex(1)> alias EggTimer.Server
EggTimer.Server
iex(2)> alias EggTimer.Alarm
EggTimer.Alarm
iex(3)> a = Alarm.new :wake_up, 5000
%EggTimer.Alarm{
  duration: 5000,
  f: #Function<1.57107384/0 in EggTimer.Alarm.new/2>,
  name: :wake_up,
  time: ~T[19:51:06.540203]
}
iex(4)> {:ok, t} = Server.start_link %{a.name => a}
{:ok, #PID<0.206.0>}
```

```
iex(5)> Server.schedule t, a
:ok
Alarm triggered!
```

We alias the files we'll need, then we build an alarm. Next, we start a GenServer, and we give it a map with our alarm already inside. Eventually, we'll have a schedule message that does that work.

Then, we schedule our timer by hand by calling the Server.schedule function with our timer and alarm. After we wait a few seconds, our alarm triggers. It's not a smooth API yet, but everything works well so far!

Next, let's build in the scheduler.

Schedule an Alarm with handle_cast

The handle_cast callback is an asynchronous message, a one-way message. Use it when you don't need a reply from a server, and you want to send the message through the GenServer API. We'll amend this advice a bit at the end of the chapter. The client calls the server and does not bother waiting for a response. We'll use a *cast* message to implement the schedule message for our timer.

Scheduling a new alarm will require a name, the duration for a timer, and a function to call when the alarm triggers. We'll implement the cast with a handle_cast.

Implement the handle_cast Callback

Key the following code into the server module. Note that our callback has a similar shape to the handle_info callback:

```
def handle_cast({:schedule, name, duration, f}, timers) do
  alarm = Alarm.new(name, duration, f)
  schedule(self(), alarm)

  {:noreply, Map.put(timers, alarm.name, alarm)}
end
```

We use handle_cast to process our message. Messages aren't function calls. Elixir will receive them with pattern matches. Idiomatic Elixir uses either simple atoms or tuples to express messages.

Since the :schedule message needs a name, duration, and function, we use a tuple instead of an atom. We schedule an alarm, and then send a response tuple telling GenServer that we don't plan to reply, and that the new state for our server is a map of timers with our new scheduled timer.

Let's dig a little bit more into response tuples.

Response Tuples

Remember, a GenServer is a *contract*, called a *behaviour*. The contract is between your application and the generic server. When you add the directive `use GenServer`, Elixir includes this behaviour so the generic server knows exactly which functions to call when certain events happen.

Part of the contract is *which callbacks* are allowed. So far, we've looked at `init`, `handle_info`, and `handle_cast`.

Another part of this contract is the *signature* of the callback functions. For example, our `handle_cast/2` callback takes a message and a state.

The final part of this contract is the *return value* of each callback. With GenServer, we'll specify responses in a *response tuple*. Depending on the type of the tuple, these tuples have two or more of the following elements:

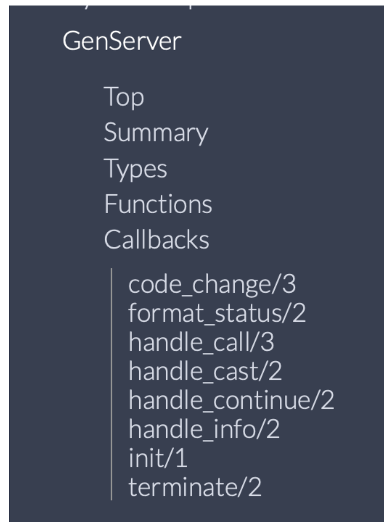
- the type of tuple, from `:noreply`, `:reply`, `:stop`, and `:ok`
- the response to send back to the server
- the value to send to the server
- extra services, which we'll get to later

So far, the two response tuples you've seen are `{:ok, state}` from the `init` callback, and the `{:noreply, state}` response tuple from `handle_info` and `handle_cast`.

The response tuples are hard to find in the GenServer documentation, unless you know where to look. Let's look at the documentation for the GenServer. We're going to dive in a bit, so open up the documentation for the GenServer on hex.[1]

Look at the lefthand margin. Click the `callbacks` link. You'll see the callbacks expanded, like in the following image:

1. https://hexdocs.pm/elixir/GenServer.html



Now click on the handle_cast callback. Look on the right-hand side, under the heading Specs. You will see the *type specs* that follow:

Specs

```
handle_cast(request :: term(), state :: term()) ::
  {:noreply, new_state}
  | {:noreply, new_state, timeout() | :hibernate | {:continue, term()}}
  | {:stop, reason :: term(), new_state}
when new_state: term()
```

This is how to interpret what you're seeing. The line

callback_name(argument_name :: argument_type, ...) :: return_type`

shows the callback name, an argument list in the form name::type, and the return type. We're looking for the return type. You can see a bunch of tuples with a | between them. Read the | as *or*.

Our handle_call can take two kinds of :noreply tuples or a :stop tuple. If you look closely at the no_reply tuple, the third argument has concepts we won't encounter in this chapter. It will be one of the following:

- An integer timeout value to specify a timeout
- A :continue tuple that GenServer will use to split a feature across two messages
- A :hibernate atom used to tell GenServer it's OK to do garbage collection

Once you know the names of the concepts, you'll know where to find them. Suffice to say, if you find yourself needing to worry about garbage collection or race conditions for long-running processes, you should explore the :hibernate and continue responses, respectively.

Next, we'll use a response tuple other than :noreply.

Shut Down Our Timer with a :stop Tuple

Now that you know what to look for, it's easy to shut down our server when something goes wrong. We just return a :stop response tuple. Add this code to our server.ex file, right below the existing handle_cast for the :schedule message:

```
def handle_cast(:stop, timers) do
  # something goes wrong
  {:stop, :we_broke_something, timers}
end
```

This callback is dead simple. We take a simple message called :stop in the function head and the existing state. We do nothing but return a :stop tuple with the required :stop atom, a reason, and the final state.

Let's try out this much. First, let's do our typical alias to make things a little less tedious, and start our server:

```
iex(1)> recompile
Compiling 1 file (.ex)
:ok
iex(2)> alias EggTimer.Server
EggTimer.Server
iex(3)> {:ok, t} = Server.start_link %{}
{:ok, #PID<0.229.0>}
```

Next, we'll schedule an alarm. While we're at it, we'll check to see if the process is alive:

```
iex(4)> GenServer.cast(t, {:schedule, :wake_up, 10_000, fn ->
  IO.puts("Wake up!")
end})
:ok
iex(5)> Process.alive? t
true
```

Eventually, our alarm fires:

```
Wake up!
```

And we can send our message to simulate failure:

```
iex(6)> GenServer.cast(t, :stop)
```

```
11:34:16.453 [error] GenServer #PID<0.229.0> terminating
** (stop) :we_broke_something
Last message: {:"$gen_cast", :stop}
State: %{wake_up: %EggTimer.Alarm{duration: 10000,
  f: #Function<20.128620087/0 in :erl_eval.expr/5>,
  name: :wake_up, time: ~U[2020-05-08 15:33:59.635523Z]}}
:ok
** (EXIT from #PID<0.198.0>)
shell process exited with reason: :we_broke_something

Interactive Elixir (1.10.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Now, you can see our supervisor at work! The process crashes, but our GenServer starts it right back up. Sure, we lose our alarms, but we'll learn to deal with that kind of failure later. Let's move on to our last major feature, a :status message.

Implement a Status Message with handle_call

The last callback we'll cover is the handle_call. The handle_call is a two-way message, or a synchronous one. Use handle_call when you want to return a result to the client. This result may be nothing more than a simple :ok acknowledgement, or it may be much more. We'll use handle_call to implement a :status feature, to get a list of all alarms.

Since a request to :status will require our server to send a message back to our client, we'll need to use handle_call. Most of the status work will come in the functional core, our alarm.ex module.

Implement the Status in the Core

Since we are dealing with timers, the most important information to the user is the timer, how long it is, and how much time is left. We'll let Elixir handle the date math for us with the DateTime module. We'll need to keep our units straight, but it should flow pretty quickly.

We'll return our status using tuples, so the client can format the status any way they want. First, we'll open up alarm.ex to add two functions, like this:

```
def status(alarm) do
  {alarm.name, alarm.duration, remaining(alarm)}
end

def remaining(alarm) do
  alarm.time
  |> DateTime.add(alarm.duration, :millisecond)
  |> DateTime.diff(DateTime.utc_now)
end
```

The status function returns a three-tuple. We get the name and duration straight from the alarm, but also must calculate the remaining time, so we write a function to do so.

The remaining function takes an alarm. We start with the time the alarm was set, add alarm.duration using the :millisecond unit, and then take the difference. Since the difference is reported in seconds by default, we're all set.

As a sanity check, let's use that much in IEx:

Test Drive the Status

Test driving will be easy. We'll create an alarm, report the status a few times, and make sure it counts down as we expect:

```
iex> alias EggTimer.Alarm
EggTimer.Alarm
iex> a = Alarm.new :timer, 20_000
%EggTimer.Alarm{
  duration: 20000,
  f: #Function<1.45243404/0 in EggTimer.Alarm.new/2>,
  name: :timer,
  time: ~U[2020-05-08 16:03:40.335048Z]
}
iex> Alarm.status a
{:timer, 20000, 14}
iex> Alarm.status a
{:timer, 20000, 9}
iex> Alarm.status a
{:timer, 20000, 2}
```

So far, so good. We're ready to wire a status message into our server.

Use handle_call to Implement a Service

Now we come to the handle_call callback. The signature is a little different from the two handle_ callbacks we've worked with so far. The reason is that the GenServer must return a result to the client. These are a few of the major differences you'll notice:

- The callback arguments include the pid of the client, usually called from
- The typical response tuple is a :reply rather than a :noreply
- The response tuple includes both the response to the client and the new state that feeds into the GenServer
- The response tuple allows for an optional timeout

Let's add our handle_call to implement our :status message. Add this code to server.ex after the handle_cast callbacks:

```
def handle_call(:status, _from, timers) do
  status =
    timers
    |> Enum.map(fn {_name, alarm} -> Alarm.status(alarm) end)

  {:reply, status, timers}
end
```

Easy enough. Our server provides the GenServer state. We build a result from the map of timers by mapping over the timers, ignoring the name key, and calling our new Alarm.status/1 function.

In the :reply tuple, we return the status list to the user, and feed the timers map back into the server. It's time for a test drive.

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> alias EggTimer.Server
EggTimer.Server
iex> {:ok, t} = Server.start_link %{}
{:ok, #PID<0.174.0>}
```

First, we recompile, alias our server, and start the server. Now to set some timers.

```
iex> GenServer.cast t, {:schedule, :eggs, 300_000, fn ->
  IO.puts "Eggs are done"
end}
:ok
iex> GenServer.cast t, {:schedule, :waffles, 600_000, fn -> IO.puts "Eggs are done" end}
:ok
```

Next, we schedule a couple of alarms for five-minute eggs and ten-minute waffles. It's time for the great payoff:

```
iex(5)> GenServer.call t, :status
[{:eggs, 300000, 280}, {:waffles, 600000, 590}]
iex(6)> GenServer.call t, :status
[{:eggs, 300000, 276}, {:waffles, 600000, 586}]
```

We call our status, and things are counting down brilliantly! They'll all go to zero and trip eventually.

There's one more issue we should cover, the concept of backpressure. We'll introduce that concept and then conclude.

Prefer Call to Cast to Create Backpressure

Often, clients send messages more quickly than a server can process them. Even services like Elixir's logger can fail. For this reason, it's often helpful to

have two-way communication between a client and server, even when it may slow down the system as a whole.

For such solutions, it's useful to build in *backpressure*, meaning when the server slows down, the clients have to slow down too. The easiest way to build this responsiveness is to use the `GenServer.call` function instead of `GenServer.cast`, and to await the return code. Then, if the server gets behind, the client must also slow down because clients can't send a following message until the previous one is completed.

For this reason, in GenServer, we typically prefer calls to casts when performance may be a problem. Also, for this reason, the *message queue length* is a great metric to check first when you're trying to address bottlenecks, as longer message queues mean a server is not getting to all messages. You can check this message with the new LiveView dashboard or Observer.

This chapter is long enough, so it's time to wrap up.

Your Turn

When you're building an Elixir multiprocess application, chances are you'll be using OTP. This chapter explored callbacks, the primary means for customizing OTP programs.

Customize GenServers with Callbacks and Response Tuples

As you communicate with your GenServers, knowing how to use the basic communication callbacks will set you up for success. The `init` callback establishes the initial state of a GenServer. The `handle_info`, `handle_call`, and `handle_cast` callbacks allow communication with a GenServer. These are the primary functions OTP will call when you send messages to a GenServer. You'll use `handle_info` to process native messages, `handle_call` to process synchronous messages, and `handle_cast` to process asynchronous messages.

Try It Yourself

We've covered the basics of OTP message passing. Now, you can experiment with those concepts by writing some of your own programs.

These *easy* problems involve changing our existing programs in ways that help you understand the core concepts.

- Change the calculator in the previous chapter to relieve backpressure with `call` instead of `cast`
- Add the API layer to our GenServer

- Make it possible to cancel an alarm without crashing the server

This *medium* problem involves working in the functional core.

- Change the `status` message to report duration and time remaining in the result tuples with hours, minutes, and seconds.

This problem is a *hard* problem. It involves trapping exit messages to build an API that shuts down the GenServer gracefully.

- Trap the exit in `init` and change the application that makes our `:stop` tuple shut down gracefully.

Next Time

In the next chapter, we're going to work on lifecycles we'll implement with supervisors. It's going to be heavy with videos and a great project, so stay tuned!

CHAPTER 3

The Lifecycle and Supervision

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.

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

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
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.

Test Drive the Application

Let's tweak the codebase to get a better feel for when it starts. Add this line within the start function, like this:

```
...
def start(_type, _args) do
  IO.puts ">>>> Starting Super-duper Super-visor <<<<"
...
```

Now, when you start up IEx, you will see it's starting:

```
[super_duper] ➔ iex -S mix
Erlang/OTP 21 [erts-10.2] [source] [64-bit] [smp:12:12] [ds:12:12:10]
[async-threads:1] [hipe]

Compiling 1 file (.ex)
>>>> Starting Super-duper Super-visor <<<<
Interactive Elixir (1.10.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

You can see that IEx automatically loads and starts the application. This IEx feature is not unique. Since Elixir has a standard way for packaging applica-

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

tions, any other project can start ours in their own supervisor children, or as extra applications. In fact, although it's just a skeleton, our application already has an extra one included! Let's try to find it.

super_duper.app File Configures Erlang Applications

If you open up the file _build/dev/lib/super_duper/ebin/super_duper.app, you'll find an Erlang data structure. The first two lines of it look like this:

```
{application,super_duper,
             [{applications,[kernel,stdlib,elixir,logger]},
```

Our application is super_duper, and a few extra applications were included for us. We get the kernel, stdlib, and elixir for free. The logger is included explicitly in mix.exs:

```
# Run "mix help compile.app" to learn about applications.
def application do
  [
    extra_applications: [:logger],
    mod: {SuperDuper.Application, []}
  ]
end
```

The so-called extra applications include our own and the logger! Normally, these listings don't include comments, but you should pay attention to that one. When you type mix compile.app, you get a few lines, including this one:

```
>> mix compile.app

>> Writes an .app file.

>> An .app file is a file containing Erlang terms
>> that defines your application. Mix automatically
>> generates this file based on your mix.exs
>> configuration.
```

Beautiful! As usual, José Valim, creator of Elixir, comes through. His foresight turned an invisible feature into an explicit step of the compilation process. This must be done to make our application execute as an OTP app within the Erlang ecosystem.

Let's dive deeper.

Run start Manually

We have pretty good control at development time over our supervisors and applications. Let's go ahead and start IEx, and tell mix not to run our application, like this:

```
[super_duper] ➔ iex -S mix run --no-start
Erlang/OTP 21 [erts-10.2] [source] [64-bit] [smp:12:12]
[ds:12:12:10] [async-threads:1] [hipe]

Compiling 1 file (.ex)
Interactive Elixir (1.10.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

This time, we get no start command. We can start the supervisor manually:

```
iex(1)> {:ok, pid} = SuperDuper.Application.start :duper, []
>>>> Starting Super-duper Super-visor <<<<
{:ok, #PID<0.137.0>}
iex(2)> Supervisor.stop pid
:ok
iex(3)> Process.alive? pid
false
iex(4)> {:ok, duper} = SuperDuper.Application.start :duper, []
>>>> Starting Super-duper Super-visor <<<<
{:ok, #PID<0.148.0>}
```

Perfect. We can control our own app, starting and stopping our supervisor at will. We have many tools at our disposal to get more information.

Process.info Gets Process Information

We can get some info about the running process, like this:

```
iex(5)> Process.info pid
[
  registered_name: SuperDuper.Supervisor,
  current_function: {:gen_server, :loop, 7},
  initial_call: {:proc_lib, :init_p, 5},
  status: :waiting,
  message_queue_len: 0,
  links: [#PID<0.135.0>],
  dictionary: [
    "$initial_call": {:supervisor, Supervisor.Default, 1},
    "$ancestors": [#PID<0.135.0>, #PID<0.75.0>]
  ],
  ...
]
iex(6)> self
#PID<0.135.0>
```

There's a ton of information packed in there! Notice that the name is SuperDuper.Supervisor. We'll talk about naming a bit later. For now, the name makes sense because SuperDuper.Application, after all, is an application, which is in turn a supervisor template.

You can see we're waiting in a GenServer message loop. In fact, a Supervisor itself is built using the GenServer specification.

You can also see that the supervisor is linked to the process ID #PID<0.135.0>. Notice that's the same pid as our IEX session! In fact, you can see that our first ancestor is also the IEx session.

We have a good start. Let's build a tiny app so we can start some children.

Add Some Children

Before we start children, we're going to need a GenServer. We'll create a few fixed characters for our server. Our nonsensical app will say a quote from a famous character. As usual, we'll put the business logic into a functional core.

Create a Core

Let's create the core in super_duper/core.

```
defmodule SuperDuper.Core do
  def say(:superdave) do
    "Next time you shoot a bullet at a metal object, watch the ricochet."
  end
  def say(:superman) do
    "It doesn't take X-Ray Vision to see you are up to no good."
  end
  def say(:supermario) do
    "Hoo hoo! Just what I needed!"
  end

  def info(name), do: {name, say(name)}
end
```

Our core has two pure functions. The say function has heads for each of our three characters. We can try it out:

```
iex(12)> recompile
Compiling 1 file (.ex)
Generated super_duper app
:ok
iex(13)> SuperDuper.Core.info(:supermario)
{:supermario, "Hoo hoo! Just what I needed!"}
iex(14)> SuperDuper.Core.info(:superman)
{:superman, "It doesn't take X-Ray Vision to see you are up to no good."}
iex(15)> SuperDuper.Core.info(:superdave)
{:superdave,
 "Next time you shoot a bullet at a metal object, watch the ricochet."}
```

It works just fine. The next step is to add a boundary layer.

Establish a Boundary

Now, let's whip up a GenServer, one that accepts :say and :die messages. We'll need an info callback, a handle_call for say, and a handle_cast for :die to simulate failure. Open up super_duper/server.ex and key this in:

```
defmodule SuperDuper.Server do
  use GenServer
  alias SuperDuper.Core

  def init(character) do
    IO.puts "Starting #{character}"
    {:ok, Core.info(character)}
  end

  def handle_cast(:die, state) do
    raise "Boom"
    {:no_reply, state}
  end

  def handle_call(:say, _from, {_name, says}=state) do
    {:reply, says, state}
  end
end
```

We build a tiny GenServer with the usual callbacks for info, handle_cast, and handle_call. The :die message is just a tool to help us simulate instability.

And add the API:

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

def die(server), do: GenServer.cast server, :die
def say(server), do: GenServer.call server, :say
```

That's the same API layer we've built before. We have a convenient interface. Now we can try that out:

```
iex(1)> recompile
Compiling 1 file (.ex)
:ok
iex(2)> alias SuperDuper.Server
SuperDuper.Server
iex(3)> {:ok, pid} = Server.start_link :superdave
Starting superdave
{:ok, #PID<0.172.0>}
iex(4)> Server.say pid
"Next time you shoot a bullet at a metal object, watch the ricochet."
```

So far so good. Now, what happens when we make a little mischief for our venerable hero?

```
iex(5)> Server.die pid
:ok
iex(6)>
18:12:51.979 [error] GenServer :superdave terminating
** (RuntimeError) Boom
    ...stacktrace...
Last message: {:"$gen_cast", :die}
State: {:superdave,
  "Next time you shoot a bullet at a metal object, watch the ricochet."}
** (EXIT from #PID<0.150.0>)
shell process exited with reason: an exception was raised:
    ** (RuntimeError) Boom
        ...stacktrace...

Interactive Elixir (1.10.3) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Well, that was disappointing. Crashing our server crashed our IEx session. That's no good! We know how to fix it, though. We can start children in our supervisor.

Add the Children to a Supervisor

Let's start each of our children in the SuperDuper.Application file. Let's review the contents of the start function:

```
def start(_type, _args) do
  IO.puts ">>>> Starting Super-duper Super-visor <<<<"
  children = []

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

We'll need to populate the children argument, but we need to specify the right format for the children. We need a *child spec*. The hex docs for child spec[3] say that a child spec is a map with six potential keys. The first two keys, a :id and a :start, are required. The :id must be unique, and is used as a key in the process registry. The :start key specifies a tuple with the name of the module and the arguments you can use to start the child.

You can check the documentation for the children. It's excellent. For now, we know enough to specify a child spec. Key in these changes:

```
def start(_type, _args) do
  IO.puts ">>>> Starting Super-duper Super-visor <<<<"
  children = [
    %{id: :superdave, start: {Server, :start_link, [:superdave]}},
```

3. https://hexdocs.pm/elixir/Supervisor.html#module-child-specification

```
    %{id: :superman, start: {Server, :start_link, [:superman]}},
    %{id: :supermario,    start: {Server, :start_link, [:supermario]}}
  ]

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

We specify which children to start, in order. We're using a *child spec*. Each child spec has an id key so we can refer to processes by their IDs instead of their pids, and a start key to tell the supervisor exactly how to start the process.

Test Drive the Supervisor

Let's take things for a test drive. We'll make sure to recompile and stop the existing supervisor. Remember, the name is SuperDuper.Supervisor:

```
iex(4)> recompile
...
iex(5)> Supervisor.stop SuperDuper.Supervisor
:ok
```

Things shut down just fine. Now, we can restart the app:

```
iex(6)> SuperDuper.Application.start SuperDuper.Supervisor, []
>>>> Starting Super-duper Super-visor <<<<
Starting superdave
Starting superman
Starting supermario
{:ok, #PID<0.169.0>}
```

Perfect! Now we have three GenServers started. We can access them by name:

```
iex> alias SuperDuper.Server
SuperDuper.Server
iex> Server.
child_spec/1    die/1           init/1          say/1
start_link/1
iex> Server.say :supermario
"Hoo hoo! Just what I needed!"
iex> Server.say :superdave
"Next time you shoot a bullet at a metal object, watch the ricochet."
iex> Server.say :superman
"It doesn't take X-Ray Vision to see you are up to no good."
```

In truth, our API is a bit messy. We can tighten up the child spec. Let's find out how.

Tighten Up the Child Spec

We could refactor our code to let a function build our child specs. It turns out that writing a function to build child specs is a common pattern, so use GenServer builds in a child_spec function automatically. Let's see what ours returns:

```
iex> Server.child_spec :supermario
%{id: SuperDuper.Server,
  start: {SuperDuper.Server, :start_link, [:supermario]}}
```

That doesn't quite work for us because we get the module name instead of the character name, but we can override it. Let's add our own child_spec/1 to our server.ex:

```
def child_spec(name) do
  %{id: name, start: {__MODULE__, :start_link, [name]}}
end
```

Now, we can use that spec to tighten up the child list in application.ex. We can specify the children like this:

```
children = [
  {Server, :superdave},
  {Server, :superman},
  {Server, :supermario}
]
```

That's better. We specify each child as a tuple with the module name and starting parameters.

The Supervisor Protects from Failure

Now is the moment we've all been waiting for. We can try to feed :superman a little kryptonite:

```
iex(11)> Server.die :superman
:ok
iex(12)>
18:47:54.559 [error] GenServer :superman terminating
** (RuntimeError) Boom
    (super_duper 0.1.0) lib/super_duper/server.ex:18:
    SuperDuper.Server.handle_cast/2
    (stdlib 3.7) gen_server.erl:637: :gen_server.try_dispatch/4
    (stdlib 3.7) gen_server.erl:711: :gen_server.handle_msg/6
    (stdlib 3.7) proc_lib.erl:249: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", :die}
State: {:superman,
  "It doesn't take X-Ray Vision to see you are up to no good."}
Starting superman
```

```
iex(12)> Server.say :superman
"It doesn't take X-Ray Vision to see you are up to no good."
```

But the man of steel restarts! The OTP story doesn't stop there, though. Part of handling lifecycles is establishing a policy for the way services started or restarted.

We can verify with `GenServer.whereis`, like this:

```
iex(14)> pid = GenServer.whereis :superman
#PID<0.179.0>
iex(15)> Process.alive? pid
true
```

Superman lives to fly another day! Let's look at some of the finer knobs and levers involved with starting a GenServer.

Lifecycle Policy

We've described the *supervisor* as a place to have lifecycle policy. There are four concerns we need to consider. Two are constant: startup order and shutdown order. Two more may vary depending on application requirements, shutdown policy, and restart policy.

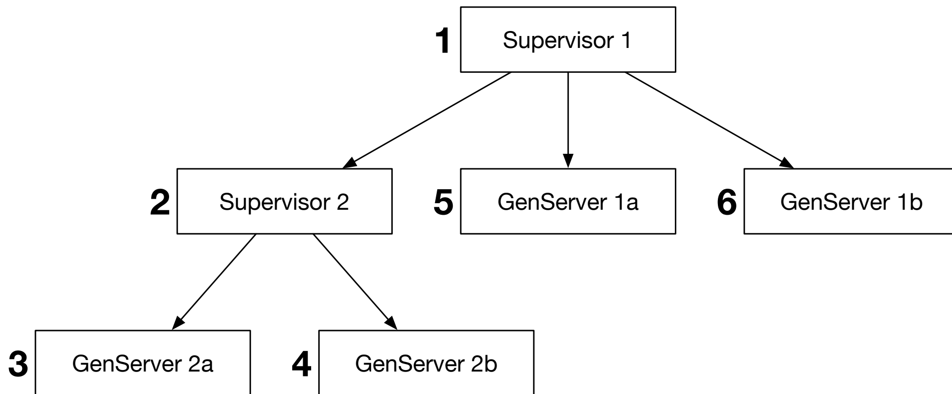
Let's start with the startup and shutdown order. Then we can move into policy.

Startup and Shutdown Are Synchronous and Ordered

When an OTP supervisor starts any application, it will start the supervisor, and then start the children in the order you specify. The startup will also be *synchronous*, meaning one process (including any children) will finish coming up before another starts. You probably recognize the *call* mechanism the supervisor is using under the hood.

So far, our supervisor's children are plain GenServers, but they don't have to be. When a supervisor starts other *applications*, the children will be other supervisors. That means we have a tree with supervisors and the children for each of those supervisors.

Think about a typical tree. Say `supervisor1` has three children: `supervisor2`, `genserver1a`, and `genserver1b`. Say `supervisor2` also has two children, as shown in the following figure.



You can see now why OTP developers use the term *supervision tree*. Let's walk through the rules we specified about ordering, synchronous messaging, and the child relationships in a proper list:

- It starts children *in the order they are listed*.
- It waits for acknowledgment after starting each child before moving on to the next one.
- As part of its own startup, each supervisor starts its own children.

That means the startup order follows the numbers in the previous figure. Shutdown happens precisely in reverse.

If you think about it, having a precise order makes sense. The logger must start before our application. A connection pool must start before applications that connect to a database. More generally, some services depend on others. Processes must come *after* dependencies in a supervisor's child list.

Let's look back at what happened when we started our application:

```
>>>> Starting Super-duper Super-visor <<<<
Starting superdave
Starting superman
Starting supermario
```

If we started the same code a million times, each successful start would look exactly like this one.

Once you know what your dependencies are, you can start to think about the startup and shutdown policy for failure cases.

Set a Restart Policy

None of our services have dependencies between each other. As such, it makes sense that in the event of failure, only the affected process should restart.

Let's try it out. Let's add a bit of code to track termination in server.ex, like this:

```
def terminate(_reason, {name, _says}=state) do
  IO.puts "Mayday! Mayday! #{name} going down..."

  {:error, "oh noes", state}
end
```

Now we can recompile, stop, and start the supervisor:

```
iex(24)> recompile
Compiling 1 file (.ex)
:ok
iex(25)> Supervisor.stop SuperDuper.Supervisor
:ok
iex(26)> SuperDuper.Application.start SuperDuper.Supervisor, []
>>>> Starting Super-duper Super-visor <<<<
Starting superdave
Starting superman
Starting supermario
{:ok, #PID<0.262.0>}
```

Our supervisor defines a strategy to control the restart policy. You can find it in application.ex:

```
opts = [strategy: :one_for_one, name: SuperDuper.Supervisor]
```

That strategy: :one_for_one atom is our restart policy. This one means we'll restart only the failing process.

The policy works for us because none of our GenServers depends on any other. Let's imagine what we would do if we did have dependencies.

Recall the listing of our children:

```
children = [
  %{id: :superdave, start: {Server, :start_link, [:superdave]}},
  %{id: :superman, start: {Server, :start_link, [:superman]}},
  %{id: :supermario,    start: {Server, :start_link, [:supermario]}}
]
```

GenServer starts them in the order of Super Dave, Superman, and Super Mario. The shutdown goes in the opposite order.

Here's what happens when we kill one node. Let's tug on Super Dave's cape. Make sure you recompile and restart servers if you need to:

```
iex(24)> Server.die :superdave
Mayday! Mayday! superdave going down...
:ok
iex(25)>
```

```
11:27:50.377 [error] GenServer :superdave terminating
** (RuntimeError) Boom
    (super_duper 0.1.0) lib/super_duper/server.ex:25:
    SuperDuper.Server.handle_cast/2
    (stdlib 3.7) gen_server.erl:637: :gen_server.try_dispatch/4
    (stdlib 3.7) gen_server.erl:711: :gen_server.handle_msg/6
    (stdlib 3.7) proc_lib.erl:249: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", :die}
State: {:superdave,
  "Next time you shoot a bullet at a metal object, watch the ricochet."}
Starting superdave
```

As we expected, only a single process died and was restarted. That policy will work fine most of the time, but there may be dependencies between children.

One for All Restarts All Children

Imagine all of our GenServers had dependencies between each other, so that each had the process ID of the other two. We would want a restart policy to restart all other children in the event any one failed. Change the strategy in your application.ex to :one_for_all, like this:

```
opts = [strategy: :one_for_all, name: SuperDuper.Supervisor]
```

That tiny bit of code packs a punch! We've completely rewired the restart policy. To see the impact, let's recompile and restart things:

```
iex(14)> recompile
Compiling 1 file (.ex)
:ok
iex(15)> Supervisor.stop SuperDuper.Supervisor
Mayday! Mayday! supermario going down...
Mayday! Mayday! superman going down...
Mayday! Mayday! superdave going down...
:ok
iex(16)> SuperDuper.Application.start SuperDuper.Supervisor, []
>>>> Starting Super-duper Super-visor <<<<
Starting superdave
Starting superman
Starting supermario
{:ok, #PID<0.368.0>}
```

We can see the shutdown goes in reverse order of the startup, just as we expected. Now, let's put a wrinkle into Super Dave's trousers:

```
Mayday! Mayday! superdave going down...
:ok
iex(18)>
11:21:09.155 [error] GenServer :superdave terminating
** (RuntimeError) Boom
```

```
    (super_duper 0.1.0) lib/super_duper/server.ex:25:
    SuperDuper.Server.handle_cast/2
    (stdlib 3.7) gen_server.erl:637: :gen_server.try_dispatch/4
    (stdlib 3.7) gen_server.erl:711: :gen_server.handle_msg/6
    (stdlib 3.7) proc_lib.erl:249: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", :die}
State: {:superdave,
  "Next time you shoot a bullet at a metal object, watch the ricochet."}
Starting superdave
Starting superman
Starting supermario
```

Perfect! Super Dave goes down. GenServer restarts all three processes. Pay close attention to the flow of callbacks. After the initial terminate callback for :superdave, we don't get any other terminate events. When you think about it, that flow makes sense because we only want callbacks for terminates that require our action, not the ones the GenServer restarts based on policy.

Before we move on, there's one more policy we should try.

Rest for One Restarts Children with Dependencies

The last interesting scenario is that a child list only implements *proper dependencies*. A proper dependency means a supervisor's children can only depend on previous children in the supervisor tree. The last policy works like this. Change the restart policy to :rest_for_one, and you'll get this result. Make sure to recompile and restart the server:

```
iex(28)> Server.die :superman
Mayday! Mayday! superman going down...
:ok
iex(29)>
11:30:23.924 [error] GenServer :superman terminating
** (RuntimeError) Boom
    (super_duper 0.1.0) lib/super_duper/server.ex:25:
    SuperDuper.Server.handle_cast/2
    (stdlib 3.7) gen_server.erl:637: :gen_server.try_dispatch/4
    (stdlib 3.7) gen_server.erl:711: :gen_server.handle_msg/6
    (stdlib 3.7) proc_lib.erl:249: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", :die}
State: {:superman,
  "It doesn't take X-Ray Vision to see you are up to no good."}
Starting superman
Starting supermario
```

Nice! This policy works great when there are dependencies between servers listed *earlier* in the child list, but no dependencies in the opposite direction.

This interesting configuration option can save many restarts in the event of failure in large supervision trees.

You can see how tweaking a bit of configuration code can give you tremendous returns. This is one of our longest chapters so far, so it's past time to wrap up.

Your Turn

In this chapter, we put OTP through its paces. You got to see firsthand how to package a generic Elixir application. We started with `mix new` using the `sup` option and built a full application piece by piece. Instead of focusing on the internals of our application, we lived mostly in the supervisor layer. Let's summarize how things work.

Supervisors Implement Lifecycles

Where GenServers handle application concerns, supervisors define lifecycles. Together, these two layers form the heart of OTP.

A supervisor is like a process server for your application. It's responsible for implementing the lifecycle of your project. Your job is to tell OTP how to start your application, including all of your GenServers, dependencies, other supervisors, and applications. The result is a tree of processes called a supervisor tree.

Supervisors start and stop their children in a specified order to preserve dependencies between them. In the event of failure, they can follow restart policies you specify. Configuring these policies is nearly trivial.

The best way to understand what's happening is to build your own supervisors, and change the existing code of services that already exist.

Try It Out

These exercises range from easy to difficult. The difficulty is not in the amount of code you need to write. Each of the solutions is short. The difficulty lies in understanding the underlying technology.

These *easy* problems build onto the existing application.

- Add an API layer to `SuperDuper` in `super_duper.ex`.
- Explicitly change the child spec in `SuperDuper` to specify that the three `GenServer` processes are workers.
- Change `SuperDuper` so that the workers are `:temporary`, and do not restart when they fail. What happens when you kill a process?

- Change SuperDuper so that the workers are :transient. How can you make sure :kill messages restart? Stay dead?

This *medium* problem involves working with the EggTimer app from Chapter 2, Communication Between Servers, on page 17.

- Build a supervisor for the EggTimer application in Chapter 2. Crash the egg timer. Does it come back as you expect?

These *hard* problems involve building a dynamic supervisor.

- If you would like to peek ahead, you can add a dynamic supervisor to the SuperDuper application. We'll cover dynamic supervisors more in the next chapter.
- Build the API layer for SuperDuper which adds a new supervised character via an API.

Next Time

In the final chapter, we'll focus on working with dynamic supervisors. These supervisors will let you create new instances at any time, not just when your application is started. It's going to be a nice complement to OTP!

CHAPTER 4

The Power of a Name

In our journey so far, we've explored both major layers of OTP. Let's take a few brief moments to explore what's happened because these ideas will shape what we do next.

First, we coded a typical Elixir application with a process and a message loop without OTP, and then with it. The OTP library relies on an application template called GenServer. All OTP applications have these servers. Rather than starting the server directly, we used a *supervisor* to manage our applications, start our applications when the application starts, and stop them when our application shuts down or a restart is required.

So far, each of the applications we start is *a fixed GenServer or supervisor* in the supervisor's child list. Since we supply that list when the supervisor starts, we have a limitation. Many of the types of processes we'd like to manage—an end user's session in a web server, a new scene in a user interface, or a new connection to a network—are dynamic.

If you think about it, Elixir is good at managing lists. OTP supervisors are good at starting and stopping things at any time. We can't refer to our processes by a process ID because they can potentially fail, so the main problem we need to solve to enable a dynamic server is referring to a process by some name other than the process ID. We need a naming strategy.

In this chapter, we'll explore the *registries, via tuples, and dynamic supervisors* that OTP uses to build dynamic supervision trees that manage our project's lifecycle. Let's get started.

Add Dynamic Characters to SuperDuper

Think about the `SuperDuper` service we built in the previous chapter. The service works just fine if there are only three static characters. Sometimes, programs

need to add services dynamically. Servers that model individual users are almost always unpredictable, requiring starts and stops. Scaling up and down based on load is another area that might require dynamically adding and removing services.

We're going to build a feature to let SuperDuper have both static and dynamic characters. Along the way, we'll take a deeper dive into how the names work.

Before we get too far, let's review our application and see how it works.

Explore SuperDuper in IEx

Navigate to your super_duper project and start IEx with iex -S mix. We'll see the supervision tree start up, piece by piece:

```
[super_duper] ➔ iex -S mix
Erlang/OTP 21 [erts-10.2] [source] [64-bit] [smp:12:12] ...
>>>> Starting Super-duper Super-visor <<<<
Starting superdave
Starting superman
Starting supermario
Interactive Elixir (1.10.3) - press Ctrl+C to exit (type h() ENTER for help)
```

Elixir, through OTP, has started our application. One of the applications it started was in SuperDuper.Application. Remember, we started our application with the start function in application.ex:

```
children = [
    {Server, :superdave},
    {Server, :superman},
    {Server, :supermario}
  ]

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

The name is SuperDuper.Supervisor. We can see metadata about all of the currently started children with the command Supervisor.which_children, using our supervisor's name:

```
iex(1)> Supervisor.which_children SuperDuper.Supervisor
[
  {:supermario, #PID<0.141.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.140.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.139.0>, :worker, [SuperDuper.Server]}
]
```

Perfect. We have three children, :supermario, :superman, and :superdave. We can see the names for each, and the process for each one. The next step is to understand how we might add children dynamically, after runtime.

A Plan for (Dynamic) Children

Let's think about how we might add a child to our family of supers, or shut one down. Type exports Supervisor into your IEx console:

```
iex(2)> exports Supervisor
__using__/1        child_spec/2      count_children/1   delete_child/2
init/2             restart_child/2   start_child/2      start_link/2
start_link/3       stop/1            stop/2             stop/3
terminate_child/2  which_children/1
```

The start_child looks promising. Briefly calling h start_child/2 tells us that the arguments are a supervisor and a child spec. That solves one problem.

We have another problem, though. Take a look at our start_link and init functions:

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

...

def init(character) do
  {:ok, Core.info(character)}
end

...

def child_spec(name) do
  %{id: name, start: {__MODULE__, :start_link, [name]}}
end
```

Our code looks up the initial data for each child GenServer directly from our core. The start_link, info callback, and the child_spec functions all participate in this strategy. We'll need to make this code dynamic, instead of looking up static information from our code. It shouldn't be too hard to do.

That means we have to use the DynamicSupervisor.start_child function to start the supervisor, then make the information we pass to a starting child dynamic. Let's get started.

Dynamic Children

When traditional supervisors start services, they usually use a static list of children. A dynamic service can't depend on a list of children, so we'll need to provide an API instead. We'll also need to be able to refer to each of the children by name. Remember, supervisors are about lifecycles. Our plan is to build an API to add_character, and another to stop a character. Then we'll touch up the rest of the functions to refer to GenServers by name.

We'll need to start with the functions we've identified: init, start_link, and child_spec. Not surprisingly, these functions all have lifecycle names. Tweaking them to support both dynamic and static services becomes pretty easy once we get Elixir's pattern matching involved.

Tweak the Lifecycle APIs

We'll start with the start_link function. It takes one argument, a character. We'll need to pass in both the character's name and its statement. We'll leave the original start_link in place and add one that accepts a tuple:

```
def start_link({character, _says}=info) do
  GenServer.start_link(__MODULE__, info, name: character)
end
def start_link(character) do
  GenServer.start_link(__MODULE__, character, name: character)
end
```

Nice. Our function head matching the new tuple must come first. Otherwise, the character would match both the tuple and atom forms. We extract the character to name the server and pass the info through.

The init callback comes next. We do the same trick, adding a function head with a tuple first:

```
def init({character, _says}=info) do
  IO.puts "Starting #{character}"
  {:ok, info}
end
def init(character) do
  IO.puts "Starting #{character}"

  {:ok, Core.info(character)}
end
```

These callbacks look much like the start_link ones. Finally, we need to tweak the child_spec, like this:

```
def child_spec({name, says}) do
  %{id: name, start: {__MODULE__, :start_link, [{name, says}]}}
end
def child_spec(name) do
  %{id: name, start: {__MODULE__, :start_link, [name]}}
end
```

Beautiful. We extract the name to pass to the spec as the id key, and then we pass through the tuple. We leave the second form of the child_spec alone to handle the short form.

Let's try out what we have so far.

Manually Add a Child in IEx

Recompile the code you have so far or fire up IEx. Let's say we wanted to add another child. You could do it this way. Let's say we wanted to add another child, for instance, Yoshi from the Smash Brothers game. You could do it this way:

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> app = SuperDuper.Supervisor
SuperDuper.Supervisor
iex> Supervisor.which_children app
[
  {:supermario, #PID<0.141.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.140.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.139.0>, :worker, [SuperDuper.Server]}
]
iex> cspec = Server.child_spec({:yoshi, "Yoshi do! Yoshi do!"})
%{
  id: :yoshi,
  start: {SuperDuper.Server, :start_link, [yoshi: "Yoshi do! Yoshi do!"]}
}
iex> Supervisor.start_child app, cspec
Starting yoshi
{:ok, #PID<0.182.0>}
iex> Supervisor.which_children SuperDuper.Supervisor
[
  {:yoshi, #PID<0.182.0>, :worker, [SuperDuper.Server]},
  {:supermario, #PID<0.141.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.140.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.139.0>, :worker, [SuperDuper.Server]}
]
iex> Server.say :yoshi
"Yoshi do! Yoshi do!"
```

Excellent! It works! Now, we can build out our SuperDuper service.

Build an API with add_character

Our API works exactly like it did before, with one small difference. Instead of calling a start_link directly, we're going to do the work through the supervisor. Open up super_duper.ex so we can add our API, like this:

```
defmodule SuperDuper do
  alias SuperDuper.Server
  @app __MODULE__.Supervisor

  def add_character(name, says) do
    Supervisor.start_child @app, Server.child_spec({name, says})
  end
```

```
  def say(character) do
    Server.say(character)
  end

  def die(character) do
    Server.die(character)
  end
end
```

It's too easy. We let our `add_character` start our server with the new child spec. We pass through the other requests directly to our `Server`.

Now, we can take it for a spin. Let's start it up:

```
[super_duper] ➔ iex -S mix
Erlang/OTP 21 [erts-10.2] [source] [64-bit] [smp:12:12]
[ds:12:12:10] [async-threads:1] [hipe]

>>>> Starting Super-duper Super-visor <<<<
Starting superdave
Starting superman
Starting supermario
Interactive Elixir (1.10.3) - press Ctrl+C to exit (type h() ENTER for help)
iex> SuperDuper.say :supermario
"Hoo hoo! Just what I needed!"
```

We start up our service, and use our existing service, as usual. Let's add a new server dynamically:

```
iex> SuperDuper.add_character :supernova, "Expanding at light speed"
Starting supernova
{:ok, #PID<0.146.0>}
iex> SuperDuper.say :supernova
"Expanding at light speed"
```

We add a service and make sure it's up. Let's test it out. What happens when we kill it?

```
iex> SuperDuper.die :supernova
:ok
iex> Mayday! Mayday! supernova going down...
11:01:08.896 [error] GenServer :supernova terminating
** (RuntimeError) Boom
...
Starting supernova
iex> SuperDuper.say :supernova
"Expanding at light speed to a theater near you"
```

It dies and starts right back up! We're referring to it by name, so that we can use the new service just fine. Let's see how that works.

Stop a Character Manually

Shutting down a server has a little more nuance than adding one, so let's bring out the trusty console and play around a bit. The console can show us how to build the service we'll need to remove a character from our list.

First, let's start with a clean console. Shut down your console and restart it with `iex -S mix`. Then set up a few bits of convenience:

```
iex> app = SuperDuper.Supervisor
SuperDuper.Supervisor
iex(22)> cs = Server.child_spec({:superfreak, "She's alright"})
%{
  id: :superfreak,
  start: {SuperDuper.Server, :start_link, [superfreak: "She's alright"]}
}
iex> alias SuperDuper.Server
SuperDuper.Server
```

Perfect. The service is up and working. Now, let's start a child process, `:superfreak`:

```
iex> Supervisor.start_child app, cs)
Starting superfreak
{:ok, #PID<0.159.0>}
iex> Server.say :superfreak
"She's alright"
```

So far, so good. We're going to need a way to shut down a process. Type `Supervisor.` and then the tab key. If you have tab completion set up, you'll see the following listing. If not, use `exports Supervisor` instead:

```
iex> Supervisor.
Default              Spec                 child_spec/2
count_children/1     delete_child/2       init/2
restart_child/2      start_child/2        start_link/2
start_link/3         stop/1               stop/2
stop/3               terminate_child/2    which_children/1
```

Both `terminate_child` and `delete_child` look promising. Let's try them out:

```
iex> Supervisor.terminate_child app, :superfreak
:ok
iex> Supervisor.which_children app
[
  {:superfreak, :undefined, :worker, [SuperDuper.Server]},
  {:supermario, #PID<0.151.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.150.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.149.0>, :worker, [SuperDuper.Server]}
]
```

The child is still listed, but its pid is :undefined. Let's see if we can use it:

```
iex> SuperDuper.say :superfreak
** (exit) exited in: GenServer.call(:superfreak, :say, 5000)
    ** (EXIT) no process: the process is not alive
    or there's no process currently associated with the given name,
    possibly because its application isn't started
    ...
```

No dice. It's down. We can delete it though:

```
iex> Supervisor.delete_child app, :superfreak
:ok
iex> Supervisor.which_children app
[
  {:supermario, #PID<0.151.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.150.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.149.0>, :worker, [SuperDuper.Server]}
]
```

Now, it's gone. There's just another bit of exploration we need to do to shut down cleanly. First, we need to be able to add a child again after it's been removed. Let's see what happens when a child is in the :undefined state:

```
iex> Supervisor.start_child app, cs)
Starting superfreak
{:ok, #PID<0.159.0>}
iex> Supervisor.terminate_child app, :superfreak
:ok
```

We start a child and terminate the child, without deleting it. Now, let's try a start:

```
iex> Supervisor.start_child app, cs)
{:error, :already_present}
iex> Supervisor.restart_child app, :superfreak
Starting superfreak
```

So we can't start it, but we can restart it, and we don't even need a child spec. That doesn't help us, though. It looks like our remove_child API is going to need to terminate and delete the child. Then, we'll be able to start another cleanly.

Let's finish up this API layer.

Shutdown Via the API

Once again, you can see that the supervision layer is about managing GenServer lifecycle. Our dynamic function to remove children will need to call

both Supervisor.terminate and Supervisor.delete_child, but that's all we'll need to do. Add this final bit of API code to super_duper.ex, like this:

```
def remove_character(name) do
  Supervisor.terminate_child(@app, name)
  Supervisor.delete_child(@app, name)
end
```

We add the tiny bits of code to terminate a child and remove it from the tree. Let's make sure it works:

```
iex> SuperDuper.add_character :superfreak, "She's alright"
Starting superfreak
{:ok, #PID<0.226.0>}
iex> Supervisor.which_children app
[
  {:superfreak, #PID<0.226.0>, :worker, [SuperDuper.Server]},
  {:supermario, #PID<0.151.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.150.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.149.0>, :worker, [SuperDuper.Server]}
]
iex> SuperDuper.remove_character :superfreak
:ok
iex> Supervisor.which_children app
[
  {:supermario, #PID<0.151.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.150.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.149.0>, :worker, [SuperDuper.Server]}
]
```

Nice! We add a child and then we remove it. We're working dynamically! There are still a few tweaks we can make. Let's look at a few problems.

Dynamic Supervisors

Our supervisor is working great. There are a few problems, though. Think about a web server such as Phoenix. Most of its servers would be dynamically added. Think about the guarantees of a supervisor. The children are *started in order* and *restarted in reverse order*. When all of these are processed sequentially, this strategy can lead to long waits when we shut down or restart servers.

Fortunately, OTP provides a good solution, the dynamic supervisor.

Since dynamically supervised servers are all independent, OTP can shut them down and even restart them at the same time. Since a dynamic supervisor is really just a supervisor, we can make a couple of tiny tweaks to our application and keep it running just fine.

Here's our plan. We'll start our three regular children and a dynamic supervisor in the SuperDuper.Application module. Then, we'll use that supervisor to start and shut down our individual modules. Let's see what that code looks like.

First, we start the dynamic supervisor.

```
children = [
  {Server, :superdave},
  {Server, :superman},
  {Server, :supermario},
  {
    DynamicSupervisor,
    name: SuperDuper.DynamicSupervisor,
    strategy: :one_for_one
  }
]
```

Easy enough. We just add a dynamic supervisor to the list. Now, we can use our existing API to delete it from the dynamic supervisor, like this:

```
@app __MODULE__.DynamicSupervisor

def add_character(name, says) do
  DynamicSupervisor.start_child @app, Server.child_spec({name, says})
end

def remove_character(name) do
  DynamicSupervisor.terminate_child(@app, GenServer.whereis(name))
end
```

We change the supervisor from SuperDuper.Supervisor to SuperDuper.DynamicSupervisor. Then, we use the API to create and stop the children. Notice there's no need to do the extra delete_child as this version of the terminate_child handles the removal for us. The final change is that terminate does take a pid, so we need to look that up.

Let's try it out. Start your server from scratch.

```
iex> alias SuperDuper.Server
SuperDuper.Server
iex> SuperDuper.add_character :supervisor, "Keeps my eyes shaded"
Starting supervisor
{:ok, #PID<0.147.0>}
iex> Supervisor.which_children SuperDuper.Supervisor
[
  {SuperDuper.DynamicSupervisor, #PID<0.142.0>, :supervisor,
   [DynamicSupervisor]},
  {:supermario, #PID<0.141.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.140.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.139.0>, :worker, [SuperDuper.Server]}
]
```

```
iex> DynamicSupervisor.which_children SuperDuper.DynamicSupervisor
[{:undefined, #PID<0.147.0>, :worker, [SuperDuper.Server]}]
```

We do a quick alias and add a child with a very bad pun. Then, we check to see that the dynamic supervisor is added, and that the supervisor has the new child. Now, we can remove the last child.

```
iex(8)> SuperDuper.remove_character :supervisor
:ok
iex(9)> DynamicSupervisor.which_children SuperDuper.DynamicSupervisor
[]
```

We remove the last child and it's gone!

The Process Registry: The Power of a Name

Everything we've done is possible because Elixir makes sure you can find the processes you need through a central registry. In fact, let's look at the snippet that removes our child:

```
DynamicSupervisor.terminate_child(@app, GenServer.whereis(name))
```

The terminate_child refers to a child by a pid. The last part of that function looks up the pid from a name. GenServer gives us an easy way to do this, but we could have just as easily used the process registry. Let's go back to IEx and see exactly what's happening.

GenServer Defaults to the Default Process Registry

```
iex(1)> app = SuperDuper.Supervisor
SuperDuper.Supervisor
iex(2)> Supervisor.which_children app
[
  {SuperDuper.DynamicSupervisor, #PID<0.152.0>, :supervisor,
   [DynamicSupervisor]},
  {:supermario, #PID<0.151.0>, :worker, [SuperDuper.Server]},
  {:superman, #PID<0.150.0>, :worker, [SuperDuper.Server]},
  {:superdave, #PID<0.149.0>, :worker, [SuperDuper.Server]}
]
```

You can see that supervisors register each of these names, and hold those names in a permanent child list. When you start a child with a child spec that has an id, or when you call a start_link with a name: option, you *register that process*.

The default mechanism to register a process is the process registry. We can look it up using either GenServer.whereis or Process.whereis:

```
iex(3)> GenServer.whereis :superdave
```

```
#PID<0.149.0>
iex(4)> Process.whereis :superdave
#PID<0.149.0>
```

These both have the same pid because they use the same registry for process lookup.

Alternative Registries

Sometimes, you need a more advanced solution for finding processes. GenServer has three ways to name processes.

Atoms.
: Since module names are atoms, we typically see this option.

Global terms.
: These look like `{:global, global_term}`. They use the *global registry*. See Erlang's `:global` module[1] for more details.

Via tuples.
: These look like `{:via, Module, term}`. This strategy lets you implement your own custom registry. Your module must implement `register_name/2`, `unregister_name/1`, `whereis/1`, and `send/2`. See the GenServer behaviour[2] for more details.

These functions make everything we've talked about possible. Instead of referring to a process by ID, we can refer to it by name. That means in the event of failure, we can simply look up the new pid and be off to the races.

Now, if you should ever decide to do so, you can change your naming strategy to meet the needs of the application. With those final details, we're ready to close out this final OTP chapter.

Your Turn

We just finished our OTP series with a dynamic supervisor, and took a brief look at the various naming strategies you might encounter. We built up our dynamic supervisor by exploring the excellent Elixir documentation. Let's review how things work.

Dynamic Supervisors Register Through an API

While supervisors use child specifications to start lists of child processes when a user or other OTP application starts your project, dynamic supervisors

1. http://erlang.org/doc/man/global.html
2. https://hexdocs.pm/elixir/1.5.1/GenServer.html

use an explicit API to add children to an existing tree. There are a few major differences. Primarily, dynamic supervisors shut down all children at the same time, without guarantees regarding shutdown or restart order, and that's what we want for the sake of efficiency.

Naming is an important part of the overall equation. Registries exist for that purpose. The Groxio video series has a video on via tuples that will walk you through those problems.

Now is a good time to put what you've learned into practice.

Try It Yourself

This *easy* problem builds onto the existing SuperDuper application.

- Rather than restarting the application on a crash, make the project restart.

These *medium* problems also build onto the existing SuperDuper application.

- Add a custom registry that allows you to look up entries using a via tuple.
- Use ETS to save state when you add a dynamic server, and then allow your GenServers to look up that state when there's a crash.
- Make the SuperDuper app distributed. You'll need to build a registry with a via tuple that looks up a server based on the existing node.

This *hard* problem shows the composition within an OTP app.

- Design a rock-paper-scissors GenServer. Then, design a waiting room so that when one player gets added, the GenServer waits until a second is added. When the second player is added, it connects them, dynamically starting a game. How did you decide to manage crashes? Did you decide to make them transient or save state somehow?

This concludes our whirlwind tour through OTP. Along the way, we've explored the same library that the OTP team built more than 30 years ago! You've built GenServers, sent them messages with call and cast, crashed them, and recovered from those crashes. You've explored simple techniques to layer your systems and even launch services dynamically. If you like the presentation of these concepts and want more, please join us on Groxio or in other Pragmatic Bookshelf books. We'd love to have you.

Bibliography

[Hal18] Lance Halvorsen. *Functional Web Development with Elixir, OTP, and Phoenix*. The Pragmatic Bookshelf, Raleigh, NC, 2018.

[Heb19] Fred Hebert. *Property-Based Testing with PropEr, Erlang, and Elixir*. The Pragmatic Bookshelf, Raleigh, NC, 2019.

[IT19] James Edward Gray, II and Bruce A. Tate. *Designing Elixir Systems with OTP*. The Pragmatic Bookshelf, Raleigh, NC, 2019.

[Jur15] Saša Jurić. *Elixir in Action*. Manning Publications Co., Greenwich, CT, 2015.

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head on over to https://pragprog.com right now, and use the coupon code BUYANOTHER2022 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of the *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-author/ today to learn more and to get started.

We thank you for your continued support, and we hope to hear from you again soon!

The Pragmatic Bookshelf

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/passotp

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date

https://pragprog.com

Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

https://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764