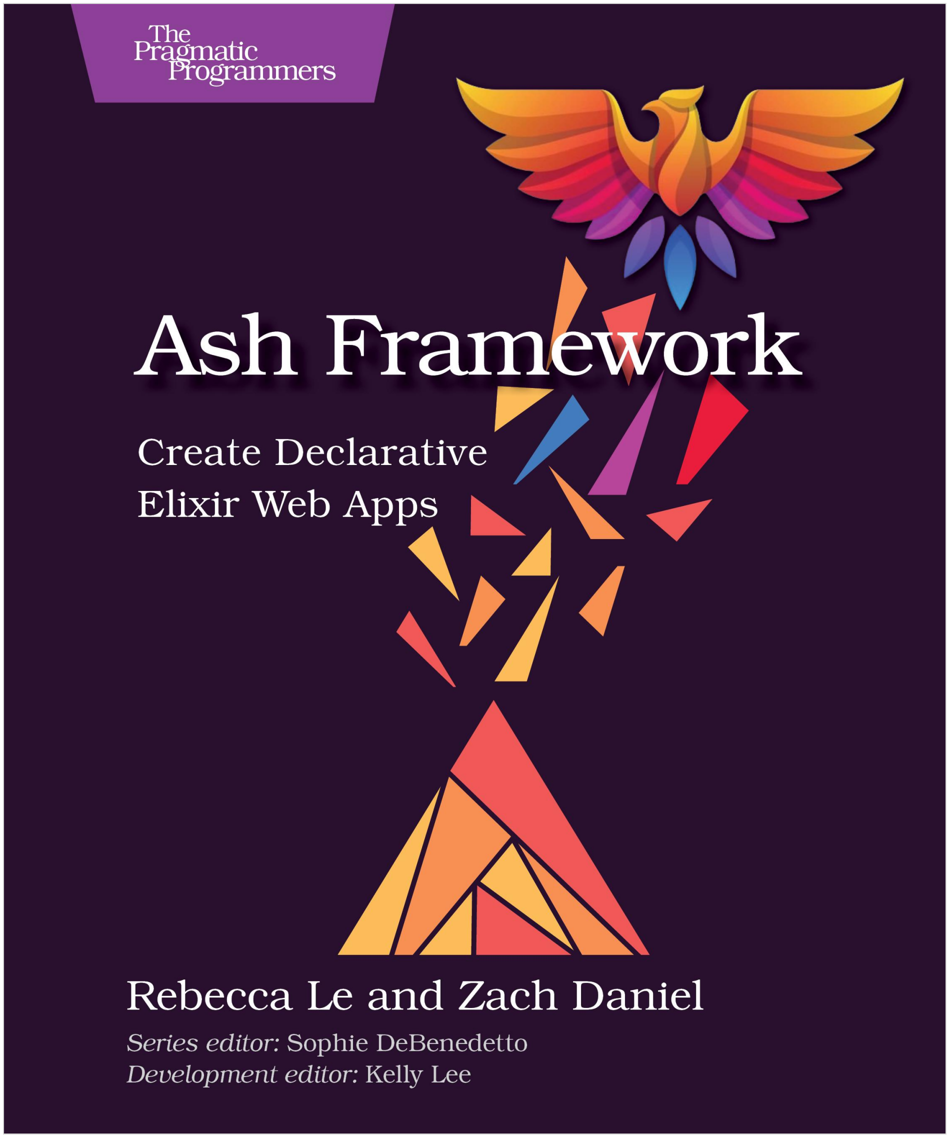
The Pragmatic Programmers
Ash Framework
Create Declarative Elixir Web Apps
Rebecca Le and Zach Daniel
Series editor: Sophie DeBenedetto
Development editor: Kelly Lee

Welcome!

As software developers, we face new and interesting challenges daily. When one of these problems appears, our instincts are to start building a mental model of the solution. The model might contain high-level concepts, ideas, or *things* that we know we want to represent, and ways they might communicate with each other to carry out the desired task.

Your next job is to find a way to map this model onto the limitations of the language and frameworks available to you. But there's a mismatch: your internal model is a fairly abstract representation of the solution, but the tooling you use demands very specific constructs, often dictated by things such as database schemas and APIs.

These problems are hard, but they're not intractible — they *can* be solved, using a framework like Ash.

Ash lets you think and code at a higher level of abstraction, and your resulting code will be cleaner, easier to manage, and you'll be less frustrated.

This book will show you the power of Ash, and how to get the most out of it in your Elixir projects.

What is Ash?

Ash is a set of tools you can use to describe and build the *domain model* of your applications — the "things" that make up what your app is supposed to do, and the business logic of how they relate and interact with each other. If you're building an e-commerce store, your domain model will have things like products, categories, suppliers, orders, customers, deliveries, and more; and you'll already have a mental model to describe how they fit together. Ash is how you can translate that mental model into code, using standardized patterns and your own terminology.

Ash is a fantastic application framework, but it is *not* a web framework. This question comes up often, so we want to be really clear up front — Ash doesn't replace Phoenix, Plug, or any other web framework when building web apps in Elixir. It does, however, slide in very nicely alongside them and work with them, and when combined they can make the ultimate toolkit for building amazing apps.

What can Ash offer an experienced Elixir/Phoenix developer? You're already familiar with a great set of tools for building web applications today, and Ash *builds* on that foundation that you know and love. It leverages the rock-solid Ecto library for its database integrations, and its resource-oriented design helps bring structure and order to the wild west of Phoenix contexts. If this sounds interesting to you, keep reading!

And if you're only just starting on your web development journey, we'd love to introduce you to our battle-tested and highly-productive stack!

Why Ash?

Ash is built on three fundamental principles. These principles are rooted in the concept of *declarative design*, and have arisen from direct encounters with the good, bad and the ugly of software in the wild. They are:

- Data > Code
- Derive > Hand-write
- What > How

To paraphrase a famous manifesto, while there is value in the items on the right, we value the items on the left more.

No principle is absolute and each has its own tradeoffs, but together they can help us build rich, maintainable and scalable applications. The "why" of Ash is rooted in the "why" of each of these core principles.

Data > Code

With Ash, we model (describe) our application components with *resource* modules, using code that compiles into pre-defined data structures. These resources describe the interfaces to, and behavior of, the various components of our application.

Ash can take the data structures created by these descriptions, and use them to do wildly useful things with little-to-no effort. Furthermore, Ash contains tools that allow you to leverage your application-as-data to build and extend your application in fully custom ways. You can introspect and use the data structures in your own code, and you can even write transformers to extend the language that Ash uses and add new behaviour to existing data.

Taking advantage of these super powers, however, requires learning the language of Ash Framework, and this is what we'll teach you in this book.

Derive > Hand-write

We emphasize *deriving* application components from our descriptions, instead of hand-writing our various application layers. When building a JSON API, for example, you might end up hand-writing controllers, serializers, OpenAPI schemas, error handling, and the list goes on. If you want to add a GraphQL API as well, you have to do it all over again with queries, mutations, and resolvers. In Ash, this is all driven from your resource definitions, using them as the single source of truth for how your application should behave. Why should you need to restate your application logic in five different ways?

There *is* value in the separation of these concerns, but that value is radically overshadowed by all of the associated costs, such as:

- The cost of bugs via functionality drift in your various components
- The cost of the conceptual overhead required to implement changes to your application and each of its interfaces
- And the cost, especially, of every piece of your application being a special snowflake with its own design, idiosyncrasies and patterns.

When you see what Ash can derive automatically, without all of the costly spaghetti code necessary with other approaches, the value of this idea becomes very clear.

What > How

This is the core principle of declarative design, and you've almost certainly leveraged this principle already in your time as a developer without even realizing it.

Two behemoths in the world of declarative design are HTML and SQL. When writing code in either language, you don't describe *how* the target is to be achieved, only *what* the target is. For HTML, a renderer is in charge of turning your HTML descriptions into pixels on a screen; and for SQL, a query planner and engine are responsible for translating your queries into procedural code that reads data from storage.

An Ash resource behaves in the exact same way, as a description of the *what*. All of the code in Ash is geared towards looking at the descriptions of what you want to happen, and making it so. This is a crucial thing to keep in mind as you go through this book — when we write resources, we are only describing their behavior. Later, when we actually call the actions we describe, or connect them to an API using an API extension for example, Ash looks at the description provided to determine what is to be done.

These principles, and the insights we derive from them, might take some time to really comprehend and come to terms with. As we go through the more concrete concepts presented in this book, revisit these principles. Ash is more than just a new tool, it's a new way of thinking about how we build applications in general.

We've seen time and time again, especially in our in-person workshops, that everyone has a moment when these concepts finally *click*. This is when Ash stops feeling like magic, and begins to look like what it really is: the principles of declarative design, taken to their natural conclusion.

Model your domain, derive the rest.

Is This Book For You?

If you've gotten this far, then yes, this book is for you!

If you have some experience with Elixir and Phoenix, have heard about this library called Ash, and are keen to find out more, then this book is *definitely* for you.

If you're a grizzled Elixir veteran wondering what all the Ash fuss is about, it's also for you!

If you've already been working with Ash, even professionally, you'll still learn new things from this book (but you can read it a bit faster).

If you haven't used Elixir before, this book is probably not for you *yet* — but it might be soon! To learn about this amazing functional programming language, we highly recommend working through *Elixir in Action [Jur15]*. To get a feel for how modern web apps are built in Elixir with Phoenix and Phoenix LiveView, *Programming Phoenix LiveView [TD24]* will get you up to speed. And then you can come back here, and keep reading!

What's In This Book

This book is divided into nine chapters, each one building on top of the previous to flesh out the domain model for a music database. We'll provide the starter Phoenix LiveView application to get up and running, and then away we'll go!

In Chapter 1, Building Our First Resource, on page ?, we'll set up the Tunez starter app, install and configure Ash, and get familiar with CRUD actions.

We'll build a full (simple) resource, complete with attributes, actions, and a database table; and integrate those actions into the web UI using forms and code interfaces.

In Chapter 2, Extending Resources with Business Logic, on page ?, we'll create a second resource, and learn about linking resources together with relationships. We'll also cover more advanced features of resources, like preparations, validations, identities, and changes.

In Chapter 3, Creating a Better Search UI, on page ?, we'll focus on features for searching, sorting, and pagination, to make our main catalog view much more dynamic. We'll also start to unlock some of the true power of Ash, by deriving new attributes with calculations and aggregates.

In Chapter 4, Generating APIs Without Writing Code, on page ?, we'll see the principle of "model your domain, derive the rest" in action, when we learn how to create full REST JSON and GraphQL APIs from our existing resource and action definitions. It's not magic, we swear!

In Chapter 5, Authentication: Who Are You?, on page ?, we'll set up authentication for Tunez, using the AshAuthentication library. We'll cover different strategies for authentication like username/password and logging in via magic link, as well as customizing the autogenerated liveviews to really make them seamless.

In Chapter 6, Authorization: What Can You Do?, on page ?, we'll introduce authorization into the app, using policies and bypasses. We'll see how we can define a policy *once* and use it throughout the entire app, from securing our APIs to showing and hiding UI buttons and more.

In *the (as yet) unwritten Chapter 7, All About Testing,* , we'll tackle the topic of testing — what should we test in an app built with Ash, and how should we do it? We'll go over some testing strategies, see what tools Ash provides to help with testing, and cover practical examples of testing Ash and LiveView apps.

In *the (as yet) unwritten Chapter 8, Fun With Nested Forms,* , we'll dig a little deeper into Ash's integration with Phoenix, by expanding our domain model and building a nested form, including drag and drop re-ordering for nested records.

And lastly, in *the (as yet) unwritten Chapter 9, PubSub and Real-Time Notifications,* , we'll use everything we've learned so far to build a user notification system. Using pubsub for broadcasting real-time updates, and AshOban for making sure batches of notifications are processed smoothly in the background, we'll create a simple yet robust system that allows for expansion as your apps grow.

Online Resources

All online resources for this book, such as errata and code samples, can be found on the Pragmatic Bookshelf product page:

>> http://pragprog.com/book/ldash <<

We also invite you to join the greater Ash community, if you'd like to learn more, or contribute to the project and ecosystem: https://ash-hq.org/community

And on that note, let's dig in! We've got a lot of exciting topics to cover, and can't wait to get started!