Extracted from:

Designing Elixir Systems with OTP

Write Highly Scalable, Self-Healing Software with Layers

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Designing Elixir Systems with OTP

Write Highly Scalable, Self-Healing Software with Layers

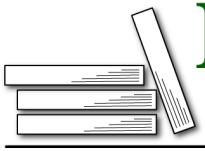
James Edward Gray, II
Bruce A. Tate
edited by Jacquelyn Carter

Designing Elixir Systems with OTP

Write Highly Scalable, Self-Healing Software with Layers

James Edward Gray, II
Bruce A. 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*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Jacquelyn Carter
Copy Editor: Jasmine Kwytin
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

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

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

ISBN-13: 978-1-68050-661-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2019

Prefer Call Over Cast to Provide Back Pressure

Intuitively, you might think that it's best to use the one-way handle_cast to send messages that don't need responses. For example, the :add_template message doesn't really need a response. We just trust that the template was added successfully. If it's not, something has gone horribly wrong. There's nothing we can do beyond crashing the server and reporting the reasons for the crash back to the user.

Interestingly, handle_cast is rarely the best option for sending messages. In this section, we'll look at one of the reasons why. They are called serializability and back pressure. Let's explore why.

As you probably know, each Elixir process has a message queue. We'll call it the mailbox. Unlike a physical mailbox, Elixir processes only receive messages from it; they don't send from the mailbox. Like a true mailbox, if the receiving process for a given message is struggling, the mailbox can overflow, often leading to severe problems that are hard to debug.

A good example is the Elixir logger. If your production code is sending log messages quicker than the logger can handle them, either because the sender is logging too many log requests or because the logger's disk I/O is somehow compromised, we don't want the logger to immediately stop logging messages.

The Elixir logger has an excellent solution for this problem. It's called selective back-pressure. That means that when the logger gets into trouble, it will detect this problem and start slowing the clients down by switching from cast to call.

Making the logger's client wait for every request to finish before sending the next one relieves the pressure on the logger itself by slowing down the flow of messages. If the logger still can't keep up, it announces this failure as a log message and begins to discard messages until the logger gets to a more manageable threshold.

Let's dive into some specific details. We'll start with configuration.

Users can configure options to represent thresholds. These thresholds specify when a healthy logger becomes sick because its message logger gets too long. Two of these thresholds specify when to go from cast to call, or when to start shedding messages.

Users can also configure thresholds defining when the system goes from sick to healthy. When an unhealthy system has a message queue that shrinks

below these thresholds, the logger can stop discarding messages, or go back to cast from call.

The logger code then uses that configuration to implement three different modes to implement the cast, call and shedding modes. They are called :async, :sync, and :discard, respectively.

Now, let's look at the specific Elixir implementation. As a general metric for system health, sometimes it helps to look at the number of messages in a processes mail box. Here's the code that does that job:

```
defp message_queue_length() do
  {:message_queue_len, messages} = Process.info(self(), :message_queue_len)
  messages
end
```

Process.info(self(),:message_queue_length) does the magic. It returns an integer value that is the number of messages in the queue. The logger can then make use of it.

Now we can see how the logger switches modes. In logger/config.ex, the logger computes the right mode, like this:

```
case mode do
  _ when messages >= discard_threshold -> :discard
  :discard when messages > keep_threshold -> :discard
  _ when messages >= sync_threshold -> :sync
  :sync when messages > async_threshold -> :sync
  _ -> :async
end
```

This snippet computes the mode given the message queue length in messages. The thresholds in this function all come from the logger configuration. These thresholds work in pairs. One threshold in each pair marks the transition from healthy to sick, and one marks the transition from sick to healthy.

We shed messages if the function is greater than discard_threshold; we stay in discard mode if we stay above the keep_threshold. Otherwise, we switch to sync mode if we are over the sync_threshold, and stay in that mode if it's already in sync mode and the messages are above the async_threshold. If none of those things are true, we're healthy, so we send async.

Now, we can compare the configured mode with the computed one, like this:

```
def handle_event(_event, {state, thresholds}) do
    %{mode: mode} = state

case compute_mode(mode, thresholds) do
^mode ->
  {:ok, {state, thresholds}}
```

If the mode matches the mode that was configured, do nothing. Otherwise:

```
new_mode ->
  if new_mode == :discard do
    message =
      "Logger has #{message_queue_length()} messages in its queue, " <>
        "which is above :discard_threshold. Messages will be discarded " <>
        "until the message queue goes back to 75% of the threshold size"

    log(:warn, message, state)
  end

  if mode == :discard do
    log(:warn, "Logger has stopped discarding messages", state)
  end
```

If things are very bad and we're beyond the discard limit, we set the :discard state so we can shed messages until we're healthy. We log a message to tell the user we're no longer logging, pending improvements.

All that remains is to set the new mode in the logger, like this:

```
  state = persist(%{state | mode: new_mode})
  {:ok, {state, thresholds}}
end
```

We set the new mode and let the logger lose. Let's see :discard in action.

```
def __should_log__(level) when level in @levels do
  ...
  if compare_levels(level, min_level) != :lt and mode != :discard do
    {level, config, pdict}
  else
    :error
  end
  ...
end
```

In a function called __should_log__ we check the mode for :discard. If it's set, regardless of log level, we'll return :error.

In logger.ex, the bare log looks like this:

```
def bare_log(level, chardata_or_fun, metadata \\ []) do
  case __should_log__(level) do
    :error -> :ok
    info -> __do_log__(info, chardata_or_fun, metadata)
  end
end
```

If the mode is :error, we do nothing, shedding the messages. Otherwise we call *do_log*, a long function which eventually does this:

```
notify(mode, {level, Process.group_leader(), tuple})
```

We're finally at the magic moment. We choose call or cast to handle back pressure. At the very bottom of logger.ex, you'll see these functions:

```
defp notify(:sync, msg), do: :gen_event.sync_notify(Logger, msg)
defp notify(:async, msg), do: :gen_event.notify(Logger, msg)
```

This means Elixir will log messages as a call (sync) or cast (async).

Here's the point. If your code uses handle_call instead of handle_cast, you don't need to worry as much because you can only send messages as fast as your server can process them. It's a great automatic governor on a server.

Rarely, you'll want to use cast messages to start multiple workers at once, or to notify multiple workers simultaneously. Try to be judicious with this approach, though.

Back pressure is one reason to avoid cast messages. It's not the only reason, though. Let's look at the next one.

Extend Your APIs Safely

So far, we've strongly advocated building many small components and managing those components through dependencies. When this strategy is working well, it simplifies your job by limiting the scope of what you need to understand to make any given change.

This strategy can go to a special hell fueled by cascading dependencies in a hurry, if you're not careful with how you build your APIs. Specifically, maintaining a healthy ecosystem is difficult if each release of an API breaks compatibility to old versions of the API. Breaking changes have several different forms:

- An API can add requirements to input parameters such as adding a new required field to our Quiz.

- An API can change the shape of the output such as changing all of our quiz functions to `{:ok, quiz}`.
- An API can change their behavior in unexpected ways such as treating an `amount` as dollars rather than cents.

Let's look quickly at an approach to APIs that will improve compatibility as you improve the various independent components in your system. We'll honor three rules.

Don't Add New Requirements to Existing APIs, Only Options

Many beginning developers tend to validate all arguments for a remote API. Then, as those APIs need to be extended, they require those as well. There's a problem with that approach.

If servers provide requests that require all parameters, each new parameter means you'll have to upgrade the client and server simultaneously. With just one client and one server component, that strategy may seem viable but as dependencies like this cascade through a system, upgrades get exponentially more difficult. Then, you lose all of the advantages you were seeking by building decoupled components in the first place.

If you want to extend an API, extend it with options. Then, servers can provide new API functionality to the same endpoints without requiring all clients to change. Later, clients can upgrade to take advantage of these new options.

Ignore Anything You Don't Understand

The "no new requirements" rule pertains to public-facing APIs. There's a similar rule for dealing with data. Ignoring everything you don't understand makes it possible to slowly add new fields, request options that may not yet be supported, and to upgrade your systems incrementally.

These first two rules work together well. For example, say there's an export program that's expecting a fixed set of fields representing a product. The server makes new fields optional. The server does two things:

- It ignores optional fields that are empty
- It ignores fields it doesn't know about

This way, the system will function well through change. It doesn't matter which system deploys first. The server exports the new fields only when both the client and server provide them. This is the ideal behavior.

Don't Break Compatibility; Provide a New Endpoint

Here's the punch line. Don't break users of an endpoint, ever. Rather than extending an existing endpoint in incompatible ways, provide a new endpoint to do the new thing. Modern languages have many ways to scope and delegate functions, and these features give us infinite flexibility with naming.

We'll go one step further. Server endpoints are not the only APIs that could stand to benefit from this approach. Everyday function libraries break these rules every day. There's a concept called semantic versioning that says minor versions are compatible, and major versions are possibly incompatible. These rules might look wise, but a far better way is to adopt rules that don't break compatibility in the first place.

It's been a busy chapter, and it's time to wrap up.

Wrap Your Core in a Boundary API

In this chapter, we left our safe bubble of the functional core and ventured out to the real world to deal with state, processes, and communication between components. Here's how we did it.

To begin our exploration, we dove into some techniques to handle composition with inputs and outputs that were less certain. We looked at ways to transform executing errors to data. We also encountered composition using `with`.

Next, we built a server layer in two pieces, the `QuizManager` and the `QuizSession`. We used a `GenServer` to build a quiz and another to let a user take a quiz. The server layer used `start_link` and `handle_call` functions to encapsulate state and handle communication between processes. We eschewed `handle_cast` to handle back pressure issues.

We built validations to make sure our servers will work on consistent data, and then we built an API layer to access our server layer in a convenient way.

It's all starting to come together, but we know our boundary layer supports only one running quiz at a time. In the next chapter, we'll build a dynamic supervisor to allow each user to run a process per quiz. We'll also build a quiz manager to let users build and store multiple quizzes.

You've reached the crux of the book, so turn the page and let's get busy!