Extracted from:

Adopting Elixir

From Concept to Production

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Adopting Elixir

From Concept to Production

Ben Marx, José Valim, Bruce Tate

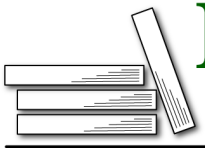
edited by Jacquelyn Carter

Adopting Elixir

From Concept to Production

Ben Marx
José Valim
Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

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
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Jasmine Kwityn
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*.

Printed in the United States of America.
ISBN-13: 978-1-68050-252-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2018

A common story we've heard from Elixir newcomers is that deployment was particularly challenging. If you're going to successfully adopt any new language, you need to be able to get that beautiful, powerful code onto production servers, but that's not enough. You need to do so reliably, without downtime, and with the ability to gracefully recover should things go wrong.

To illustrate this point, meet Tetiana Dushenkivska. She's a Ruby developer who adopted Elixir early on and was the keynote speaker at ElixirConf Europe 2017. She mastered Elixir concepts when we had one-tenth of the available learning resources that we do today:

> ***Bruce:*** *How was your first encounter with Elixir?*
>
> ***Tetiana:*** *I was happily working with Ruby, when a colleague shared his finding, Elixir. At first, I didn't get too excited. I was thinking: "Those languages and frameworks keep popping up and I don't have time right now to learn another language." Regardless of that thought, I still took a quick look. At first glance it looked much like Ruby, but soon enough I started to understand that maybe it looks like Ruby, but it doesn't behave like Ruby. The more I read about Elixir, however, the more I wanted to keep learning about it. The first thing to motivate me to start building something in Elixir was the ability to do things concurrently. Then I thought: "Oh, this language looks VERY interesting, I should definitely learn more about it."*
>
> ***Bruce:*** *How did you move forward from there?*
>
> ***Tetiana:*** *Programming Elixir by Dave Thomas was my introduction to Elixir, together with the official getting started guide on the website.*
>
> *The concept of functional languages resonated quickly with me. When I was studying electronic engineering at university I learned about signals and how they're transformed from one shape to another. Functional programming is somewhat similar. A signal is like data in functional programming which when put through some filters, or functions in Elixir, results in a new signal. You can't rebind a signal.*

You just have an input signal and when it comes out of the black box it's a new signal. And every time we pass the same input to the black box, we get the same output.

Bruce: *Have you had hiccups or roadblocks along the way? How did you overcome them?*

Tetiana: *I would say deployment was hard. There are lots of ways to "build releases" and it took a bit of time to research and find a way that would work for me. Thankfully, the Elixir community is a great place to ask questions. Michał Muskała pointed me in the right direction, which helped me solve the deploying applications challenge.*

The Elixir community is doing a great job helping people who are stuck, to solve their problems. I am glad that people who have learned something are happy to share their knowledge, so that everyone else can learn faster.

Tetiana is not alone. For new languages, the deployment story almost always takes time to crystallize. We've heard story after story from happy early adopters of many emerging languages identifying deployment as a pain point. The same is true with Elixir.

Even so, we're starting to see some overarching strategies and contenders begin to surface in the deployment space. In this chapter, you'll learn about these emerging technologies. Elixir developers are moving beyond the Mix tool for deployment, and they're formally defining releases using tools such as Distillery. Then, rather than focusing on hot-code-swapping, they're using a technique called blue-green deployments. We'll walk you through how these tools and techniques work. That's what we'll focus on, but there are a few topics we won't cover.

In this chapter, we won't discuss any particular stack. We won't give you specific recipes for deploying to Heroku or using Docker containers, automating with Chef, or managing your cluster with Kubernetes. In fact, we've seen all of those options being successfully used to run Elixir systems. Instead of giving a way-too-thin blow by blow for each option out there or anointing a winner when the market has yet to decide, we're going to focus on the Elixir bits. After all, this book is called *Adopting Elixir*. Let's get to it.

Deploying with Mix

The emergence of deployment tools within git and Elixir's basic tooling makes it pretty simple to stand up a dead-simple deployment strategy for a single machine. The easiest way to run an Elixir application in production is by fetching or pushing the source code to your servers and calling:

```
$ MIX_ENV=prod mix run --no-halt
```

mix run will compile and start the current application and all of its dependencies. --no-halt guarantees Elixir won't terminate just after the application is booted. Phoenix is similar. Instead of mix run --no-halt, you will execute mix phx.server, still setting the Mix environment to "prod".

MIX_ENV=prod ensures your application is running in the production environment with the relevant configurations. One of those configurations is the :start_permanent option, which you will find in your mix.exs file:

```
start_permanent: Mix.env == :prod
```

Each application runs as :temporary or :permanent. If a permanent application shuts down, it automatically causes the whole VM to shut down too, so something else can restart it.

Here's the problem :permanent was designed to solve. Say you were to start a Phoenix application without setting :start_permanent. Suppose its top-level supervisor has to restart its children multiple times in a short period due to a fault. If the supervisor exceeds the amount of restarts allowed in a timeframe, it terminates, causing your application to also terminate. If your application has not been set to permanent, the remaining applications will continue running without your Phoenix app, so you can't accept any more requests. In development, that's likely fine, but in production, you want to shut the VM down so something else can restart it cleanly.

If you are using a Platform-as-a-Service (PaaS) offering such as Heroku for your deployment, it's likely using mix run or a similar task for starting your applications. The advantage of using Mix in production is that you can rely on the same tooling that you use for your development. All you need is the source code. It is an option that works well for very simple deployments.

As soon as you want to leverage some of the more advanced capabilities that the VM offers you, this approach starts to fall apart. That's what we will do now. We will add some nuts and bolts to our Mix deployment and show it quickly becomes unmanageable.

The –no-compile Flag

Our first modification will add support for a multi-server deployment. We'll compile once and push that code to each server.

Mix was designed primarily as a development tool. When you execute the mix run task, Mix checks to see whether your code requires compilation. Since

you're deploying to multiple servers, you may want to compile your application only once and not per server. One option is to have a build machine that exists specifically to build the deployment artifact. When done, the build machine can push the artifact to your production servers or your production servers can fetch it directly from the build machine. Let's see how to construct this artifact with Mix.

On your build machine, you'd run:

```
$ MIX_ENV=prod mix compile
```

And in production:

```
$ MIX_ENV=prod mix run --no-halt
```

Mix works by tracking the modification times of source files and of the generated beam files. There's a problem with this approach. Moving your files changes your modification times, so the mix run task notices the changed times and recompiles, defeating the whole purpose of the build server!

There's a simple fix. You can pass the --no-compile flag when starting in production, like this:

```
$ MIX_ENV=prod mix run --no-halt --no-compile
```

It is one small change, but the first of many. There's more work to do.

The –no-deps-check Flag

There's another Mix downside. It requires the whole source code tree and its dependencies in production, so if you have git dependencies, Mix will require git on the production server. To solve this problem, you'd pass the --no-deps-check flag to disable dependency checking.

On your build machine, you would run:

```
$ MIX_ENV=prod mix compile
$ rm -rf deps/*/.git
```

And in production:

```
$ MIX_ENV=prod mix run --no-halt --no-compile --no-deps-check
```

The previous command still requires dependency source code but does allow removal of any version control metadata from our dependencies. That in turn reduces the size of your production artifacts. The problem is solved, but wait, there's more.

VM Configuration

On production, you'll often want to fine-tune both Elixir and the VM. You can handle some of this tuning in the config/config.exs file. For example, you can choose the proper Logger level by setting:

```
config :logger, :level, :warn
```

That change will show log entries at the :warn level of severity or stronger. That's not the only configuration you'll encounter because some configuration happens when the VM boots.

For example, many applications may want to tweak +K and +A flags for production. +K true enables kernel pooling, which provides an OS-specific I/O event notification system. +A increases the async thread pool, which is a group of threads started by the VM responsible for all of the I/O work done by your code. By default the async pool has 10 threads, but if you are doing a lot of I/O, you likely want to increase that count to about 8 threads per core. If you have 8 cores, 64 threads is a better starting point.

Unfortunately the mix run command can't receive VM configurations because you need to specify those commands when the VM starts. The solution is to invoke mix through elixir, like this:

```
$ MIX_ENV=prod elixir --erl "+K true +A 16" -S \
>   mix run --no-halt --no-compile --no-deps-check
```

By using the elixir command-line script, you've eliminated the problem. You can simply pass VM commands with the --erl flag, using the -S flag to instruct elixir to run the mix command available in your system. Those aren't the only flags to consider, though. If you want to run distributed Erlang, you'll need still more flags.

This kind of application startup complexity is common for running all but the simplest applications. You can try to juggle startup parameters in this way, but you'd be playing with fire because it's an error-prone approach.

In case you're not yet convinced, let's continue pushing the boundaries and see how far we can go.

run_erl and heart

Erlang is more than the standard library and virtual machine. As you might expect after thirty years of history, it ships with many tools for successfully running Erlang in production. Two of those tools are run_erl and heart.

Managing Shared I/O

run_erl[1] helps you manage the standard input and output of a program. The Unix tool redirects all output to log files. For those so inclined, there's a similar Windows tool named start_erl.[2]

run_erl expects a pipe name, the log directory, and the command to execute. *Remember the log directory must be created before you invoke run_erl, otherwise it will silently fail.* Let's give run_erl a try:

```
$ mkdir ./log
$ run_erl ./loop ./log "elixir -e 'Enum.map Stream.interval(1000), &IO.puts/1'"
```

This command runs an Elixir script that prints a number to standard output every second. Assuming you've created a log directory beforehand, you'll see a new file at log/erlang.log.1 with the convenient sequence of logs. run_erl automatically rotates logs every 100KB, keeping the last four files.

In production, run_erl is usually executed with the -daemon flag. Let's give it a try but this time with iex:

```
$ run_erl -daemon ./iex_sample ./log "iex"
```

Here we used run_erl to start iex as a daemon. Notice we have no access to iex though. That's where to_erl comes in.

The first run_erl argument is a named pipe. The pipe lets us interface with any running program via the to_erl tool, like this:

```
$ to_erl ./iex_sample
Attaching to ./iex_sample (^D to exit)

iex(1)> 1 + 2
3
```

to_erl allows us to interact with any system through standard I/O. If you want to shut down the VM, you can invoke System.stop(), which gracefully shuts the Erlang system down, stopping all applications with their respective supervision trees. You can directly invoke System.stop() in your IEx session or send it via to_erl:

```
$ echo "System.stop()" | to_erl ./iex_sample
```

If your application requires specific shutdown instructions, you can send them as well:

1. http://erlang.org/doc/man/run_erl.html
2. http://erlang.org/doc/man/start_erl.html

```
$ echo "MyApp.clean_shutdown()" | to_erl ./my_app
```

While run_erl provides logging and log rotation, to_erl can be an excellent tool for debugging live systems. Teams running Elixir in production should definitely account for those tools in their stack. Let's continue to build on our mix run commands, adding run_erl:

```
$ mkdir ./log
$ run_erl -daemon ./my_app ./log \
>   "MIX_ENV=prod iex --erl '+K true +A 16' -S \
>   mix run --no-halt --no-compile --no-deps-check"
```

We are now using iex instead of elixir to boot the app, allowing us to use to_erl and interact with our application at any moment. Note we are nesting single and double quotes. Pay attention. With each step, the blob continues to grow.