The Pragmatic Programmers

grox.io Learning

Programmer Passport

OTP

Bruce A. Tate

Edited by Jacquelyn Carter

Early praise for Programmer Passport: OTP

It was the best of books, it was the worst of books...

➤ **Eddy the Gerbil**
Chief Gerbil, Gerbils-r-us

We've left this page blank to make the page numbers the same in the electronic and paper books.

We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say “hello.”

Programmer Passport: OTP

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

ISBN-13: pending
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—Monthname, yyyy

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B1.0: May 13, 2020

- Programmer Passport OTP. The Basics

  This is the final chapter of the Joe Armstrong Celebration. We'll get to the core of Joe's genius, the OTP library. This amazing piece of work was created in the late 1980s, and formed the infrastructure that still keeps many of the world's programming infrastructure so robust and reliable today.

  In this chapter, we help build your intuition for OTP. We'll build an OTP-like process with two kinds of messages, call and cast. Then, we'll manage lifecycle with a process server. Finally, we'll do the same with the OTP API.

B2.0: June 1, 2020

- Programmer Passport OTP. Explore OTP Servers.

3.0: June 15, 2020

- Programmer Passport OTP. Manage Lifecycles.

B4.0: June 15, 2020

- Working with State Machines

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 be a replacement for *Designing Elixir Systems with OTP [IT19]* by James E 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 where you can write 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 try to 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 in a simpler way, 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 over simplification. 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 a 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 at all; 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 make use of 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 trasnform 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 does the work of modifying 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. It takes 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 calendar 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 calendar 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 in 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 name space, 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 custom(calculator, f, n), do: send(calculator, {:custom, f, 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, :timeuot}
  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 calculator, :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)
    (calculator) lib/core.ex:2: Calculator.Core.add/2
    (calculator) lib/boundary.ex:10: Calculator.Boundary.run/1

nil
iex> Process.alive? calculator
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.

What we need is 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 no 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

      {:custom, f, number} ->
        Core.custom(state, f, 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! Over the next few weeks, 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 *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 messages: tick, which prints the time every minute. You must also send that message every minute.

Next Time

In the next chapter, we'll dig in to 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.

We'll see you in two weeks!

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line “Book Feedback.”

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2020 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn’t apply to *The Pragmatic Programmer* ebook because it’s older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support,

Andy Hunt, Publisher

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