Extracted from:

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

This PDF file contains pages extracted from *Functional Web Development with Elixir, OTP, and Phoenix*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

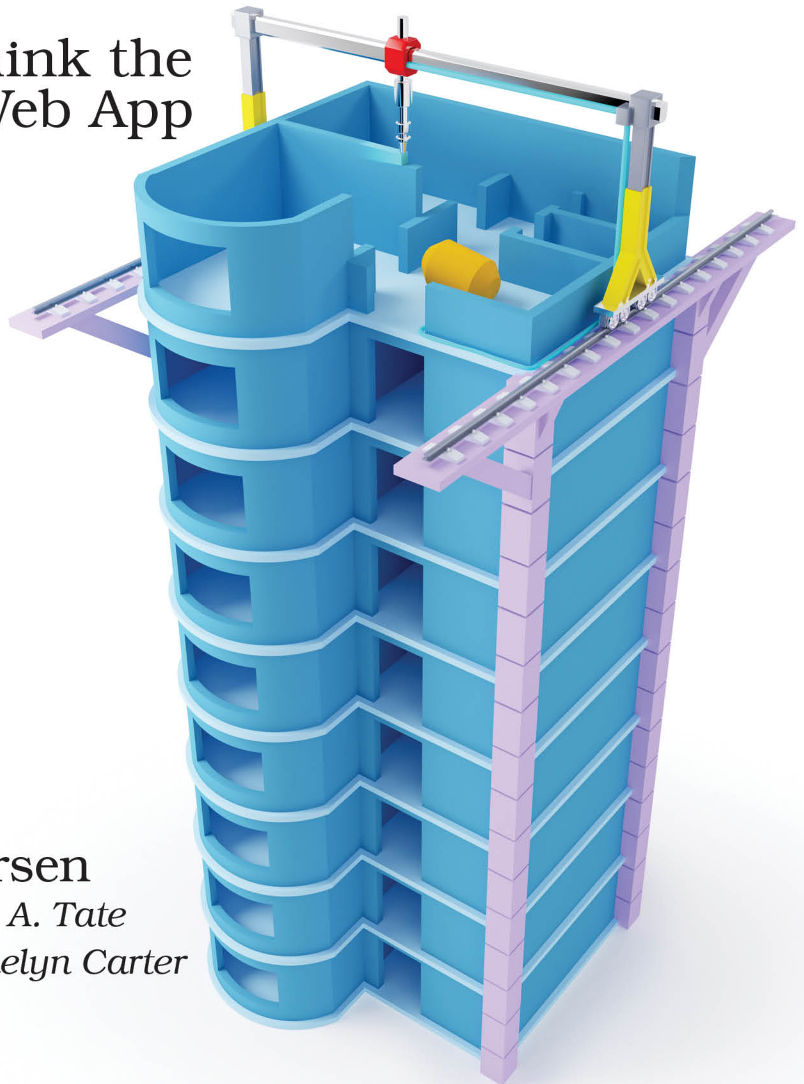
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App



Lance Halvorsen

Series editor: *Bruce A. Tate*

Development editor: *Jacquelyn Carter*

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

Lance Halvorsen

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

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

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

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-243-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

Phoenix is a great web framework. It's fast, really fast. Its components are familiar and easy to work with. Phoenix is lightweight, modular, and explicit. There's almost no hidden magic. That's a big boost for maintainability.

Frameworks are nearly ubiquitous in web development today. For either the front end or back end, almost everyone uses some form of framework to build web applications.

There's good reason for this. Frameworks get us up and running quickly. They remove the need to reimplement common tasks for every project—routing, handling request parameters, and the like. Frameworks let us focus on our individual application's behavior instead of repetitive tasks.

The slippery slope is that frameworks make it all too easy to tangle the framework components and the application together in ways that really hurt us.

Elixir Applications let us get around this in an elegant way. Phoenix itself is an Application. The game engine we built in Parts 1 and 2 of the book is also an Application.

Our task in Part 3 of this book is to create a web interface with Phoenix for our game. We're going to use the Phoenix and game engine Applications as building blocks to create a third Application that will keep the Phoenix interface separate from the game in a way that will make our job trivially easy.

Frameworks

Framework components represent what is common to all web applications. That's why the framework creators extracted them out into the framework. This is a great boon to developers because we don't need to solve the same problems over and over again. Things like routing requests to the right handler functions, getting the request parameters, handling response templates, setting cookies—the framework takes care of all that for us. The framework components make it easy to interact with the business logic over the web. They make up the web interface for the application.

The business logic is unique to each application. This is the part that we can't extract into a common framework. It's what makes our application do interesting things and gives it value. It's the most important part to us, because the success or failure of our application depends on how well this works.

But there's a serious, hidden-in-plain-sight problem here. We're so accustomed to it that we hardly even notice.

Coupling

The way we normally build business logic with a framework is completely backward. We create application behavior by adding more pieces of the framework—routes, controllers, models, and views. Each new model or controller we add contains a bit more logic. This mixes our business domain with the domain of the framework, and it couples the two inextricably and forever.

Why is that a big deal? We can't easily reuse the business logic with another interface. We can't test our business logic in isolation, outside the context of the framework code.

Let's say we wanted both a web interface and a Nerves device version of Islands. If we didn't have a separate Application for Islands, we would need to completely reimplement the business logic for each interface.

Whenever we need to send an HTTP request to test a business rule, an alarm should go off at our workstation. Business rules should be completely separate from how we handle HTTP requests. Yet this is how we've been trained to test web applications.

This is why upgrading a framework to a new major version can sometimes be so painful. It's also why switching frameworks entirely seems like a herculean task. The way we normally work with frameworks makes this pain almost inevitable.

Which brings us to one of the most important points in the entire book.

Phoenix Is Not Your Application

It's important to think about how we got into this situation, so we know how to get out of it.

The way we talk about web applications gets us in trouble right away. We say, "I'm building a Rails app" or an Ember app or a Phoenix app or an Elm app.

But that's not true. What we're really doing is building a chat app, or a banking app, or a game called Islands, with a Phoenix interface or an Ember interface or an Elm interface.

The problem is deeper than that, though. There are a number of ways to look at it, but this resonates most with me. ORMs lead us directly into this coupling of business logic and framework components.

ActiveRecord models in Rails offer the clearest example of this, but the same idea applies across many frameworks. Let's say we are working in a domain

in which one of the entities is a bicycle. We could begin modeling this with a plain Ruby class:

```
class Bicycle
  # We define bicycle-specific properties and behavior here.
end
```

We might define bicycle properties here like wheels, handlebars, pedals, and brakes. We could also define behaviors like pedaling, steering, and braking.

Rails tends to push us toward putting domain models, like our Bicycle class, in a database. ActiveRecord makes this very easy. We just have a model class inherit from ActiveRecord::Base:

```
class Bicycle < ActiveRecord::Base
  # Suddenly, bicycle behavior is mixed with database behavior.
end
```

With this small change, everything is different. Our Bicycle class suddenly knows a lot more than just bicycle things. It knows how to connect to a database, read from and write to a table, validate data, perform transactions, generate queries, and a whole host of other things.

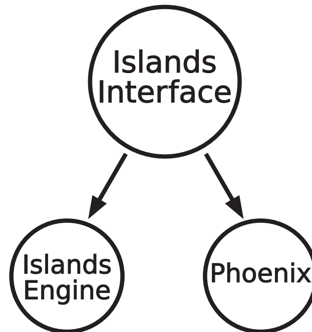
Our domain, in which a bicycle is just a bicycle, is suddenly entwined with the Rails domain, in which a Bicycle model is an interface to a database table. Once this happens, the two domains become glued together and can't be separated without a rewrite.

Decoupling

That's not going to happen here. We already have our game logic separated out. Now we're going to layer on the web interface. The two will live happily side by side, and they won't be tightly coupled.

We built the core logic of the game as an Application. That means we can bring it into any other Application as a dependency, and all of its functionality will be available to us. Phoenix happens to be an Application, which makes this job a snap. This one idea, this way of managing dependencies and building applications, is quietly revolutionary. It's going to make the rest of our work with Phoenix seem trivial.

• Click HERE to purchase this book now. discuss



Of course, Erlang developers have been working in this quietly revolutionary way for a couple of decades now.

All our core logic needs is a web interface, and we'll use Phoenix to build one. Phoenix has all of the MVC components you're used to for those times when HTTP's request/response cycle fits best. It's also got a real-time, persistent connection layer called channels built right in.

This is where all our hard work up to now is going to really pay off. From here, we'll be able to generate a fresh Phoenix project and bring our whole game in as a dependency. As we build out the interface, we won't be mixing in any application logic. We'll just call into the public interface of the game server that we've already built. There won't be any entanglement between the game and the interface.

Applications are what allow us to build these separate, self-contained components. They're what allow us to compose them back together into larger applications as well. We'll explore them next.

Applications

Despite the name, Applications are not what we normally think of as software applications. They are reusable units of code that are bigger than modules. In fact, they most often contain multiple modules. They're similar in scale to libraries in other ecosystems. While they can function as libraries, they can also be so much more.

Applications can act as true building blocks for our programs, a means of putting together integral pieces of business logic to build a larger whole. Working with larger building blocks like these makes us really productive.

Applications can also stand on their own as what we traditionally think of as an application. The IslandsEngine Application we developed in the first part

• Click HERE to purchase this book now. discuss

of the book is one example. It is a fully functioning game just as it is, albeit with a pretty unfriendly user interface. As complete as it is, we can still use it as a building block for something larger, as we'll soon see.

:application is a specific OTP Behaviour written in Erlang, just like :gen_server. There is a module in OTP that defines :application-specific functions as well as a list of callbacks we need to implement. Elixir provides a wrapper module around the pure Erlang one called Application. We'll be using the Elixir wrapper most often in this chapter.

The Application Behaviour lets us do three things. It lets us define and name Applications. It facilitates dependency management among Applications. We can define hierarchies of Application dependencies, and the Behaviour will make sure they work correctly. The Behaviour also facilitates cleanly starting and stopping individual applications in a running BEAM.

"Cleanly" here means two things. It makes sure to start any dependent Applications before it starts itself. It also keeps track of any processes the Application spawns during startup or while it's running, and makes sure to stop them when the Application stops.

Now that we've got an idea of the significance of Applications, let's dig a little deeper and see how they work.

Understanding Applications

The good news is that we've been working with an Application all along. At the beginning of the book, when we generated the brand-new IslandsEngine project, Mix automatically created it as an Application. We didn't need to look

deeply at the Application Behaviour then because IslandsEngine stood on its own for our purposes.

Now, though, we need to use it as a dependency, as a building block to create a larger project—the web interface we're going to build in this chapter. Understanding Application dependencies will clarify our work on this project, and any other Elixir projects we work on.

We already have examples of the Application Behaviour–related files in IslandsEngine. We'll use them to understand dependency management as well as starting and stopping individual applications inside the BEAM. We'll see firsthand the independence of Applications that lets us solve the coupling problem so prevalent in web applications.

There are three parts to the implementation of an Application, and Mix has a hand in all of them.

When we generate a project with mix new we get a file named mix.exs at the root of our project. mix.exs defines key aspects of the Application, everything from its name and version number to a list of applications it depends on to build the project.

Mix also generates a Behaviour callback module in the /lib directory that is named after our project. In the case of our game engine, it generated /lib/islands_engine/application.ex. If we supply the --sup flag to mix new, the callback module will contain the start/2 callback function necessary to start the top-level supervisor for the Application. Without --sup, the file will be there, but it will be empty.

Once we compile the project, mix will generate an application resource file, written in Erlang, that the BEAM will use to work with our Application.

Let's take a look at these files now starting with mix.exs.

Managing Dependencies

Any project's mix.exs file has two main functions—defining a project's metadata and managing its dependencies. Of the two, dependency management is by far the most common thing developers do, and it's the most important for our purposes as well.

Three functions defined in mix.exs do all the work for us. The project/0 function returns a keyword list of metadata about the application.

```
def project do
  [
    app: :islands_engine,
    version: "0.1.0",
    elixir: "~> 1.5",
    start_permanent: Mix.env == :prod,
    deps: deps()
  ]
end
```

The app name, version number, and Elixir version are pretty self-explanatory. start_permanent: starts the system in such a way that the BEAM will crash if the top-level supervisor crashes. This will be true for the production environment as well.

The deps: key holds a list of build-time dependencies this application depends on. The value here is the return value of the deps/0 function, also defined in mix.exs.

```
defp deps do
  []
end
```

IslandsEngine has no dependencies, so the return value here is an empty list. When we generate a new Phoenix project in the next section, we'll see an example with a number of dependencies.

There are actually two types of dependencies for Applications: those that matter for runtime, and those that come into play for build/compile time. Mix uses the dependencies listed in the deps/0 function to build the project. Any Application in this list can have its own dependencies. This is how we can compose a larger tree of dependencies, just as we saw with supervision trees in Chapter 5, *Process Supervision for Recovery*, on page ?.

The last function in mix.exs is application/0. It returns a keyword list of data related to starting the application. The value of the :extra_applications key is a list of application names, which are the runtime dependencies. Mix will make sure these are running before it starts :islands_engine. :mod holds a tuple for the module name of the callback module as well as a list of options that the start/2 function in that module might need.

```
def application do
  [extra_applications: [:logger],
   mod: {IslandsEngine.Application, []}]
end
```

IslandsEngine depends only on the :logger Application at runtime. This dependency is a default for all Applications Mix generates. Elixir itself supplies this, so we don't need to list it in the deps/0 function.

The reason that there are two different places to define dependencies is that it's possible to need a dependency for compilation but not need it to be running inside the BEAM, and vice versa.

If our Application doesn't have a supervision tree—for example, if we omitted the --sup flag when we generated the project—we can omit the mod key completely:

```
def application do
  [extra_applications: [:logger]]
end
```

You might be thinking that this seems a little redundant. Shouldn't Mix be able to infer the Application list from the deps list as long as we give it some clues? As of Mix 1.4, it can.

If the runtime dependencies are the same as the compile-time ones, we can omit the :extra_applications key in application/0:

```
def application do
  [mod: {IslandsEngine.Application, []}]
end
```

If there are runtime dependencies not listed in the deps/0 function—:logger, for instance—we can handle that with the :extra_applications key:

```
def application do
  [extra_applications: [:logger],
   mod: {IslandsEngine.Application, []}]
end
```

And if we have compile-time dependencies that we don't need to start when we start our application, we can mark them as runtime: false in the deps/0 function:

```
defp deps do
  [{:some_new_dep, "> 0.0.0", runtime: false}]
end
```

That brings us to the end of dependency management in mix.exs. Once we have defined the dependencies, we need to be able to start them inside the BEAM. That's where we're headed next.