Extracted from:

Programmer Passport: OTP

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

grox.io Learning

Programmer Passport

OTP

Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: OTP

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

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

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

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:

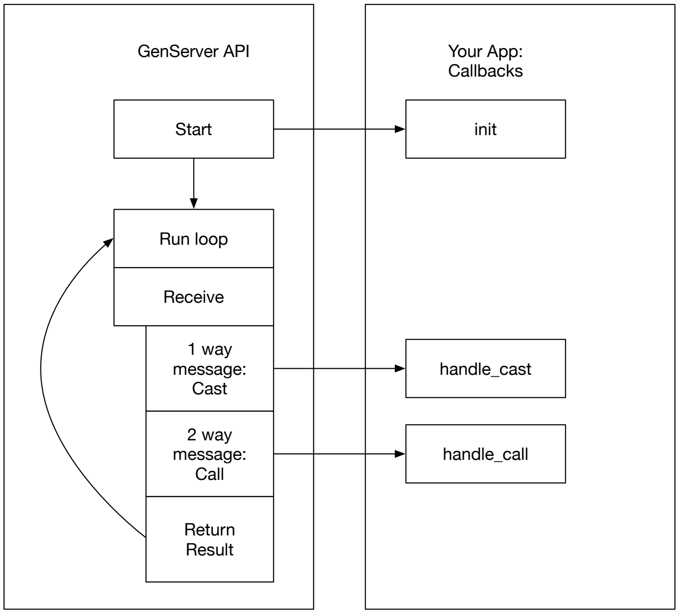
```
def start(initial_state) do
  spawn(fn -> run(initial_state) end)
end

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 ?. 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.