Extracted from:

Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenState, Flow, and Broadway

The Pragmatic Bookshelf

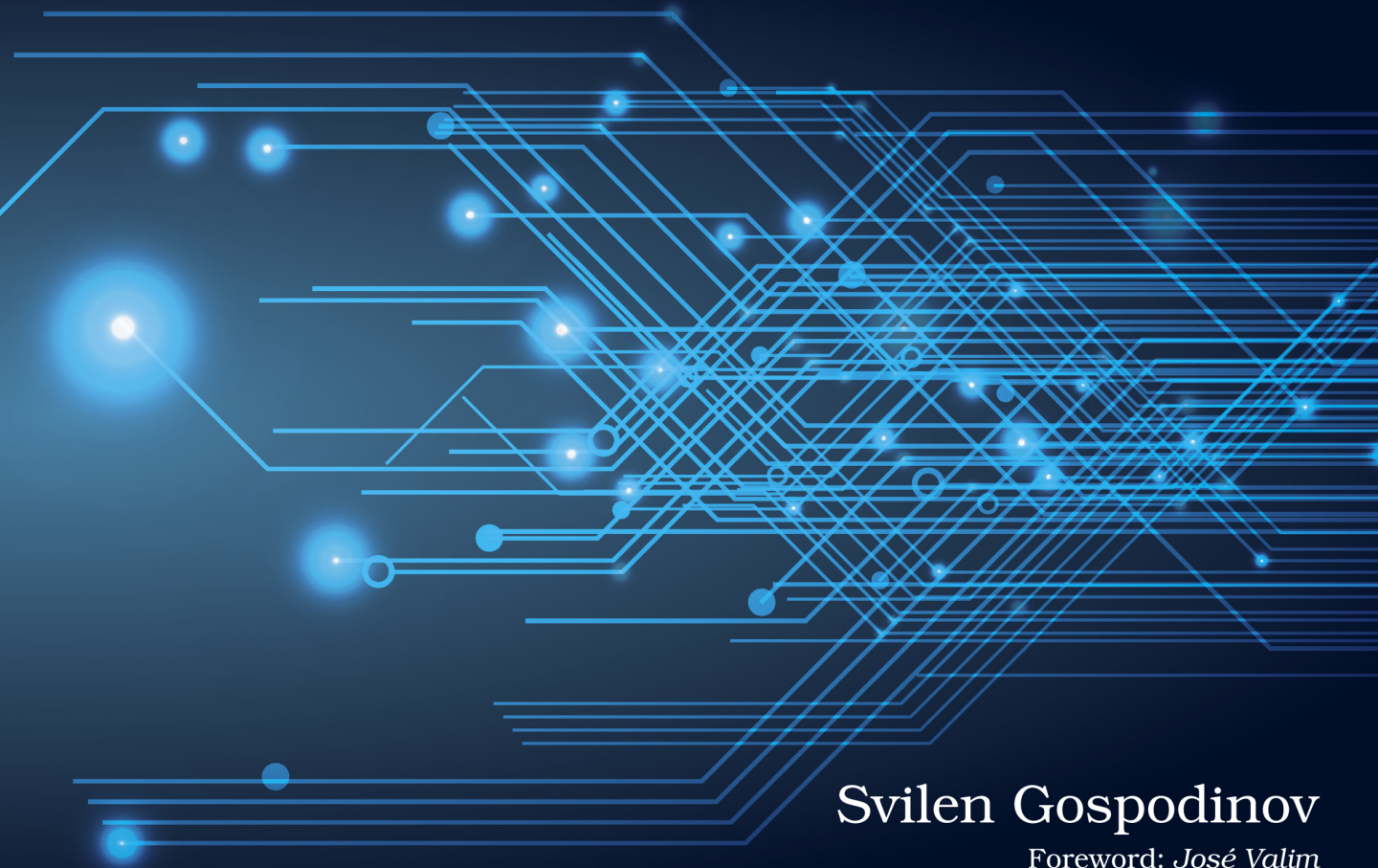
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenStage, Flow, and Broadway



Svilen Gospodinov

Foreword: *José Valim*
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

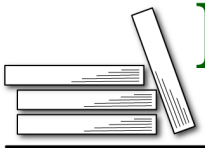
Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenState, Flow, and Broadway

Svilen Gospodinov

The Pragmatic Bookshelf

Raleigh, North Carolina



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

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

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

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

ISBN-13: 978-1-68050-819-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2021

CHAPTER 2

Long-Running Processes Using GenServer

The Task module is useful for running single async functions, but as logic becomes more complex, you will need a sharper tool. In this chapter, we'll look at how to create long-lived processes that also run in the background, but offer greater control and more flexibility.

Concurrent work often takes a long time to complete, such as when you are importing a large amount of user data from another service or relying on third-party APIs, for example. This presents several challenges when using the Task module. What if an API service goes briefly offline or a network error causes the task to fail? Do you need to increase the :timeout setting because the network is slow? How do you show progress to the user and provide better user experience?

There are also times when we want to run some code in the background continuously. In a banking application, you may need the latest foreign currency exchange rates, which means periodically retrieving the data and storing it somewhere, for as long as the application runs.

These and many other problems could be solved by using a GenServer. GenServer, which is short for *generic server*, enables us to create concurrent processes that we can interact with. Just like a Web server, it listens for requests and can respond with a result. GenServer processes also have their own state, which they keep in memory until the process exits. You also have a lot more options when it comes to supervising GenServer processes and dealing with potential errors. Since GenServer is part of the OTP, it is already available for you to use in Elixir—there is no need to install any dependencies.

In this chapter, you are going to learn how to use GenServer to create stateful processes that you can interact with. We'll look at the different supervisor strategies and see how restart options work in practice. We'll tie everything

together by building a simple but effective job-processing system, and you'll understand how to design fault-tolerant applications and manage processes using the Elixir Registry.

There is a lot to cover in this chapter, so let's get started!

Starting with a Basic GenServer

We are going to continue working on the sender project from the previous chapter. If you decided to skip that part, please see Creating Our Playground, on page ? and follow the steps to create a new Elixir project and make the required change in sender.ex. We are going to create a GenServer process from scratch and see how it compares to Task processes.

First, we're going to create a new file in the lib directory. Let's name it send_server.ex. In that file, we'll define our SendServer module like so:

```
defmodule SendServer do
  use GenServer

end
```

The use macro for the GenServer module does two things for us. First, it automatically injects the line @behaviour GenServer in our SendServer module. If you're not familiar with behaviours in Elixir, they're similar to interfaces and contracts in other programming languages. Second, it provides a default GenServer implementation for us by injecting all functions required by the GenServer behaviour.

We can verify that our code compiles by starting IEx:

```
$ iex -S mix
Compiling 3 files (.ex)
warning: function init/1 required by behaviour GenServer is not
  implemented (in module SendServer).

We will inject a default implementation for now:

    def init(init_arg) do
      {:ok, init_arg}
    end

You can copy the implementation above or define your own that converts the
  arguments given to GenServer.start_link/3 to the server state.

  lib/send_server.ex:1: SendServer (module)

Generated sender app

iex(1)>
```

You just created your first GenServer process! However, we didn't write any logic for SendServer so it doesn't do anything useful at this point. We also got a warning message by the compiler, telling us that the init/1 function is missing and was replaced by a default implementation.

The GenServer module provides default implementations for several functions required by the GenServer behaviour. These functions are known as *callbacks*, and init/1 is one of them. Callbacks are important because they allow you to customize the GenServer process by adding your own business logic to it. In the following section, you will learn about the most frequently used callbacks and some examples of how to use them.

GenServer Callbacks In Depth

The best way to learn how callbacks work is to see them in action. We are going to add some functionality to the SendServer module and introduce the most common GenServer callbacks along the way.

You can implement a callback by declaring it in the SendServer module like any other function. By doing this, you are replacing the default function that GenServer provides. This is sometimes called *overriding* the default implementation. When implementing a callback, there are two things you need to know:

- What arguments the callback function takes
- What return values are supported

We're going to cover the following callback functions for the GenServer behaviour:

- handle_call/3
- handle_cast/2
- handle_continue/2
- handle_info/2
- init/1
- terminate/2

Learning about these callbacks will enable you take full advantage of your GenServer process. There are two callbacks which we're going to skip, code_change/2 and format_status/2. We will also focus on the most frequently used return values, leaving some behind. All callbacks and return values are well documented on Elixir's HexDocs page, so feel free to go online and explore further what's possible with GenServer.

Exploring Callbacks for Behaviours

The easiest way to find out what callbacks are available for a specific behaviour is to check the documentation for the module on HexDocs. For example, you can go to Elixir's own HexDocs page.[1] Using the left sidebar, navigate to Modules and find the module name in the list of categories. GenServer can be found under "Processes & Applications". When you click on the module name, it will expand and show you a few sub-items: Top, Sections, Summary, Types, Functions, and Callbacks.

Click on Callbacks and you will see the links of the functions available for you to implement.

Remember the warning about the init/1 function that we saw earlier? Let's fix it by implementing the init/1 callback first.

Initializing the Process

The init/1 callback runs as soon as the process starts. When you start a GenServer process, you can optionally provide a list of arguments. This list is made available to you in the init/1 callback. This is a convenient way to provide some configuration details at runtime for the process. We mentioned that each GenServer process has its own in-memory state. This state is created by the init/1 function as well.

We are going to extend SendServer with the ability to send emails. If an email fails to send for whatever reason, we will also retry sending it, but we will limit the maximum number of retries. We will keep the business logic as simple as possible, so we can focus on learning how GenServer works.

First, let's add the following code to send_server.ex right after use GenServer:

```
def init(args) do
  IO.puts("Received arguments: #{inspect(args)}")
  max_retries = Keyword.get(args, :max_retries, 5)
  state = %{emails: [], max_retries: max_retries}
  {:ok, state}
end
```

We use the argument max_retries if present, otherwise we default to five retries max. We will also keep track of all sent emails using the emails list. These variables will be kept in the initial state for the process. Finally, the function returns {:ok, state}. This means that the process has successfully initialized.

1. https://hexdocs.pm/elixir

We will cover the other possible return values in just a moment, but first, let's start SendServer to make sure it works as expected.

With the IEx shell open, run recompile() and then the following code:

```
iex> {:ok, pid} = GenServer.start(SendServer, [max_retries: 1])
```

You should see output similar to this one:

```
Received arguments: [max_retries: 1]
{:ok, #PID<0.228.0>}
```

SendServer is now running in the background. There are several ways to stop a running process, as you will see later, but for now, let's use GenServer.stop/3:

```
iex> GenServer.stop(pid)
:ok
```

There are a number of result values supported by the init/1 callback. The most common ones are:

```
{:ok, state}
{:ok, state, {:continue, term}}
:ignore
{:stop, reason}
```

We already used {:ok, state}. The extra option {:continue, term} is great for doing post-initialization work. You may be tempted to add complex logic to your init/1 function, such as fetching information from the database to populate the GenServer state, but that's not desirable because the init/1 function is synchronous and should be quick. This is where {:continue, term} becomes really useful. If you return {:ok, state, {:continue, :fetch_from_database}}, the handle_continue/2 callback will be invoked after init/1, so you can provide the following implementation:

```
def handle_continue(:fetch_from_database, state) do
  # called after init/1
end
```

We will discuss handle_continue/2 in just a moment.

Finally, the last two return values help us stop the process from starting. If the given configuration is not valid or something else prevents this process from continuing, we can return either :ignore or {:stop, reason}. The difference is that if the process is under a supervisor, {:stop, reason} will make the supervisor restart it, while :ignore won't trigger a restart.