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

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

In the past ten years, programmers have made tremendous strides in craftsmanship. Collectively, we're paying more attention to code structure, testing, types, and more because *these concepts matter*. New adopters might not have enough experience to completely control every implementation detail, but they can embrace code consistency.

Whether you're working on a big team or a young team, you'll want to establish a baseline so that your code stays *fresh*, and the coding stays *fun*. It's natural that technical debt accrues more quickly as inexperienced programmers learn the best ways to write code that's easy to understand and maintain. That's why code standards are so crucial. Churn without boundaries is chaotic; churn within a framework is annoying but tolerable.

In this chapter, we're going to walk you through code quality. We'll provide some guidance in five primary areas:

Coding standards
The Elixir community has settled on coding standards so code looks the same not just from one module to the next, but also one project to the next.

Types
Type annotations provide documentation for the programmer and information for tools that help you find bugs.

Documentation
For your public-facing modules, documentation will help you describe what's happening in your codebase so others will know how to best use your code.

Tests
: Testing for functional languages is different. The focus on immutability will let you build shorter, simpler tests.

Reviews
: Fungus grows in the dark. Each different set of eyes is like sunlight into a damp, dark corner, improving quality and adding accountability.

Many of the tools we'll show you are not just guidelines you have to police yourself. They're *automations*. That way, you can continuously get many of these benefits with a fraction of the cost of manual intervention. When you commit to guidelines throughout your organization and as part of your whole lifecycle, from setting expectations when you hire your first developer to maintaining code that's already in production, you'll profit.

Before we get started, let's do one bit of housekeeping. You may be asking yourself, "How much is too much?" We don't have an answer for you. Which tools you install will depend on the size and experience levels for your team, the size and complexity of your codebase, and your affinity for the approaches we suggest. We'll offer two pieces of advice:

- You almost certainly don't want to implement all of this at once.
- If it feels good, do it more. If it hurts, stop.

None of the authors on this team use all of the tools in this chapter. We all select the best tools for our teams and circumstances. We suggest you do the same. With that guidance in mind, let's get to work. We'll start with automated coding standards.

Coding Standards

Every programming language has built-in idioms and practices that collectively shape the look and feel of a codebase's structure and contents. That's coding *style*. Good style is especially important to adoption because it reduces friction between developers and makes a collective codebase easier to read.

As you can imagine, some tools can help you manage many of these elements automatically. Such tools are typically divided in two categories:

- *Code formatters* focus on code layout concerns, such as indentation, use of spaces and newlines, line length, and the like.

- *Linters* focus on code quality and code structure concerns that go beyond layout, such as function and variable names.

The Elixir community embraced linters years ago but formatters are a more recent addition. Let's start with them.

Code Formatters

You and your teams have probably already had one or more heated discussions about code style. Should you use tabs or spaces? Should you add spaces after commas or not? To mature, all new language communities must go through these discussions at some point.

Even when teams are in perfect agreement and choose a style guide that already exists in the community, enforcing such guidelines requires constant effort during development as well as code reviews. To make matters worse, as your company grows, each new developer needs to get acquainted with the house rules, and that may take some time getting used to.

The Elixir team has heard those complaints loud and clear. To address them, they have recently announced that Elixir will include a Code Formatter starting with release v1.6. This new feature can format your code using a consistent style. Assume a file like this one:

```
defmodule HelloWorld do
  def greet( first, last ) do
    name = first<>" "<>last
    IO.puts "Hello #{ name }!"
  end
end
```

mix format will rewrite that code to:

```
defmodule HelloWorld do
  def greet(first, last) do
    name = first <> " " <> last
    IO.puts("Hello #{name}!")
  end
end
```

We strongly advise all teams and companies to adopt Elixir's code formatter. With it, your team no longer needs to worry about small style decisions that sap productivity. They can now focus on the issues that matter.

The Elixir formatter is also a great teaching tool. If a new developer joins your team and they are not yet familiar with Elixir, they can learn how to write idiomatic code that is consistent with your company and the whole community by simply running the formatter as they program. They get immediate feedback and grow more confident their code will fit right in.

Elixir's code formatter provides as little configuration as possible. A formatter with too many options would lead to many different sets of rules, causing fragmentation inside companies and in the community. Instead, the community gets greater consistency and new hires or open source contributors know exactly what to expect.

Finally, note that the formatter will never change the code semantics. The formatter guarantees any code before and after formatting will behave exactly the same. This guarantee implies Elixir won't be able to handle all code style rules such as `underscored_names` versus `camelCase` because such a change would impact the meaning of the code.

Luckily, the Elixir community provides other tools, such as linters, to handle all other concerns that the formatter cannot.

Credo: Linter as Teacher

Linters are important because they automate tedious style and code quality checks. Linter rules don't exist in a vacuum; the coding rules come from the language community. As the language evolves, so does the linter. One of the most useful libraries for code consistency you'll find is René Föhring's Credo.[1] It's a linter like Ruby's Rubocop or JSLint for JavaScript, but as it proclaims in the tagline on GitHub, it's a linter "with a focus on code consistency and teaching." That aim is what makes Credo so interesting.

With a standard linter, you might get some warning or suggestion, and instead of understanding the issue you make the change and move on. The linter helps improve the quality of the code, but it doesn't give the developer much context as to why these changes are necessary.

Credo, too, tells you "what," but also answers "why."

Let's look at a contrived simple Mix application. Elixir has a package manager called Hex.[2] Every time you add a dependency to your project, Hex is responsible for downloading it. Many packages in Hex are named after the package domain followed by an `_ex` prefix, such as `html_sanitize_ex`, `kafka_ex`, and many others.

Our application will "hexify" library names by appending `_ex` to the given string unless one already exists. Let's create it:

```
$ mix new belief_structure
```

1. https://github.com/rrrene/credo
2. https://hex.pm/

The BeliefStructure module in lib/belief_structure.ex defines the main hexify function:

ensuring_code_consistency/belief_structure/lib/belief_structure.ex
```
defmodule BeliefStructure do
  def hexify(package) do
    case String.ends_with?(package, "ex") do
      true  -> package
      false -> BeliefStructure.Hexify.name(package)
    end
  end
end
```

And in lib/belief_structure/hexify.ex, you'll find this:

ensuring_code_consistency/belief_structure/lib/belief_structure/hexify.ex
```
defmodule BeliefStructure.Hexify do
  def name(package) do
    package(package)
  end

  defp package(package) do
    package <> "_ex"
  end
end
```

As you can guess, it works like this:

```
iex(1)> BeliefStructure.hexify("warden")
"warden_ex"

iex(2)> BeliefStructure.hexify("aws_ex")
"aws_ex"
```

It works fine, and the code looks OK, but let's run Credo to check our code. First, add Credo to your deps:

ensuring_code_consistency/belief_structure/mix.exs
```
defp deps do
  [
    {:credo, "~> 0.8.8", only: [:dev], runtime: false}
  ]
end
```

From the command line, run the command mix credo. Credo has multiple levels of warnings and suggestions. If you'd like to see all levels, run mix credo --strict, which should return the following output:

```
Software Design
?
? [D] ? Nested modules could be aliased at the top of the invoking module.
?       lib/belief_structure.ex:6:16 (BeliefStructure.hexify)
```

```
Code Readability
?
? [R] ? Modules should have a @moduledoc tag.
?       lib/hexify.ex:1:11 (BeliefStructure.Hexify)
? [R] ? Modules should have a @moduledoc tag.
?       lib/belief_structure.ex:1:11 (BeliefStructure)

Please report incorrect results: https://github.com/rrrene/credo/issues

Analysis took 0.1 seconds (0.00s to load, 0.1s running checks)
5 mods/funs, 2 code readability issues, 1 software design suggestion.
```

Credo reports improvements over a wide range of categories. While those suggested improvements may quickly resonate with experienced Elixir developers, new adopters may not understand what they mean or why they matter. That's why Credo goes a step further. Let's try it out (note that your output might vary based on the particular version of Credo you're running):

```
  mix credo lib/belief_structure.ex:1:11

?
?   [R] Category: readability
?    ?  Priority: normal
?
?       Modules should have a @moduledoc tag.
?       lib/hexify.ex:1:11 (BeliefStructure.Hexify)
?
?     __ CODE IN QUESTION
?
?      1 defmodule BeliefStructure.Hexify do
?                  ^^^^^^^^^^^^^^^^^^^^^^
?      2   def name(package) do
?      3     package(package)
?
?     __ WHY IT MATTERS
?
?       Every module should contain comprehensive documentation.
?
?       Many times a sentence or two in plain english, explaining why
?       the module exists, will suffice. Documenting your train of
?       thought this way will help both your co-workers and your
?       future-self.
?
?       Other times you will want to elaborate even further and show some
?       examples of how the module's functions can and should be used.
?
?       In some cases however, you might not want to document things about
?       a module, e.g. it is part of a private API inside your project.
?       Since Elixir prefers explicitness over implicit behavior, you
?       should "tag" these modules with
?
```

```
?           @moduledoc false
?
?       to make it clear that there is no intention in documenting it.
?
?    __ CONFIGURATION OPTIONS
?
?       You can disable this check by using this tuple
?
?         {Credo.Check.Readability.ModuleDoc, false}
?
?       There are no other configuration options.
```

It's concise and clear. With such an explanation, anyone in the organization could act on it. If your organization doesn't use these tags, or at least uses them sparsely, you can add {Credo.Check.Readability.ModuleDoc, false} to your .credo.exs and supress such warnings as suggested by Credo here.

Credo has significantly helped Elixir adoption at Bleacher Report for all of the reasons just mentioned. They adopted Elixir around October 2014 when the language was quite young. Credo didn't yet exist and style guides were just developing. Each app had its own personality but since the team at Bleacher Report was all learning Elixir, there were varying degrees of technical debt and experimentation. This state of constant churn made it harder to switch between apps and no one was exactly sure how to style or unify the apps in development.

As the community grew, tools started to emerge and they nudged the Bleacher Report team in the right direction. Most of the tools came in the form of documents outlining advice or coding suggestions. Using such manuals, developers can often miss these stylistic and design inconsistencies, or worse will only focus on these types of issues during a code review and miss critical logical errors or regressions. With code formatters and linters such as Credo, most of those concerns are automated away.

Other elements of code consistency go much deeper. In the next section, we will work on consistency of types.