The Pragmatic Programmers

Ash Framework

Create Declarative Elixir Web Apps

Rebecca Le and Zach Daniel

Foreword: Josh Price
Series editor: Sophie DeBenedetto
Development editor: Kelly Lee

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/ldash/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

The Pragmatic Bookshelf

Ash Framework

Create Declarative Elixir Web Apps

Rebecca Le
Zach Daniel

The Pragmatic Bookshelf
Dallas, Texas

See our complete catalog of hands-on, practical,
and Pragmatic content for software developers:
https://pragprog.com

Sales, volume licensing, and support:
support@pragprog.com

Derivative works, AI training and testing,
international translations, and other rights:
rights@pragprog.com

Copyright © 2025 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced by any means, nor may any derivative works be made from this publication, nor may this content be used to train or test an artificial intelligence system, without the prior consent of the publisher.

When we are aware that a term used in this book is claimed as a trademark, the designation is printed with an initial capital letter 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.

ISBN-13: 979-8-88865-152-0
Encoded using recycled binary digits.
Book version: B6.0—May 30, 2025

In loving memory of Monty.

We miss your fluffy presence every single day.

- Rebecca

Contents

Foreword

Congratulations!

You're about to read a book that I believe will fundamentally change how you think about building software.

It's often mistaken for one, but Ash isn't a web framework, it's an *application* framework. The Ash tagline "Model your domain, derive the rest" describes it succinctly once you understand how it works, so let's quickly unpack what that means.

The big idea behind Ash is surprisingly simple: express your domain model using the Domain Specific Language (DSL) that Ash provides, then Ash encodes it as an introspectible data structure. Then as if by magic, an incredible vista of time saving opportunities opens up to you. You can generate anything you like!

Ash has many ways to do this already as pre-built extensions: Data Layers, Admin UIs, APIs, Authentication, the list goes on. You can also build your own extensions. What exists today is merely a taste of what's possible — the only limit is your imagination. Since anything can be derived, it can seem overwhelming at first. Don't worry, you're in good hands.

The Lisp programmers of old have often taken a similar approach: Build a DSL for the problem at hand, then build the solution using that. Ash generalises and extends this approach, making it accessible for everyone. It's a convenient syntax for expressing your domain, and a consistent way to specify your application's behaviour in a way that can be analysed, transformed, and extended.

My Ash journey was not a straight path. As Alembic's Technical Director, I looked for opportunities to try Ash on a client project for years. We eventually tried it out on an ambitious and complex client project because we thought it could help generate a GraphQL API and build an Admin UI without much code. On closer inspection Ash did way more than what it said on the tin.

The client architect and I both eventually came to the same conclusion — we couldn't contemplate building such a large application without something like Ash, and we definitely didn't want to build it from scratch.

Our version of *Greenspun's tenth rule of programming*[1] is as follows:

> *Any sufficiently large software application contains an ad hoc, informally specified, bug-ridden, slow implementation of half of Ash.*

It's funny because it's true — when applications grow beyond a certain size, developers inevitably start building frameworks to manage complexity. They create utilities for common patterns, abstractions for repeated logic, and tools for generating boilerplate. Ash provides all of this out of the box, in a well-tested, battle-hardened package.

Don't tell anyone, but Ash is our secret sauce. Since then, Ash has been our preferred stack for large scale projects, and our clients are loving the benefits. Elixir is our preferred language ecosystem because it's incredibly efficient. We'll build software using other technologies but it's never quite as simple, comfortable or efficient. We also deeply believe in open source software, because it's fundamentally a positive sum game. When improvements are made to the ecosystem, all projects can immediately benefit — a rising tide indeed lifts all boats!

For 3 years, Rebecca has worked on some of the most ambitious client projects Alembic has built. She was one of the first at turning her hand at building large Elixir applications with Ash, and has the scar tissue to prove it. Her work has informed the development of Ash into the polished product it is today. She is an exemplary technical communicator, and you will feel like you're in an extremely safe pair of hands as you work your way through this book. I certainly did!

Zach, the creator of the Ash Framework, has been building the ecosystem for over 5 years. He works tirelessly on making Ash better every day. His vision for what Ash could be has evolved through constant feedback from real-world usage. His commitment to maintaining and improving the framework is remarkable. What started as a tool for generating APIs has grown into a comprehensive framework for building robust, maintainable applications.

Together, they bring both the deep architectural understanding and the practical experience of building real-world applications. This combination means you're getting both the "why" and the "how" — the theoretical under-

1. https://philip.greenspun.com/research/

pinnings that make Ash powerful, and the practical knowledge of how to use it effectively.

By the time you finish this book, you'll have a new perspective on how to manage complexity in software applications. You'll see how making your domain model explicit and introspectible opens up new possibilities for building and maintaining software. Whether you're building a small service or a large enterprise application, the ideas in this book will help you create more maintainable, consistent, and powerful software.

I'm personally delighted to have been a small part of the journey so far, and am very excited about where we can take this in future.

Let's build something amazing together!

Josh Price
Founder and Technical Director, Alembic
Sydney, Australia, February 2025

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B6.0: May 30, 2025

- The book is content complete and headed to production.

B5.0: May 1, 2025

- Added content for chapter 9, Now Following: Your Favorite Artists, and chapter 10, PubSub and Real-Time Notifications. This includes code changes to the initial Tunez app, mainly to `TunezWeb.Artists.IndexLive`.

  To upgrade your existing Tunez apps with the new code, you can update your forked version of the Tunez repo, or patch your existing codebase with the new code using the following command:

  ```
  curl https://github.com/sevenseacat/tunez/commit/0e00b4e.patch | git apply
  ```

- Updated `relate_actor` change for `Tunez.Music.Artist` and `Tunez.Music.Album`, to allow nil values when updating records. System actions can be updates too, like background jobs!
- Replaced usage of `transform_params` when creating a `Tunez.Music.Album` record with customizing the AshPhoenix `forms` DSL, in chapter 2
- Fix bug with the `Tunez.Music.Track` form not showing an error for a duration of 0:00, in chapter 8

B4.0: March 6th, 2025

- You asked, we listened: Tunez now longer depends on having NodeJS installed. This has required quite a few little tweaks to components and templates, most of which should be able to be seamlessly copied over.

To upgrade your existing Tunez apps with the new code, you can update your forked version of the Tunez repo, or patch your existing codebase with the new code using the following command:

```
curl https://github.com/sevenseacat/tunez/commit/03d0ce26.patch | git apply
```

- Add note about how to generate indexes for foreign keys in chapter 2, as PostgreSQL doesn't do this by default

B3.0: February 13th, 2025

- Added content for chapter 8 — Fun with Nested Forms. This includes code changes to the initial Tunez app, mainly to `TunezWeb.Artists.ShowLive` and `TunezWeb.Albums.FormLive`.

  To upgrade your existing Tunez apps with the new code, you can update your forked version of the Tunez repo, or patch your existing codebase with the new code using the following command:

  ```
  curl https://github.com/sevenseacat/tunez/commit/9e54a54.patch | git apply
  ```

- Added steps on how to incrementally enable seeds for the `mix seed` alias in `mix.exs`
- Moved the sorting of albums to be done via the `Tunez.Music.Artist` `has_many :albums` relationship, instead of a global preparation on the `Tunez.Music.Album` resource
- Updated the section around needing to remove `allow_nil? false` for the `hashed_password` attribute of the `Tunez.Music.User` resource when adding magic link authentication in chapter 5 — this is now done automatically when adding the magic link strategy
- Updated the name of the `ash.patch.extend` Mix task to be `ash.extend`

B2.0: January 27th, 2025

- Added content for chapter 7 — All About Testing. This is a big one, and includes a full test suite for the Tunez app.

  To upgrade your existing Tunez apps with the new code, you can update your forked version of the Tunez repo, or patch your existing codebase with the new code using the following command:

  ```
  curl https://github.com/sevenseacat/tunez/commit/86a2b2.patch | git apply
  ```

- Removed sections about connecting generated email senders to mailers in chapter 5 — this is now done automatically by AshAuthentication when adding authentication strategies
- Modified `cover_image_url` validation for `Tunez.Music.Album` resource — this validation now has the condition `where: [changing(:cover_image_url)]`
- Modified how aggregates are loaded when calling `Tunez.Music.search_artists` — these are now done as part of the `search_artists` code interface with `default_options`, instead of in the search action itself

B1.0: January 14th, 2025

- Initial beta release.

Acknowledgments

Writing a book like this takes so much more than two authors putting words and code on the page.

We'd like to thank the amazing team at PragProg that have worked with us and supported us every step of the way, especially our intrepid editor Kelly Lee, as well as Dave Thomas, Sophie DeBenedetto, Margaret Eldridge, Susannah Davidson, and Juliet Thomas.

Many reviewers helped us out by reading and providing feedback on even the earliest versions of each chapter. Thank you to James Harton, Kathryn Prestridge, Mike Buhot, Stefan Wintermeyer, Daniel Pipkin, Nicholas Moen, Peter Wurm, Thomas Fejes, Andrew Ek and Chaz Watkins.

And finally, to all of our readers: thank you for picking up this book and giving Ash a chance. You are the reason we write.

Rebecca Le

What a wild ride this has been! I'll keep this short and sweet.

Thank you to Jeff Chan, who has always told me to *go for it*, and has made me a better developer, communicator, and leader. You've dragged me out of the mud and talked me off the metaphorical ledge more times than I can count, and I sincerely appreciate it.

Thank you to the fluffy feline members of my family, Monty, Scooter, and Ziggy; who try their best to get me to take regular breaks and give them lots of attention. Your cuddles (and sometimes claws) keep me grounded and help me breathe.

Many thanks to Zach, for bringing us this amazing framework and tirelessly supporting it every day. For fixing all my bugs, letting me rant and whinge (and curse his name) every now and then, coming up with game-changing ideas, and pushing me wayyyyy out of my comfort zone. The book is better for all of it!

But most importantly, thank you to my awesome husband, Thuc. Words can't express how much you mean to me, but I can only try. You're the Boston Rob to my Amber, the Stoinis to my Zampa, the potato to my gravy. This book is for you.

Well, it's for the boys too. But mostly for you.

Zach Daniel

First and foremost always, my wife, Meredith. Without her, nothing that I do would be possible. She bears the burden of my work just as much as I, and has supported me unreservedly. I never truly understood happiness, the deep and boundless kind, before her.

I count among my blessings a family that values kindness, excellence and good humor. To my family, Mom, Dad, Allison, Kat, Ann and Dave, who shaped me and continue to inspire me to be the best that I can be.

To my furry family, who are little goblins that, I could not possibly live without. They are the best reminders to get my head out of my laptop. Pippin, Kuma, Juno, Zeus, Rory and Khloe (yes I live in a zoo).

To Brandon, who has sacrificed too many of our online gaming nights to count on account of my work obsession. He is the most steadfast friend one could ask for.

To Geena, who took a chance on me and hired me for my first ever job in tech. A boss, turned life long friend, who's company and council I value dearly.

To James who, knowing that I will work myself to the bone if left to my own devices, sends me pictures of him and his dogs playing in the river to remind me that there is more to life than code.

To Rebecca, who is far and above the mastermind behind this book. Put simply, my job here is the tech. Her keen eye, and the process of writing this book, has refined and improved Ash in immeasurable ways. The spirit in this book, the educational value and word smithing is all to her credit. It has been my privilege to work alongside her.

To my colleagues at Alembic and to its leadership, Josh and Suzie, who have believed in Ash since it was barely a diamond in the rough. Its a pleasure to work among such great minds.

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 intractable — 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 1, 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 33, 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 59, 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 85, 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 107, 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 123, 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 Chapter 7, All About Testing, on page 157, 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 Chapter 8, Fun With Nested Forms, on page 179, 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.

In Chapter 9, Now Following: Your Favorite Artists, on page 207, we'll explore many-to-many relationships, to allow users to follow their favorite artists. We'll improve our code interface game to create some really nice functions for following and unfollowing, and use the new follower information in some surprising ways!

And lastly, in Chapter 10, PubSub and Real-Time Notifications, on page 225, we'll use everything we've learned so far to build a user notification system. Using bulk actions for efficiency and pubsub for broadcasting real-time updates, 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:

>> https://pragprog.com/titles/ldash/ash-framework/ <<

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!

CHAPTER 1

Building Our First Resource

Hello! You've arrived! Welcome!!

In this very first chapter, we'll start from scratch and work our way up. We'll set up the starter Tunez application, install Ash, and build our first resource (as the chapter title suggests!). We'll define attributes, set up actions, and connect to a database, all while seeing firsthand how Ash's declarative principles simplify the process. By the end, you'll have a working resource fully integrated with the Phoenix frontend — and the confidence to take the next step.

Getting the Ball Rolling

Throughout this book, we'll build Tunez, a music database app. Think of it as a lightweight Spotify, without actually playing music — users can browse a catalog of artists and albums, follow their favorites, and receive notifications when new albums are released. On the management side, we'll implement a role-based access system with customizable permissions, and create APIs that allow users to integrate Tunez data into their own apps.

But Tunez is more than just an app — it's your gateway to mastering Ash's essential building blocks. By building Tunez step by step, you'll gain hands-on experience with resources, relationships, authentication, authorization, APIs, and more. Each feature we build will teach you foundational skills you can apply to any project, giving you the toolkit and know-how to tackle larger, more complex applications with the same techniques. Tunez may be small, but the lessons you'll learn here will have a big impact on your development workflow.

A demo version of the final Tunez app can be seen here:

https://tunez.sevenseacat.net/

Setting up your development environment

One of the (many) great things about the Elixir ecosystem is that we get a lot of great new functionality with every new version of Elixir, but nothing gets taken away (at worst, it gets deprecated). So while it would be awesome to always use the latest and greatest versions of everything, sometimes that's not possible, and that's okay! Our apps will still work with most recent versions of Elixir, Erlang, and PostgreSQL.

To work through this book, you'll need at least:

- Elixir 1.15
- Erlang 26.0
- PostgreSQL 14.0

Any newer version will also be just fine!

To install these dependencies, we'd recommend a tool like asdf[1] or mise.[2]

We've built an initial version of the Tunez app, for you to use as a starting point. To follow along with this book, clone the app from the following repository:

https://github.com/sevenseacat/tunez

If you're using asdf, once you've cloned the app, you can run asdf install from the project folder to get all the language dependencies set up. The .tool-versions file in the app lists slightly newer versions than the dependencies listed above, but you can use any versions you prefer as long as they meet the minimum requirements.

Follow the setup instructions in the app README, including mix setup, to make sure everything is good to go. If you can run mix phx.server without errors and see a styled homepage with some sample artist data, you're ready to begin!

The code for each chapter can also be found in the Tunez repo on GitHub, in branches named after the end of the chapter, e.g. the app at the end of chapter 1 can be found at https://github.com/sevenseacat/tunez/tree/end-of-chapter-1.

1. https://asdf-vm.com/
2. https://mise.jdx.dev/

Welcome to Ash!

Before we can start using Ash in Tunez, we'll need to install it and configure it within the app. Tunez is a blank slate — it has a lot of the views and template logic, but no way of storing or reading data. This is where Ash comes in — it will be our main tool for building out the domain model layer of the app, the code responsible for reading and writing data from the database, and implementing our app's business logic.

To install Ash, we'll use the Igniter[3] toolkit, which is already installed as a development dependency in Tunez. Igniter gives library authors tools to write smarter code generators, including installers, and we'll see that here with the igniter.install Mix task.

Run mix igniter.install ash in the tunez folder, and it will patch the mix.exs file with the new package:

```
$ mix igniter.install ash
compile ✓

Update: mix.exs

     ...|
35 35   |  defp deps do
36 36   |    [
   37 + |      {:ash, "~> 3.0"},
37 38   |      {:phoenix, "~> 1.8.0"},
38 39   |      {:phoenix_ecto, "~> 4.5"},
     ...|

These dependencies should be installed before continuing. Modify mix.exs and
install? [Y/n]
```

Confirm the change, and Igniter will install and compile the latest version of the ash package. This will trigger Ash's own installation Mix task, which will add Ash-specific formatting configuration in .formatter.exs and config/config.exs. The output is a little too long to print here, but we'll get consistent code formatting and section ordering across all of the Ash-related modules we'll write over the course of the project.

Starting a new app and want to use Igniter and Ash?

Much like the phx_new package is used to generate new Phoenix projects, Igniter has a companion igniter_new package for generating projects. You can install it with:

3. https://hexdocs.pm/igniter/

Starting a new app and want to use Igniter and Ash?

```
$ mix archive.install hex igniter_new
```

This gives access to the igniter.new[4] Mix task, which is *very* powerful. It can also combine with phx.new, so you can use Igniter to scaffold Phoenix apps that come pre-installed with any package you like (and will also pre-install Igniter). For example, a Phoenix app with Ash and ErrorTracker:

```
$ mix igniter.new my_app --with phx.new --install ash,error_tracker
```

If you'd like to get up and running with new apps *even faster*, there's an interactive installer on the AshHQ homepage.[5] You can select the packages you want to install, and get a one-line command to run in your terminal — and then you'll be off and racing!

The Ash ecosystem is made up of many different packages for integrations with external libraries and services, allowing us to pick and choose only the dependencies we need. As we're building an app that will talk to a PostgreSQL database, we'll want the PostgreSQL Ash integration. Use mix igniter.install to add it to Tunez as well:

```
$ mix igniter.install ash_postgres
```

Confirm the change to our mix.exs file, and the package will be downloaded and installed. After completion, this will:

- Add and fetch the ash_postgres Hex package (in mix.exs and mix.lock)
- Add code auto-formatting for the new dependency (in .formatter.exs and config/config.exs)
- Update the database Tunez.Repo module to use Ash, instead of Ecto (in lib/tunez/repo.ex). This also includes a list of PostgreSQL extensions to be installed and enabled by default
- Updated some Mix aliases to use Ash, instead of Ecto (in mix.exs)
- Generate our first migration to set up the ash-functions pseudo-extension listed in the Tunez.Repo module (in priv/repo/migrations/<timestamp>_initialize_extensions_1.exs)
- Generate an extension config file so Ash can keep track of which PostgreSQL extensions have been installed (in priv/resource_snapshots/repo/extensions.json)

You'll also see a notice from AshPostgres. It has inferred the version of PostgreSQL you're running, and configured that in Tunez.Repo.min_pg_version/0.

4. https://hexdocs.pm/igniter_new/Mix.Tasks.Igniter.New.html
5. https://ash-hq.org/#get-started

And now we're good to go and can start building!

Resources and domains

In Ash, the central concept is the *resource*. Resources are domain model objects, the nouns that our app revolves around — they typically (but not always!) contain some kind of data and define some actions that can be taken on that data.

Related resources are grouped together into *domains*[6] — context boundaries where we can define configuration and functionality that will be shared across all connected resources. This is also where we'll define the interfaces that the rest of the app uses to communicate with the domain model, much like a Phoenix context does.

What does this mean for Tunez? Over the course of the book, we'll define several different domains for the distinct ideas within the app such as `Music` and `Accounts`; and each domain will have a collection of resources such as `Album`, `Artist`, `Track`, `User`, and `Notification`.

Each resource will define a set of *attributes* — data that maps to keys of the resource's struct. An Artist resource will read/modify records in the form of Artist structs, and each attribute of the resource will be a key in that struct. The resources will also define *relationships* — links to other resources — as well as actions, validations, pubsub configuration and more.

Do I need multiple domains in my app?

Technically you don't *need* multiple domains. For small apps, you can get away with defining a single domain and putting all of your resources in it, but we want to be really clear about keeping closely-related resources like `Album` and `Artist` away from other closely-related resources such as `User` and `Notification`.

You may also want to provide different interfaces for the same resource, in different domains. An `Order` resource, for example, would have different functionality depending on where it is being used — someone packing boxes in a warehouse needs a very different view of an order, than someone working in a customer service department.

We've just thrown a lot of words and concepts at you — some may be familiar to you from other frameworks, others may not. We'll go over each of them as they become relevant to the app, including lots of other resources that can help you out, as well.

6. https://hexdocs.pm/ash/domains.html

Generating the Artist resource

The first resource we'll create is for an Artist. It's the most important resource for anything music-related in Tunez, that other resources such as albums will link back to. The resource will store information about an artist's name and biography — important so users can know who they're looking at!

To create our Artist resource, we'll use an Igniter generator. You could create the necessary domain and resource files yourself, but the generators are pretty convenient. We'll then generate the database migration to add a database table for storage for our resource, and then we can start fleshing out actions to be taken on our resource.

The basic resource generator will create a nearly-empty Ash resource, so we can step through it and look through the parts. Run the following in your terminal:

```
$ mix ash.gen.resource Tunez.Music.Artist --extend postgres
```

This will generate a new resource module named Tunez.Music.Artist that extends PostgreSQL, a new domain module named Tunez.Music, and has automatically included the Tunez.Music.Artist module as a resource in the Tunez.Music domain.

The code for the generated resource is in lib/tunez/music/artist.ex:

01/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  use Ash.Resource, otp_app: :tunez, domain: Tunez.Music,
    data_layer: AshPostgres.DataLayer

  postgres do
    table "artists"
    repo Tunez.Repo
  end
end
```

Let's break down this generated code piece by piece, because this is our first introduction to Ash's domain-specific language (DSL).

Because we specified --extend postgres when calling the generator, the resource will be configured with PostgreSQL as its data store for reading from and writing to via AshPostgres.DataLayer. Each Artist struct will be persisted as a row in an artist-related database table.

This specific data layer is configured using the postgres code block. The minimum information we need is the repo and table name, but there is a lot of other behaviour that can be configured[7] as well.

Ash has several different data layers built in using storage such as Mnesia[8] and ETS.[9] More can be added via external packages like we added for PostgreSQL, such as SQLite[10] or CubDB.[11] Some of these external packages aren't as fully-featured as the PostgreSQL package, but they're pretty usable!

To add attributes to our resource, add another block in the resource named attributes. Because we're using PostgreSQL, each attribute we define will be a column in the underlying database table. Ash provides macros we can call to define different types of attributes,[12] so let's add some attributes to our resource.

A primary key will be critical to identify our artists, so we can call uuid_primary_key to create a autogenerated UUID primary key. Some timestamp fields would be useful, so we know when records are inserted and updated, and we can use create_timestamp and update_timestamp for those. Specifically for artists, we also know we want to store their *name* and a short *biography*, and they'll both be string values. They can be added to the attributes block using the attribute macro.

01/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  attributes do
    uuid_primary_key :id

    attribute :name, :string do
      allow_nil? false
    end

    attribute :biography, :string

    create_timestamp :inserted_at
    update_timestamp :updated_at
  end
end
```

7. https://hexdocs.pm/ash_postgres/dsl-ashpostgres-datalayer.html
8. https://hexdocs.pm/ash/dsl-ash-datalayer-mnesia.html
9. https://hexdocs.pm/ash/dsl-ash-datalayer-ets.html
10. https://hexdocs.pm/ash_sqlite/
11. https://hexdocs.pm/ash_cubdb/
12. https://hexdocs.pm/ash/dsl-ash-resource.html#attributes

And that's all the code we need to write to add attributes to our resource!

There's a rich set of configuration options for attributes. You can read more about them in the attribute DSL documentation.[13] We've used one here, `allow_nil?`, but there are many more available.

You can also pass extra options like `--uuid-primary-key id` or `--attribute name:string` to the `ash.gen.resource` generator[14] to generate attribute-related code (and more!) if you prefer.

Right now our resource is only a module. We've configured a database table for it, but that database table doesn't yet exist. To change that, we can use another generator, `ash.codegen`.[15] This one we'll get pretty familiar with over the course of the book.

Auto-generating database migrations

If you've used Ecto for working with databases before, you'll be familiar with the pattern of creating or updating a schema module, then generating a blank migration and populating it with commands to mirror that schema. It can be a little bit repetitive, and has the possibility of your schema and your database getting out of sync. If someone updates the database structure but *doesn't* update the schema module, or vice versa, you can get some tricky and hard-to-debug issues.

Ash side-steps these kinds of issues by generating complete migrations for you based on your resource definitions. This is our first example of Ash's philosophy of "model your domain, derive the rest". Your resources are the source of truth for what your app should be and how it should behave, and everything else is derived from that.

What does this mean in practice? Every time you run the `ash.codegen` mix task, Ash (via AshPostgres) will:

- Create *snapshots* of your current resources
- Compare them with the previous snapshots (if they exist)
- And finally, generate deltas of the changes to go into the new migration.

This is data-layer agnostic: any data layer can provide its own implementation for what to do when `ash.codegen` is run. Because we're using AshPostgres, which is backed by Ecto, we get Ecto migrations.

13. https://hexdocs.pm/ash/dsl-ash-resource.html#attributes-attribute
14. https://hexdocs.pm/ash/Mix.Tasks.Ash.Gen.Resource.html
15. https://hexdocs.pm/ash/Mix.Tasks.Ash.Codegen.html

Now we have an Artist resource with some attributes, so we can generate a migration for it using the mix task:

```
$ mix ash.codegen create_artists
```

The create_artists argument given here will become the name of the generated migration module, eg. Tunez.Repo.Migrations.CreateArtists. This can be anything, but it's a good idea to describe what the migration will actually do.

Running the ash.codegen task will create a few files:

- A snapshot file for our Artist resource, in priv/resource_snapshots/repo/artists/[timestamp].json. This is a JSON representation of our resource as it exists right now.
- A migration for our Artist resource, in priv/repo/migrations/[timestamp]_create_artists.ex. This contains the schema differences that Ash has detected between our current snapshot which was just created, and the previous snapshot (which in this case, is empty).

This migration contains the Ecto commands to set up the database table for our Artist resource, with the fields we added for a primary key, timestamps, name and biography:

01/priv/repo/migrations/[timestamp]_create_artists.exs

```
def up do
  create table(:artists, primary_key: false) do
    add :id, :uuid, null: false, default: fragment("gen_random_uuid()"),
      primary_key: true
    add :name, :text, null: false
    add :biography, :text

    add :inserted_at, :utc_datetime_usec,
      null: false,
      default: fragment("(now() AT TIME ZONE 'utc')")

    add :updated_at, :utc_datetime_usec,
      null: false,
      default: fragment("(now() AT TIME ZONE 'utc')")
  end
end
```

This looks a lot like what you would write if you were setting up a database table for a pure Ecto schema — but we didn't have to write it. We don't have to worry about keeping the database structure in sync manually. We can run mix ash.codegen every time we change anything database-related, and Ash will figure out what needs to be changed and create the migration for us.

This is the first time we've touched the database, but the database will already have been created when running mix setup earlier. To run the migration we generated, use Ash's ash.migrate Mix task:

```
$ mix ash.migrate
Getting extensions in current project...
Running migration for AshPostgres.DataLayer...

[timestamp] [info] == Running [timestamp] Tunez.Repo.Migrations
  .InitializeExtensions1.up/0 forward
«truncated SQL output»
[info] == Migrated [timestamp] in 0.0s

[timestamp] [info] == Running [timestamp] Tunez.Repo.Migrations
  .CreateArtists.up/0 forward
[timestamp] [info] create table artists
[timestamp] [info] == Migrated [timestamp] in 0.0s
```

Now we have a database table, ready to store Artist data!

To roll back a migration, Ash also provides an ash.rollback Mix task, as well as ash.setup, ash.reset, and so on. These are more powerful than their Ecto equivalents — any Ash extension can set up their own functionality for each task. For example, AshPostgres provides an interactive UI to select how many migrations to roll back when running ash.rollback.

Note that if you roll back and then delete a migration to re-generate it, you'll also need to delete the snapshots that were created with the migration.

How do we actually *use* the resource to read or write data into our database, though? We'll need to define some *actions* on our resource.

Oh, CRUD! — Defining Basic Actions

An *action* describes an operation that can be performed for a given resource; it is the *verb* to a resource's *noun*. Actions can be loosely broken down into four types:

- Creating new persisted records (rows in the database table);
- Reading one or more existing records;
- Updating an existing record; and
- Destroying (deleting) an existing record.

These four types of actions are very common in web applications, and are often shortened to the acronym CRUD.

Ash also supports *generic actions* for any action that doesn't fit into any of those four categories. We won't be covering those in this book, but you can read the online documentation[16] about them.

With a bit of creativity, we can use these four basic action types to describe almost any kind of action we might want to perform in an app.

Registering for an account? That's a type of `create` action on a `User` resource.

Searching for products to purchase? That sounds like a `read` action on a `Product` resource.

Publishing a blog post? It could be a `create` action if the user is writing the post from scratch, or an `update` action if they're publishing an existing saved draft.

In Tunez, we'll have functionality for users to list artists and view details of a specific artist (both `read` actions), `create` and `update` artist records (via forms), and also `destroy` artist records; so we'll want to use all four types of actions. This is a great time to learn how to define and run actions using Ash, with some practical examples.

In our `Artist` resource, we can add an empty block for `actions`, and then start filling it out with what we want to be able to do:

01/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  actions do
  end
end
```

Let's start with creating records with a `create` action, so we have some data to use when testing out other types of actions.

Defining a create action

Actions are defined by adding them to the `actions` block in a resource. At their most basic, they require a type (one of the four mentioned earlier — `create`, `read`, `update` and `destroy`), and a name. The name can be any atom you like, but should describe what the action is actually supposed to do. It's common to give the action the same name as the action type, until you know you need something different.

16. https://hexdocs.pm/ash/generic-actions.html

01/lib/tunez/music/artist.ex
```
actions do
  create :create do
  end
end
```

To create an Artist record, we need to provide the data to be stored — in this case, the name and biography attributes, in a map. (The other attributes, such as timestamps, will be automatically managed by Ash.) We call these the attributes that the action *accepts*, and can list them in the action with the accept macro.

01/lib/tunez/music/artist.ex
```
actions do
  create :create do
    accept [:name, :biography]
  end
end
```

And that's actually all we need to do to create the most basic create action. Ash knows that the core of what a create action should do is create a data layer record from provided data, so that's exactly what it will do when we run it.

Running actions

There are two basic ways we can run actions: the generic query/changeset method, and the more direct code interface method. We can test them both out in an iex session:

```
$ iex -S mix
```

Creating records via a changeset

If you've used Ecto before, this pattern may be familiar to you:

- Create a *changeset* (a set of data changes to apply to the resource)
- Pass that changeset to Ash for processing

In code, this might look like the following:

```
iex(1)> Tunez.Music.Artist
        |> Ash.Changeset.for_create(:create, %{
          name: "Valkyrie's Fury",
          biography: "A power metal band hailing from Tallinn, Estonia"
        })
        |> Ash.create()
{:ok,
 #Tunez.Music.Artist<
   __meta__: #Ecto.Schema.Metadata<:loaded, "artists">,
```

```
   id: [uuid],
   name: "Valkyrie's Fury",
   biography: "A power metal band hailing from Tallinn, Estonia",
   ...
 >}
```

We specify the action that the changeset should be created for, with the data that we want to save. When we pipe that changeset into Ash, it will handle running all of the validations and creating the record in the database, and then return the record as part of an :ok tuple. You can verify this in your database client of choice, for example, using psql tunez_dev in your terminal to connect using the inbuilt command-line client:

```
tunez_dev=# select * from artists;
-[ RECORD 1 ]----------------------------------------
id          | [uuid]
name        | Valkyrie's Fury
biography   | A power metal band hailing from Tallinn, Estonia
inserted_at | [now]
updated_at  | [now]
```

What happens if we submit invalid data, such as an Artist without a name?

```
iex(2)> Tunez.Music.Artist
        |> Ash.Changeset.for_create(:create, %{name: ""})
        |> Ash.create()
{:error,
 %Ash.Error.Invalid{
   bread_crumbs: ["Error returned from: Tunez.Music.Artist.create"],
   changeset: #Ash.Changeset<...>,
   errors: [
     %Ash.Error.Changes.Required{
       field: :name,
       type: :attribute,
       resource: Tunez.Music.Artist,
       ...
```

The record *isn't* inserted into the database, and we get an error record back telling us what the issue is: the name is required. This error comes from the allow_nil? false that we set for the name attribute. Later on in this chapter, we'll see how these returned errors are used when we integrate the actions into our web interface.

Like a lot of other Elixir libraries, most Ash functions return data in :ok and :error tuples. This is handy because it lets you easily pattern match on the result, to handle the different scenarios. To raise an error instead of returning an error tuple, you can use the bang version of a function ending in an exclamation mark, ie. Ash.create! instead of Ash.create.

Creating records via a code interface

If you're familiar with Ruby on Rails or ActiveRecord, this pattern may be more familiar to you. It allows us to skip the step of manually creating a changeset, and lets us call the action directly as a function.

Code interfaces can be defined on either a domain module or on a resource directly. We'd generally recommend defining them on domains, similar to Phoenix contexts, because it lets the domain act as a solid boundary with the rest of your application. With all your resources listed in your domain, it also gives a great overview of all your functionality in one place.

To enable this, use Ash's define macro when including the Artist resource in our Tunez.Music domain:

01/lib/tunez/music.ex

```
resources do
  resource Tunez.Music.Artist do
    define :create_artist, action: :create
  end
end
```

This will connect our domain function create_artist, to the create action of the resource. Once you've done this, if you recompile within iex the new function will now be available, complete with auto-generated documentation:

```
iex(2)> h Tunez.Music.create_artist

              def create_artist(params \\ nil, opts \\ nil)

Calls the create action on Tunez.Music.Artist.

# Inputs

  • name
  • biography
```

You can call it like any other function, with the data to be inserted into the database:

```
iex(7)> Tunez.Music.create_artist(%{
          name: "Valkyrie's Fury",
          biography: "A power metal band hailing from Tallinn, Estonia"
        })
{:ok, #Tunez.Music.Artist<...>}
```

When would I use changesets instead of code interfaces, or vice versa?

Under the hood, the code interface is creating a changeset and passing it to the domain, but that repetitive logic is hidden away. So there's no real functional benefit, but the code interface is easier to use and more readable.

> Where the changeset method shines is around forms on the page — we'll see shortly how AshPhoenix provides a thin layer over the top of changesets to allow all of Phoenix's existing form helpers to work seamlessly with Ash changesets instead of Ecto changesets.

We've provided some sample content for you to play around with — there's a mix seed alias defined in the aliases/0 function in the Tunez app's mix.exs file. It has three lines for three different seed files, all commented out. Uncommend the first line:

01/mix.exs

```
defp aliases do
  [
    setup: ["deps.get", "ash.setup", "assets.setup", "assets.build", ...],
    "ecto.setup": ["ecto.create", "ecto.migrate"],
    seed: [
      "run priv/repo/seeds/01-artists.exs",
      # "run priv/repo/seeds/02-albums.exs",
      # "run priv/repo/seeds/08-tracks.exs"
    ],
    # ...
```

Running mix seed will now import a list of sample (fake) artist data into your database. There are other seed files listed in the function as well but we'll mention those when we get to them! (The chapter numbers in the filenames are a bit of a giveaway…)

```
$ mix seed
```

Now that we have some data in our database, let's look at other types of actions.

Defining a read action

In the same way we defined the create action on the Artist resource, we can define a read action by adding it to the actions block. We'll add just one extra option: we'll define it as a *primary action*.

01/lib/tunez/music/artist.ex

```
actions do
  # ...
  read :read do
    primary? true
  end
end
```

A resource can have one of each of the four action types (create, read, update and destroy) marked as the primary action of that type. These are used by Ash behind the scenes when actions aren't or *can't* be specified. We'll cover these in a little bit more detail later.

To be able to call the read action as a function, add it as a code interface in the domain just like we did with create.

01/lib/tunez/music.ex

```
resource Tunez.Music.Artist do
  # ...
  define :read_artists, action: :read
end
```

What does a read action *do*? As the name suggests, it will read data from our data layer based on any parameters we provide. We haven't defined any parameters in our action, so when we call the action, we should expect it to return all the records in the database.

```
iex(1)> Tunez.Music.read_artists()
{:ok, [#Tunez.Music.Artist<...>, #Tunez.Music.Artist<...>, ...]}
```

While actions that modify data use changesets under the hood, read actions use *queries*. If we *do* want to provide some parameters to the action, such as filtering or sorting, we need to modify the query when we run the action. This can be done either as part of the action definition (we'll learn about that in Designing a search action, on page 59!), or inline when we call the action.

Manually reading records via a query

While this isn't something you'll do a lot of when building applications, it's still a good way of seeing how Ash builds up queries piece by piece.

There are a few steps to the process:

- Creating a basic query from the action we want to run
- Piping the query through other functions to add any extra parameters we want, and then
- Passing the final query to Ash for processing.

In iex you can test this out step by step, starting from the basic resource, and creating the query:

```
iex(2)> Tunez.Music.Artist
Tunez.Music.Artist
iex(3)> |> Ash.Query.for_read(:read)
#Ash.Query<resource: Tunez.Music.Artist>
```

Then you can pipe that query into Ash's query functions like sort and limit. The query keeps getting the extra conditions added to it, but it isn't yet being run in the database.

```
iex(4)> |> Ash.Query.sort(name: :asc)
#Ash.Query<resource: Tunez.Music.Artist, sort: [name: :asc]>
iex(5)> |> Ash.Query.limit(1)
#Ash.Query<resource: Tunez.Music.Artist, sort: [name: :asc], limit: 1>
```

Then when it's time to go, Ash can call it and return you the data you requested, with all conditions applied:

```
iex(6)> |> Ash.read()
SELECT a0."id", a0."name", a0."biography", a0."inserted_at", a0."updated_at"
FROM "artists" AS a0 ORDER BY a0."name" LIMIT $1 [1]
{:ok, [#Tunez.Music.Artist<...>]}
```

For a full list of the query functions Ash provides, check out the documentation.[17] Note that to use any of the functions that use special syntax like filter, you'll need to require Ash.Query in your iex session first.

Reading a single record by primary key

One very common requirement is to be able to read a single record by its primary key. We're building a music app, so we'll be building a page where we can view an artist's profile, and we'll want an easy way to fetch that single Artist record for display.

We have a basic read action already, and we *could* write another read action that applies a filter to only fetch the data by an ID we provide, but Ash provides a simpler way.

A neat feature of code interfaces is that they can automatically apply a filter for any attribute of a resource, that we expect to return at most *one* result. Looking up records by primary key is a perfect use case for this, because they're guaranteed to be unique!

To use this feature, add another code interface for the same read action, but adding the get_by option[18] for the primary key, an attribute named :id:

01/lib/tunez/music.ex

```
resource Tunez.Music.Artist do
  # ...
  define :get_artist_by_id, action: :read, get_by: :id
end
```

17. https://hexdocs.pm/ash/Ash.Query.html
18. https://hexdocs.pm/ash/dsl-ash-domain.html#resources-resource-define-get_by

Adding this code interface defines a new function on our domain:

```
iex(4)> h Tunez.Music.get_artist_by_id

              def get_artist_by_id(id, params \\ nil, opts \\ nil)

Calls the read action on Tunez.Music.Artist.
```

Copy the ID from any of the records you loaded when testing the read action, and you'll see that this new function does exactly what we hoped — return the single record that has that ID.

```
iex(3)> Tunez.Music.get_artist_by_id("an-artist-id")
SELECT a0."id", a0."name", a0."biography", a0."inserted_at", a0."updated_at"
FROM "artists" AS a0 WHERE (a0."id"::uuid::uuid = $1::uuid::uuid)
["an-artist-id"]
{:ok, #Tunez.Music.Artist<id: "an-artist-id", ...>}
```

Perfect! We'll be using that very soon.

Defining an update action

A basic update action is conceptually very similar to a create action. The main difference is that instead of building a new record with some provided data and saving it into the database, we provide an existing record to be updated with the data and saved.

Let's add the basic action and code interface definition:

01/lib/tunez/music/artist.ex

```
actions do
  # ...
  update :update do
    accept [:name, :biography]
  end
end
```

01/lib/tunez/music.ex

```
resource Tunez.Music.Artist do
  # ...
  define :update_artist, action: :update
end
```

How would we call this new action? First we need a record to be updated. You can use the read action you defined earlier to find one, or if you've been testing the get_artist_by_id function we just wrote, you might have one right there. Note the use of the bang version of the function here (get_artist_by_id!), to get back a record instead of an :ok tuple.

```
iex(3)> artist = Tunez.Music.get_artist_by_id!("an-artist-id")
#Tunez.Music.Artist<id: "an-artist-id", ...>
```

Now we can either use the code interface we added or create a changeset and apply it, like we did for create.

```
iex(4)> # Via the code interface
iex(5)> Tunez.Music.update_artist(artist, %{name: "Hello"})
UPDATE "artists" AS a0 SET "name" = $1, "updated_at" = (CASE WHEN
$2::text::text != a0."name"::text::text THEN $3::timestamp ELSE
a0."updated_at"::timestamp END)::timestamp WHERE (a0."id"::uuid::uuid =
$4::uuid::uuid) RETURNING a0."id", a0."name", a0."biography", a0."inserted_at",
a0."updated_at" ["Hello", "Hello", [now], "an-artist-id"]
{:ok, #Tunez.Music.Artist<id: "an-artist-id", name: "Hello", ...>}

iex(6)> # Or via a changeset
iex(7)> artist
        |> Ash.Changeset.for_update(:update, %{name: "World"})
        |> Ash.update()
«an almost-identical SQL statement»
{:ok, #Tunez.Music.Artist<id: "an-artist-id", name: "World", ...>}
```

Like create actions, we get either an :ok tuple with the updated record, or an :error tuple with an error record back.

Defining a destroy action

The last type of core action that Ash provides are destroy actions — when we want to get rid of data, delete it from our database. Like update actions, they work on existing data records, so we need to provide a record when we call a destroy action… but that's the only thing we need to provide. Ash can do the rest!

You might be able to guess by now, how to implement a destroy action in our resource:

01/lib/tunez/music/artist.ex

```
actions do
  # ...
  destroy :destroy do
  end
end
```

01/lib/tunez/music.ex

```
resource Tunez.Music.Artist do
  # ...
  define :destroy_artist, action: :destroy
end
```

This will allow us to call the action, either by creating a changeset and submitting it or calling the action directly.

```
iex(3)> artist = Tunez.Music.get_artist_by_id!("the-artist-id")
#Tunez.Music.Artist<id: "an-artist-id", ...>
```

```
iex(4)> # Via the code interface
iex(5)> Tunez.Music.destroy_artist(artist)
DELETE FROM "artists" AS a0 WHERE (a0."id" = $1) ["the-artist-id"]
:ok

iex(6)> # Or via a changeset
iex(7)> artist
        |> Ash.Changeset.for_destroy(:destroy)
        |> Ash.destroy()
DELETE FROM "artists" AS a0 WHERE (a0."id" = $1) ["the-artist-id"]
:ok
```

And with that, we have a solid explanation of the four main types of actions we can define in our resources.

Let's take a bit to let this really sink in. By creating a resource and adding a few lines of code to describe its attributes and what actions we can take on it, we now have:

- A database table to store records in
- Secure functions we can call to read and write data to the database (no storing any attributes that aren't explicitly allowed!)
- Database-level validations to ensure that data is present
- Automatic type-casting of attributes before they get stored

We didn't have to write any functions that query the database, or update our database schema when we added new attributes to the resource, or manually cast attributes. A lot of the boilerplate we would typically need to write has been taken care of for us because Ash handles translating *what* our resource should do into *how* it should be done. This is a pattern we'll see a lot!

Default actions

Now that you've learned about the different types of actions and how to define them in your resources, we'll let you in on a little secret…

You don't actually *have* to define empty actions like this for CRUD actions.

You know how we said that Ash knows that the main purpose of a `create` action is to take the data and save it to the data layer? This is what we call the *default* implementation for a `create` action. Ash provides default implementations for all four action types, and if you want to use these implementations without any customization, you can use the `defaults` macro in your `actions` block:

01/lib/tunez/music/artist.ex

```
actions do
  defaults [:create, :read, :update, :destroy]
end
```

We still need the code interface definitions if we want to be able to call the actions as functions, but we can cut out the empty actions to save time and space. This also marks all four actions as primary? true, as a handy side-effect.

But what about the accept definitions that we added to the create and update actions, the list of attributes to save? We can define default values for that list, with the default_accept[19] macro. This default list will then apply to all create and update actions unless specified otherwise (as part of the action definition).

So the actions for our whole resource as it stands right now could be written in a few short lines of code:

01/lib/tunez/music/artist.ex

```
actions do
  defaults [:create, :read, :update, :destroy]
  default_accept [:name, :biography]
end
```

That is a *lot* of functionality packed into those four lines!

Which version of the code you write is up to you — for actions other than read, we would generally err on the side of explicitly defining the actions with the attributes they accept, as you'll need to convert them whenever you need to add business logic to your actions anyway. We don't tend to customize the basic read action, and using a default action for read actually adds some extra functionality as well (mostly around pagination) so that usually gets placed in the defaults list.

For quick prototyping though, the shorthand for all four actions can't be beat. Whichever way you go, it's critical to know what your code is doing for you, under the hood — generating a full CRUD interface to your resource, allowing you to manage your data.

Integrating Actions into LiveViews

We've talked a lot about Tunez the web app, but we haven't even *looked* at the app in an actual browser yet. Now that we have a fully-functioning resource, let's integrate it into the web interface so we can see the actions in, well, action!

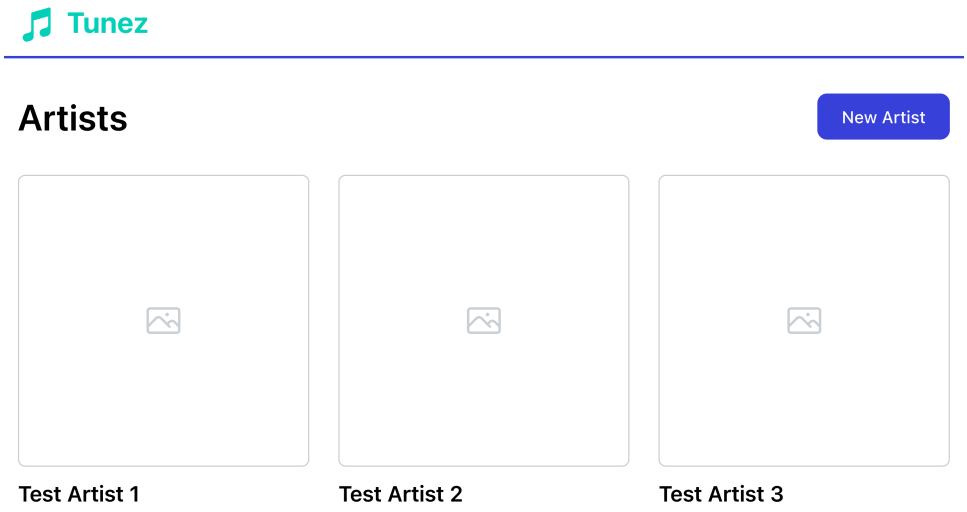
Listing artists

In the app folder, start the Phoenix webserver with the following command in your terminal:

19. https://hexdocs.pm/ash/dsl-ash-resource.html#actions-default_accept

```
$ mix phx.server
[info] Running TunezWeb.Endpoint with Bandit 1.6.11 at 127.0.0.1:4000 (http)
[info] Access TunezWeb.Endpoint at http://localhost:4000
[watch] build finished, watching for changes...
≈ tailwindcss v4.1.4

Done in [time]ms.
```

Once you see that the build is ready to go, open a web browser at http://localhost:4000 and you can see what we've got to work with.



The app homepage is the artist catalog, listing all of the Artists in the app. In the code, this catalog is rendered by the TunezWeb.Artists.IndexLive module, in lib/tunez_web/live/artists/index_live.ex.

We're not going to go into a lot of detail about Phoenix and Phoenix LiveView, apart from where we need to to ensure that our app is secure. If you need a refresher course (or want to learn them for the first time!), we can recommend reading through *Programming Phoenix 1.4 [TV19]* and *Programming Phoenix LiveView [TD24]*.

If you're more of a video person, you can't go past Pragmatic Studio's Phoenix LiveView course.[20]

In the IndexLive module, we have some hardcoded maps of artist data defined in the handle_params/3 function:

01/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(_params, _url, socket) do
```

20. https://pragmaticstudio.com/courses/phoenix-liveview

```
  artists = [
    %{id: "test-artist-1", name: "Test Artist 1"},
    %{id: "test-artist-2", name: "Test Artist 2"},
    %{id: "test-artist-3", name: "Test Artist 3"},
  ]

  socket =
    socket
    |> assign(:artists, artists)

  {:noreply, socket}
end
```

These are what are iterated over in the render/1 function, using a function component to render an artist "card" for each artist — the image placeholders and names we can see in the browser.

Earlier in Defining a read action, on page 15, we defined a code interface function on the Tunez.Music domain for reading records from the database. It returns Artist structs that have id and name keys, just like the hardcoded data does. So to load real data from the database, replace the hardcoded data with a call to the read action.

01/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(_params, _url, socket) do
  artists = Tunez.Music.read_artists!()
  # ...
```

And that's it! The page should reload in your browser when you save the changes to the liveview, and you should be seeing the names of the seed artists and the test artists you created, rendered on the page.

Each of the placeholder images and names links to a separate profile page, where you can view details of a specific artist, and we'll connect that up next.

Viewing an artist profile

Clicking on the name of one of the artists will bring you to their profile page. This liveview is defined in TunezWeb.Artists.ShowLive, which you can verify by checking the logs of the web server in your terminal:

```
[debug] MOUNT TunezWeb.Artists.ShowLive
  Parameters: %{"id" => "[the artist UUID]"}
```

Inside that module, in lib/tunez_web/live/artists/show_live.ex, you'll again see some hardcoded artist data defined in the handle_params/3 function and added to the socket.

01/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(_params, _url, socket) do
```

```
  artist = %{
    id: "test-artist-1",
    name: "Artist Name",
    biography: «some sample biography content»
  }
  # ...
```

Earlier in Reading a single record by primary key, on page 17, we defined a get_artist_by_id code interface function on the Tunez.Music domain, that reads a single Artist record from the database by its id attribute. The URL for the profile page contains the ID of the Artist to show on the page, and the terminal logs showed that the ID is available as part of the params. So we can replace the hardcoded data with a call to get_artist_by_id, after first using pattern matching to get the ID from the params.

01/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(%{"id" => artist_id}, _url, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)
  # ...
```

After saving the changes to the liveview, the page should refresh and you should see the correct data for the artist you're viewing the profile of.

Creating artists with AshPhoenix.Form

To create and edit Artist data, we'll have to learn how to handle forms and form data with Ash.

If we were building our app directly on Phoenix contexts using Ecto, we would have a schema module that would define the attributes for an Artist. The schema module would also define a changeset function to parse and validate data from forms before the context module would attempt to insert or update it in the database. If the data validation fails, the liveview (or a template) can take the resulting changeset with errors on it and use it to show the user what they need to fix.

In code, the changeset function might look something like this:

```
defmodule Tunez.Music.Artist do
  def changeset(artist, attrs) do
    artist
    |> cast(attrs, [:name, :biography])
    |> validate_required([:name])
  end
```

And the context module that uses it might look like this:

```
defmodule Tunez.Music do
  def create_artist(attrs \\ %{}) do
```

```
    %Artist{}
    |> Artist.changeset(attrs)
    |> Repo.insert()
  end
```

We can use a similar pattern with Ash, but we need a slightly different abstraction.

We have our Artist resource defined with attributes, similar to an Ecto schema. It has actions to create and update data, replacing the context part as well. What we're missing is the integration with our UI, a way to take the errors returned if the create action fails and show them to the user — and this is where AshPhoenix comes in.

Hello, AshPhoenix

As the name suggests, AshPhoenix is a core Ash library to make it much nicer to work with Ash in the context of a Phoenix application. We'll use it a few times over the course of building Tunez, but its main purpose is *form integration*.

Like AshPostgres and Ash itself, we can use mix igniter.install to install AshPhoenix in a terminal:

```
$ mix igniter.install ash_phoenix
```

Confirm the addition of AshPhoenix to your mix.exs file, and the package will be installed and we can start using it straight away.

A form for an action

Our Artist resource has a create action that accepts data for name and biography attributes. Our web interface will reflect this exactly — we'll have a form with a text field to enter a name, and a textarea to enter a biography.

We can tell AshPhoenix that what we want is a form to match the inputs for our create action, or more simply, a form *for* our create action. AshPhoenix will return an AshPhoenix.Form struct, and provides a set of intuitively-named functions for interacting with it. We can *validate* our form, we can *submit* our form, we can *add* and *remove* forms (for nested form data!) and more.

In an iex session, we can get familiar with using AshPhoenix.Form:

```
iex(1)> form = AshPhoenix.Form.for_create(Tunez.Music.Artist, :create)
#AshPhoenix.Form<
  resource: Tunez.Music.Artist,
  action: :create,
  type: :create,
  params: %{},
```

```
  source: #Ash.Changeset<
    domain: Tunez.Music,
    action_type: :create,
    action: :create,
    attributes: %{},
    ...
```

An AshPhoenix.Form wraps an Ash.Changeset, which behaves similarly to an Ecto.Changeset. This allows the AshPhoenix form to be a drop-in replacement for an Ecto.Changeset, when calling the function components that Phoenix generates for dealing with forms. Let's keep testing.

```
iex(2)> AshPhoenix.Form.validate(form, %{name: "Best Band Ever"})
#AshPhoenix.Form<
  resource: Tunez.Music.Artist,
  action: :create,
  type: :create,
  params: %{name: "Best Band Ever"},
  source: #Ash.Changeset<
    domain: Tunez.Music,
    action_type: :create,
    action: :create,
    attributes: %{name: "Best Band Ever"},
    relationships: %{},
    errors: [],
    data: #Tunez.Music.Artist<...>
➤   valid?: true
  >,
  ...
```

If we call AshPhoenix.Form.validate with valid data for an Artist, the changeset in the form is now valid. In a liveview, this is what we would call in a phx-change event handler, to make sure our form in-memory stays up-to-date with the latest data. Similarly, we can call AshPhoenix.Form.submit on the form in a phx-submit event handler.

```
iex(5)> AshPhoenix.Form.submit(form, params: %{name: "Best Band Ever"})
INSERT INTO "artists" ("id","name","inserted_at","updated_at") VALUES
($1,$2,$3,$4) RETURNING "updated_at","inserted_at","biography","name","id"
[[uuid], "Best Band Ever", [timestamp], [timestamp]]
{:ok,
 #Tunez.Music.Artist<
   __meta__: #Ecto.Schema.Metadata<:loaded, "artists">,
   id: [uuid],
   name: "Best Band Ever",
   ...
```

And it works! We get back a form-ready version of the return value of the action. If we had called submit with invalid data, we would get back an {:error, %AshPhoenix.Form{}} tuple back instead.

Using the AshPhoenix domain extension

We've defined code interface functions like Tunez.Music.read_artists for all of the actions in the Artist resource and used those code interfaces in our liveviews. It might feel a bit odd to now revert back to using action names directly when generating forms. And if the names of the code interface function and the action are really different, it could get confusing!

AshPhoenix provides a solution for this, with a *domain extension*. If we add the AshPhoenix extension to the Tunez.Music domain module, it will define some new functions on the domain around form generation.

In the Tunez.Music module in lib/tunez/music.ex, add a new extensions option to the use Ash.Domain line:

01/lib/tunez/music.ex

```
defmodule Tunez.Music do
  use Ash.Domain, otp_app: :tunez, extensions: [AshPhoenix]

  # ...
```

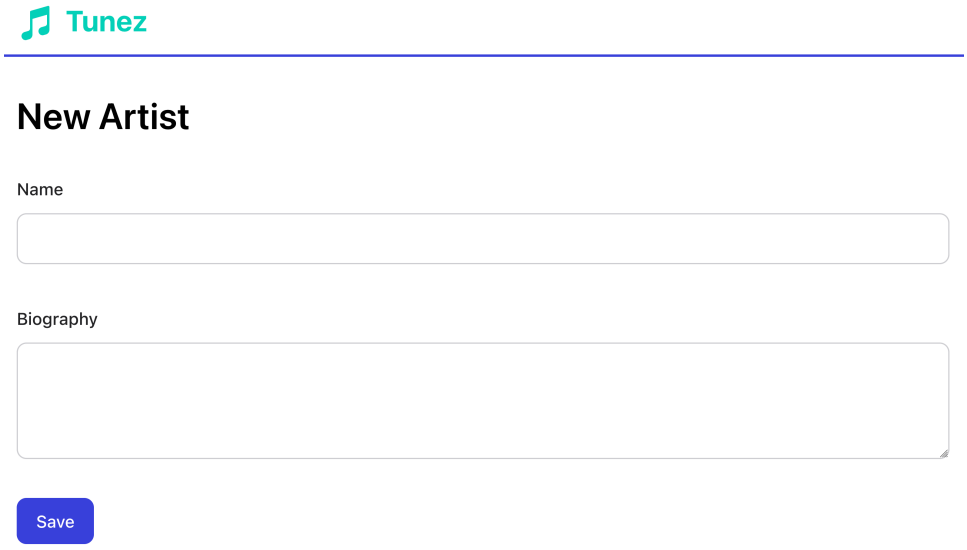
Now, instead of calling AshPhoenix.Form.for_create(Tunez.Music.Artist, :create), we can use a new function Tunez.Music.form_to_create_artist. This works for any code interface function, even for read actions, by prefixing form_to_ to the function name.

```
iex(5)> AshPhoenix.Form.for_create(Tunez.Music.Artist, :create)
#AshPhoenix.Form<resource: Tunez.Music.Artist, action: :create, ...>
iex(6)> Tunez.Music.form_to_create_artist()
#AshPhoenix.Form<resource: Tunez.Music.Artist, action: :create, ...>
```

The result is the same — you get an AshPhoenix.Form struct to validate and submit, as before — but the way you get it is a lot more consistent with other function calls.

Integrating a form into a liveview

The liveview for creating an Artist is the TunezWeb.Artists.FormLive module, located in lib/tunez_web/live/artists/form_live.ex. In the browser, you can view it by clicking the New Artist button on the artist catalog, or visiting /artists/new.



It *looks* good, but it's totally non-functional right now. We can use what we've learned so far about AshPhoenix.Form to make it work as we would expect.

It starts from the top — we want to build our initial form in the mount/3 function. Currently form is defined as an empty map, just to get the form to render. We can replace it with a function call to create the form, like we did in iex:

01/lib/tunez_web/live/artists/form_live.ex

```
def mount(_params, _session, socket) do
  form = Tunez.Music.form_to_create_artist()

  socket =
    socket
    |> assign(:form, to_form(form))
    # ...
```

The form has a phx-change event handler attached that will fire after every pause in typing on the form. This will send the "validate" event to the liveview, handled by the handle_event/3 function head with the first argument "validate".

01/lib/tunez_web/live/artists/form_live.ex

```
def handle_event("validate", %{"form" => _form_data}, socket) do
  {:noreply, socket}
end
```

It doesn't currently do anything, but we know we need to update the form in the socket with the data from the form.

01/lib/tunez_web/live/artists/form_live.ex

```
def handle_event("validate", %{"form" => form_data}, socket) do
  socket =
```

```
    update(socket, :form, fn form ->
      AshPhoenix.Form.validate(form, form_data)
    end)

  {:noreply, socket}
end
```

Lastly, we need to deal with form submission. The form has a phx-submit event handler attached that will fire when the user presses the Save button (or presses Enter). This will send the "save" event to the liveview. The event handler currently doesn't do anything either (we told you the form was non-functional!) but we can add code to submit the form with the form data.

We also need to handle the response after submission, handling both the success and failure cases. If the user submits invalid data then we want to show errors, otherwise we can go to the newly-added artist's profile page and display a success message.

01/lib/tunez_web/live/artists/form_live.ex

```
def handle_event("save", %{"form" => form_data}, socket) do
  case AshPhoenix.Form.submit(socket.assigns.form, params: form_data) do
    {:ok, artist} ->
      socket =
        socket
        |> put_flash(:info, "Artist saved successfully")
        |> push_navigate(to: ~p"/artists/#{artist}")

      {:noreply, socket}

    {:error, form} ->
      socket =
        socket
        |> put_flash(:error, "Could not save artist data")
        |> assign(:form, form)

      {:noreply, socket}
  end
end
```

Give it a try! Submit some invalid data, see the validation errors, correct the data, and submit the form again. It works great!

But what happens if you make a typo when entering data? No-one wants to read about Metlalica, do they? We need some way of editing artist records, and updating any necessary information.

Updating artists with the same code

When we set up the update actions in our Artist resource in Defining an update action, on page 18, we noted that it was pretty similar to the create action and

that the only real difference for update is that we need to provide the record being updated. The rest of the flow — providing data to be saved, saving it to the database — is exactly the same.

In addition, the web interface for editing an artist should be exactly the same as for creating an artist. The only difference will be that the form for editing has the artist data pre-populated on it, so that it can be modified, and the form for creating will be totally blank.

We can actually use the same TunezWeb.Artists.FormLive liveview module for both creating and updating records. The routes are already set up for this: clicking the Edit Artist button on the profile page will take you to that liveview.

```
[debug] MOUNT TunezWeb.Artists.FormLive
  Parameters: %{"id" => "[the artist UUID]"}
```

This won't be the case for all resources, all the time. You may need very different interfaces for creating and updating data. A lot of the time, though, this can be a neat way of building out functionality very quickly, and it can be changed later if your needs change.

The FormLive liveview will need to have different forms, depending on if an artist is being created or updated. Everything else can be the same because we still want to validate the data on keystroke, submit the form on form submission, and perform the same actions after submission.

As we build the form for create in the mount/3 function, we can add another mount/3 function head specifically for update, which sets form in the socket to a form built for a different action.

01/lib/tunez_web/live/artists/form_live.ex

```
def mount(%{"id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)
  form = Tunez.Music.form_to_update_artist(artist)

  socket =
    socket
    |> assign(:form, to_form(form))
    |> assign(:page_title, "Update Artist")

  {:ok, socket}
end

def mount(_params, _session, socket) do
  form = Tunez.Music.form_to_create_artist()
  # ...
```

This new function head (which has to come *before* the existing function head!) is differentiated by having an artist id in the params, just like the ShowLive

module did when we viewed the artist profile. It sets up the form specifically for the update action of the resource, using the loaded Artist record as the first argument. It also sets a different page title, and… that's all that has to change! Everything else should keep behaving exactly the same.

Save the liveview and test it out in your browser. You should now to be able to click Edit Artist, update the artist's details, save, and see the changes reflected back in their profile.

Deleting artist data

The last action we need to integrate is the destroy_artist code interface function for removing records from the database. In the UI, this is done from a button at the top of the artist profile page, next to the edit button. The button, located in the template for TunezWeb.Artists.ShowLive, will send the "destroy-artist" event when pressed.

01/lib/tunez_web/live/artists/show_live.ex

```
<.button_link kind="error" inverse phx-click="destroy-artist"
  data-confirm={"Are you sure you want to delete #{@artist.name}?"}>
  Delete Artist
</.button_link>
```

We've already loaded the artist record from the database when rendering the page, and stored it in socket.assigns, so you can fetch it out again and attempt to delete it with the Tunez.Music.destroy_artist function. The error return value would probably never be seen in practice, but just in case, we'll show the user a nice message anyway.

01/lib/tunez_web/live/artists/show_live.ex

```
def handle_event("destroy-artist", _params, socket) do
  case Tunez.Music.destroy_artist(socket.assigns.artist) do
    :ok ->
      socket =
        socket
        |> put_flash(:info, "Artist deleted successfully")
        |> push_navigate(to: ~p"/")

      {:noreply, socket}

    {:error, error} ->
      Logger.info("Could not delete artist '#{socket.assigns.artist.id}':
        #{inspect(error)}")

      socket =
        socket
        |> put_flash(:error, "Could not delete artist")

      {:noreply, socket}
  end
```

```
end
```

There are lots of different stylistic ways that this type of code could be written, but this is typically the way we would write it. If things go wrong, we want errors logged as to what happened, and we always want users to get feedback about what's going on.

And that's it! We've set up Ash in the Tunez app, and implemented a full CRUD interface for our first resource — and we haven't had to write very much code to do it.

We've learned a bit about the *declarative* nature of Ash. We didn't need to write functions that accepted parameters, processed them, saved the records, etc — we haven't needed to write any functions at all. We declared what our resource should look like, where data should be stored, and what our actions should do. Ash has handled the actual implementations for us.

We've also seen how AshPhoenix provides a tidy `Form` pattern for integration with web forms, allowing for a really streamlined integration with very little code.

In the next chapter, we'll look at building a second resource and how the two can be integrated together!

CHAPTER 2

Extending Resources with Business Logic

In the first chapter, we learned how to set up Ash within our Phoenix app, created our first resource for Artists within a domain, and built out a full web interface so that we could create, read, update and delete Artist records. This would be a great starting point for any application, to pick your most core domain model concept and build it out.

Now we can start fleshing out the domain model for Tunez a little bit more, because one resource does not a full application make. Having multiple resources, and connecting them together, will allow us to do things like querying and filtering based on related data. So, in the real world, artists release albums, right? Let's build a second resource representing an Album with a more complex structure, link them together, and learn some other handy features of working with declarative resources.

Resources and Relationships

Like we generated our Artist resource, we can start by using Ash's generators to create our basic Album resource. It's music-related, so it should also be part of the Tunez.Music domain:

```
$ mix ash.gen.resource Tunez.Music.Album --extend postgres
```

This will generate the resource file in lib/tunez/music/album.ex, as well as adding the new resource to the list of resources in the Tunez.Music domain module.

The next step, just like when we built our first resource, is to consider what kinds of attributes our new resource needs. What information should we record about an Album? Right now we probably care about:

- The artist who released the album
- The album name
- The year the album was released

- And an image of the album cover. That'll make Tunez look really nice!

Ash has a lot of inbuilt data types,[1] that can let you model just about anything. If we were building a resource for a product in a clothing store, we might want attributes for things like the item size, colour, brand name, and price. A listing on a real estate app might want to store the property address, the number of bedrooms and bathrooms, and the property size.

If none of the inbuilt data types cover what you need, you can also create custom or composite data types.[2] These can neatly wrap logic around discrete units of data, such as phone numbers, URLs, or latitude/longitude co-ordinates.

In the attributes block of the Album resource, we can start adding our new attributes:

02/lib/tunez/music/album.ex

```
attributes do
  uuid_primary_key :id

  attribute :name, :string do
    allow_nil? false
  end

  attribute :year_released, :integer do
    allow_nil? false
  end

  attribute :cover_image_url, :string

  create_timestamp :inserted_at
  update_timestamp :updated_at
end
```

The name and year_released attributes will be required, but the cover_image_url will be optional. We might not have a high-quality photo on hand for every album, but we can add them in later when we get them.

We haven't added any field to represent the artist though — and that's because it's not going to be just a normal attribute. It's going to be a *relationship*.

Defining Relationships

Relationships, also known as *associations*, are how we describe connections between resources in Ash. There are a couple of different relationship types

1. https://hexdocs.pm/ash/Ash.Type.html#module-built-in-types
2. https://hexdocs.pm/ash/Ash.Type.html#module-defining-custom-types

we can choose from, based on the numbers of resources involved on each side:

- has_many relationships relate *one* resource to *many* other resources. These are really common — a User can have many Posts, or a Book can have many Chapters. These don't store any data on the *one* side of the relationship, but each of the items on the *many* side will have a reference back to the *one*.
- belongs_to relationships relate *one* resource to *one* parent/containing resource. They're usually the inverse of a has_many — in the examples above, the resource on the *many* side would typically belong to the *one* resource. A Chapter belongs to a Book, and a Post belongs to a User. The resource belonging to another will have a reference to the related resource, eg. a Chapter will have a book_id attribute, referencing the id field of the Book resource.
- has_one relationships are less common, but are similar to belongs_to relationships. They relate *one* resource to *one* other resource, but differ in which end of the relationship holds the reference to the related record. For a has_one relationship, the related resource will have the reference. A common example of a has_one relationship is Users and Profiles — a User could have one Profile, but the Profile resource is what holds a user_id attribute.
- many_to_many relationships, as the name suggests, relate many resources to many other resources. These are where you have two pools of different objects, and can link any two resources between the pools. Tags are a really common example — a Post can have many Tags applied to it, and a Tag can also apply to many different Posts.

In our case, we'll be using belongs_to and has_many relationships — an Artist has_many albums, and an Album belongs_to an artist.

In code, we define these in a separate top-level relationships block in each resource. In the Artist resource, in lib/tunez/music/artist.ex, we can add a relationship with Albums:

02/lib/tunez/music/artist.ex
```
relationships do
  has_many :albums, Tunez.Music.Album
end
```

And in the Album resource in lib/tunez/music/album.ex, a relationship back to the Artist resource:

02/lib/tunez/music/album.ex
```
relationships do
  belongs_to :artist, Tunez.Music.Artist do
    allow_nil? false
  end
end
```

Now that our resource is set up, generate a database migration for it, using the ash.codegen mix task.

```
$ mix ash.codegen create_albums
```

This will generate a new Ecto migration in priv/repo/migrations/[timestamp]_create_albums.exs to create the albums table in the database, including a foreign key representing the relationship — this will link an artist_id field on the albums table to the id field on artists table. A snapshot JSON file will also be created, representing the current state of the Album resource.

The migration *doesn't* contain a function call to create a database index for the foreign key, though, and PostgreSQL doesn't create indexes for foreign keys by default. To tell Ash to create an index for the foreign key, you can customize the *reference* of the relationship, as part of the postgres block in the resource.

02/lib/tunez/music/album.ex
```
postgres do
  # ...
  references do
    reference :artist, index?: true
  end
end
```

This changes the database, so you'll need to codegen another migration for it (or delete the CreateAlbums migration and snapshot that we just generated, and generate them again).

If you're happy with the migrations, run them:

```
$ mix ash.migrate
```

And now we can start adding functionality. A lot of this will seem pretty familiar from building out the Artist interface, so we'll cover it quickly. There are a few new interesting parts, however, due to the added relationship, so let's dig right in.

Album actions

If we look at an Artist's profile page in the app, we can see a list of their albums, so we're going to need some kind of read action on the Album resource,

to read the data to display. There's also a button to add a new album, at the top of the album list, so we'll need a create action; and each album has Edit and Delete buttons next to the title, so we'll write some update and destroy actions as well.

We can add those to the Album resource pretty quickly:

02/lib/tunez/music/album.ex

```
actions do
  defaults [:read, :destroy]

  create :create do
    accept [:name, :year_released, :cover_image_url, :artist_id]
  end

  update :update do
    accept [:name, :year_released, :cover_image_url]
  end
end
```

We don't have any customizations to make to the default implementation of read or destroy, so we can define those as default actions. You might be thinking, but won't we need to customize the read action, to only show albums for a specific artist? We actually don't! When we load an artist's albums on their profile page, which we'll see how to do shortly, we won't be calling this action directly — we'll be asking Ash to load the albums through the albums relationship on the Artist resource, which will automatically apply the correct filter.

We do have tweaks for the create and update actions, though — for the accept list, of attributes that can be set when calling those actions. When creating a record, it makes sense to need to set the artist_id for an album, otherwise it won't be set at all! When updating an album, though, does it really need to be changeable? Can we see ourselves creating an album via the wrong artist profile and then needing to change it later? It seems unlikely, so we don't need to accept the artist_id attribute in the update action.

We'll also add code interface definitions for our actions, to make them easier to use in an iex console, and easier to read in our liveviews. Again, these go in our Tunez.Music domain module, with the resource definition.

02/lib/tunez/music.ex

```
resources do
  # ...
  resource Tunez.Music.Album do
    define :create_album, action: :create
    define :get_album_by_id, action: :read, get_by: :id
    define :update_album, action: :update
    define :destroy_album, action: :destroy
  end
```

```
end
```

Similar to artists, we've provided some sample album content for you to play around with. To import it, you can run the following on the command line:

```
$ mix run priv/repo/seeds/02-albums.exs
```

This will populate a handful of albums for each of the sample artists we seeded in code on page 15.

You can also uncomment the second seed file in the mix seed alias, in the aliases function in mix.exs:

02/mix.exs

```
defp aliases do
  [
    setup: ["deps.get", "ash.setup", "assets.setup", "assets.build", ...],
    "ecto.setup": ["ecto.create", "ecto.migrate"],
    seed: [
      "run priv/repo/seeds/01-artists.exs",
      "run priv/repo/seeds/02-albums.exs",
      # "run priv/repo/seeds/08-tracks.exs"
    ],
    # ...
```

Now you can run mix seed to reset the seeded artist and album data in your database. Currently the seed scripts aren't *idempotent* (you can't re-run them repeatedly) due to how we've set up our Album -> Artist relationship, but we'll address that in Deleting All of the Things, on page 49. And now we can start connecting the pieces, to view and manage the album data in our liveviews.

Creating and updating albums

Our Artist page has a button on it to add a new album for that artist. This links to the TunezWeb.Albums.FormLive liveview module, and renders a form template similar to the artist form, with text fields for entering data. We can use AshPhoenix to make this template functional, the same way we did for artists.

First, we construct a new form for the Album.create action, in mount/3:

02/lib/tunez_web/live/albums/form_live.ex

```
def mount(_params, _session, socket) do
  form = Tunez.Music.form_to_create_album()

  socket =
    socket
    |> assign(:form, to_form(form))
    ...
```

We validate the form data and update the form in the liveview's state, in the "validate" handle_event/3 event handler:

02/lib/tunez_web/live/albums/form_live.ex
```
def handle_event("validate", %{"form" => form_data}, socket) do
  socket =
    update(socket, :form, fn form ->
      AshPhoenix.Form.validate(form, form_data)
    end)

  {:noreply, socket}
end
```

We submit the form in the "save" handle_event/3 event handler, and process the return value:

02/lib/tunez_web/live/albums/form_live.ex
```
def handle_event("save", %{"form" => form_data}, socket) do
  case AshPhoenix.Form.submit(socket.assigns.form, params: form_data) do
    {:ok, album} ->
      socket =
        socket
        |> put_flash(:info, "Album saved successfully")
        |> push_navigate(to: ~p"/artists/#{album.artist_id}")

      {:noreply, socket}

    {:error, form} ->
      socket =
        socket
        |> put_flash(:error, "Could not save album data")
        |> assign(:form, form)

      {:noreply, socket}
  end
end
```

And finally, we add another function head for mount/3, so we can differentiate between viewing the form to *add* an album, and viewing the form to *edit* an album, based on whether or not album_id is present in the params:

02/lib/tunez_web/live/albums/form_live.ex
```
def mount(%{"id" => album_id}, _session, socket) do
  album = Tunez.Music.get_album_by_id!(album_id)
  form = Tunez.Music.form_to_update_album(album)

  socket =
    socket
    |> assign(:form, to_form(form))
    |> assign(:page_title, "Update Album")

  {:ok, socket}
end
```

```
def mount(_params, _session, socket) do
  form = Tunez.Music.form_to_create_album()
  ...
```

If this was a bit *too* fast, there is a much more thorough rundown on how this code works in Creating artists with AshPhoenix.Form, on page 24.

Using artist data on the album form

There's one thing missing from this form, that will stop it from working as we expect to manage Album records — there's no mention at all of the Artist that the album should belong to. There's a field to enter an artist on the form, but it's disabled.

We do *know* which artist the album should belong to, though. We clicked the button to add an album on a specific artist page, the album should be for that artist! In the server logs in your terminal, you'll see that we do have the artist ID as part of the params to the FormLive liveview:

```
[debug] MOUNT TunezWeb.Albums.FormLive
  Parameters: %{"artist_id" => "an-artist-id"}
```

We can use this ID to load the artist record, show the artist details on the form, and relate the artist to the album in the form.

In the second mount/3 function head, for the create action, we can load the artist record using Tunez.Music.get_artist_by_id, like we do on the artist profile page. The artist can be assigned to the socket alongside the form.

02/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"artist_id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)
  form = Tunez.Music.form_to_create_album()

  socket =
    socket
    |> assign(:form, to_form(form))
    |> assign(:artist, artist)
    ...
```

In the first mount/3 function head, for the update action, we have the artist ID stored on the album record we load. We can use it to load the Artist record in a similar way:

02/lib/tunez_web/live/albums/form_live.ex

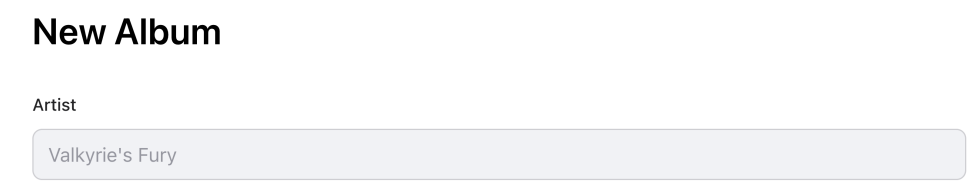
```
def mount(%{"id" => album_id}, _session, socket) do
  album = Tunez.Music.get_album_by_id!(album_id)
  artist = Tunez.Music.get_artist_by_id!(album.artist_id)
  form = Tunez.Music.form_to_update_album(album)

  socket =
```

```
    socket
    |> assign(:form, to_form(form))
➤   |> assign(:artist, artist)
    ...
```

Now that we have an artist record assigned in the liveview, we can show the artist name in the disabled field, in render/3:

02/lib/tunez_web/live/albums/form_live.ex

```
<.input name="artist_id" label="Artist" value={@artist.name} disabled />
```



This doesn't actually add the artist info to the form params, though, so we'll still get an error when submitting the form for a new album, even if all of the data is valid. There are two ways we can address this: we *could* manually update the form data before submitting the form, adding the artist_id from the artist record already in the socket.

```
def handle_event("save", %{"form" => form_data}, socket) do
  form_data = Map.put(form_data, "artist_id", socket.assigns.artist.id)
  ...
```

This is easy to reason about, but feels messy. This code also runs when submitting the form for both creating *and* updating an album, and the update action on our Album resource specifically does *not* accept an artist_id attribute. Submitting the form won't raise an error — AshPhoenix throws away any data that won't be accepted by the underlying action — but it's a sign that we're probably doing things wrong.

Instead, we'll look at building the form for creating an album slightly differently, to pre-populate the artist ID. The form_to_create_album function is auto-generated from our create_album code interface, defined in the Tunez.Music domain module:

02/lib/tunez/music.ex

```
resource Tunez.Music.Album do
  define :create_album, action: :create
  # ...
end
```

Any changes we make to the code interface won't only affect the generated form_to_ function, they'll update the create_album function too. We're not currently

using that function, but we might want to later! Instead, we can customize *only* the form_to_create_album action by using the forms[3] DSL, from the AshPhoenix domain extension.

By specifying a form with the same name as the code interface, we can then add a list of args that are required when building the form, that will be submitted with the rest of the form data.

02/lib/tunez/music.ex

```
defmodule Tunez.Music do
  use Ash.Domain, otp_app: :tunez, extensions: [AshPhoenix]

  forms do
    form :create_album, args: [:artist_id]
  end

  # ...
```

This changes the signature of the generated function, which you can see if you recompile your app in iex:

```
iex(1)> h Tunez.Music.form_to_create_album

             def form_to_create_album(artist_id, form_opts \\ [])

Creates a form for the create action on Tunez.Music.Album.
```

We can now update how we call form_to_create_action, specifying the artist ID as the first argument, and it will be used when submitting the form.

02/lib/tunez_web/live/albums/form_live.ex

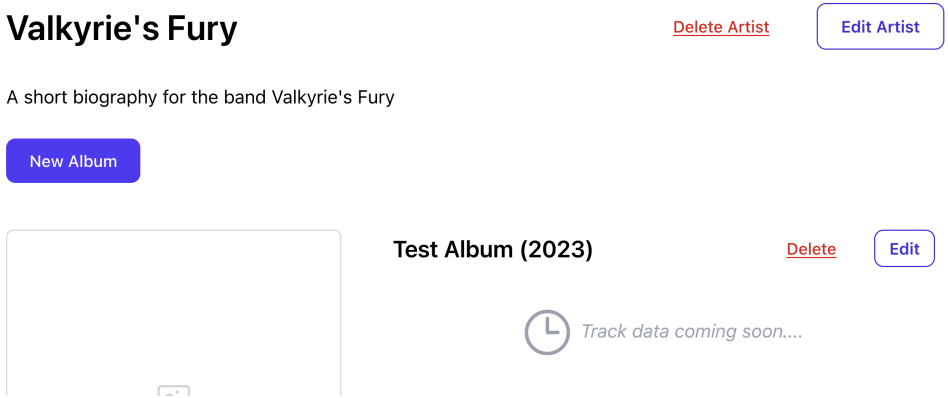
```
def mount(%{"artist_id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)
  form = Tunez.Music.form_to_create_album(artist.id)
  ...
```

This is a common pattern to use when you want to provide data to an action via a form, but it shouldn't be editable by the user. Even users are being sneaky in their browser dev tools and adding form fields with data they shouldn't be editing, they get overwritten with the correct value we specified earlier, so nothing nefarious can happen. And now our TunezWeb.Album.FormLive form should work properly, for creating album data.

Loading Related Resource Data

On the profile page for an Artist in TunezWeb.Artists.ShowLive, we want to show a list of albums released by that artist. It's currently populated with placeholder data:

3. https://hexdocs.pm/ash_phoenix/dsl-ashphoenix.html#forms



And this data is defined in the handle_params/3 callback:

02/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(%{"id" => artist_id}, _url, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id)

  albums = [
    %{
      id: "test-album-1",
      name: "Test Album",
      year_released: 2023,
      cover_image_url: nil
    }
  ]

  socket =
    socket
    |> assign(:artist, artist)
    |> assign(:albums, albums)
    |> assign(:page_title, artist.name)

  {:noreply, socket}
end
```

The Edit button for the album will still take you to the form we just built, but it will error because the album ID doesn't match a valid album in the database!

Because we've defined albums as a relationship in our Artist resource, we can automatically *load* the data in that relationship, similar to an Ecto preload. All actions support an extra argument of *options*, and one of the options for read actions[4] is load — a list of relationships we want to load alongside the requested data. This will use the primary read action that we defined on the Album resource, but include the correct filter to only load albums for the artist specified.

4. https://hexdocs.pm/ash/Ash.html#read/2

To do this, update the call to get_artist_by_id! to include loading the albums relationship, and remove the hardcoded albums:

02/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(%{"id" => artist_id}, _url, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id, load: [:albums])

  socket =
    socket
    |> assign(:artist, artist)
    |> assign(:page_title, artist.name)

  {:noreply, socket}
end
```

We do also need to update a little bit of the template, as it referred to the @albums assign (which is now deleted). In the render/1 function, we currently iterate over @albums and render album details for each. This needs to be updated, to render albums from the @artist instead:

02/lib/tunez_web/live/artists/show_live.ex

```
<li :for={album <- @artist.albums}>
  <.album_details album={album} />
</li>
```

Now when we view the profile page for one of our sample artists, we should be able to see their actual albums, complete with album covers. Neat!

We can use load to simplify how we loaded the artist for the album on the Album edit form, as well. Instead of making a second request to load the artist after loading the album, they can be combined into one call:

02/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"id" => album_id}, _session, socket) do
  album = Tunez.Music.get_album_by_id!(album_id, load: [:artist])
  form = Tunez.Music.form_to_update_album(album)

  socket =
    socket
    |> assign(:form, to_form(form))
    |> assign(:artist, album.artist)
    ...
```

The album data on the artist profile looks a little bit funny though — the albums aren't in any kind of order on the page. We should probably show them in chronological order, with the most recent album release listed first. We can do this by defining a *sort* for the album relationship, using the sort option[5] on the :albums relationship iin Tunez.Music.Artist.

5. https://hexdocs.pm/ash/dsl-ash-resource.html#relationships-has_many-sort

02/lib/tunez/music/artist.ex

```
relationships do
  has_many :albums, Tunez.Music.Album do
    sort year_released: :desc
  end
end
```

This takes a list of fields to sort by, and will sort in ascending order by default. To flip the order, you can use a keyword list instead, with the field names as keys and either :asc or :desc as the value for each key, just like Ecto.

Now if we reload an artist's profile, we should see the albums being displayed in chronological order, most recent first. That's much more informative!

Structured Data with Validations and Identities

Tunez can now accept form data that should be more structured, instead of just text. We're also looking at data in a smaller scope. Instead of "any artist in the world that ever was", which is a massive data set; we're looking at albums for any individual artist, which is a much smaller and well-defined list.

Let's set some stricter rules for this data, for better data integrity.

Consistent data with validations

With Albums, we want users to enter a valid year for an album's year_released attribute, instead of any old integer; and a valid-looking image URL for the cover_image_url attribute. We can enforce these rules with *validations*.

Any defined validations are checked when calling an action, *before* the core functionality (eg. saving or deleting) is run; and if any of the validations fail, the action will abort and return an error. We've seen implicit cases of this already, when we declared that some attributes were allow_nil? false — as well as setting the database field for the attribute to be non-nullable, Ash also validates that the value is present before it even *gets* to the database.

Validations can be added to resources either for an individual action, or globally for the entire resource. In our case, we want to ensure that the data is valid at all times, so we'll add global validations by adding a new top-level validations block in the Album resource:

02/lib/tunez/music/album.ex

```
defmodule Tunez.Music.Album do
  # ...

  validations do
    # Validations will go in here
```

```
  end
end
```

We'll add two validations to this block, one for year_released, and one for cover_image_url. Ash provides a lot of built-in validations,[6] and two of them are relevant here: numericality and match.

For year_released, we want to validate that the user enters a number between, say, 1950 (arbitrarily-chosen year) and next year (to allow for albums that have been announced but not released), but we should only validate the field if the user has actually entered data. This is written like so:

02/lib/tunez/music/album.ex

```
validations do
  validate numericality(:year_released,
            greater_than: 1950,
            less_than_or_equal_to: &__MODULE__.next_year/0
          ),
          where: [present(:year_released)],
          message: "must be between 1950 and next year"
end
```

Ash will accept any zero-arity (no-argument) function reference here. The next_year function doesn't exist, so we'll add it to the very end of the the Album module:

02/lib/tunez/music/album.ex

```
def next_year, do: Date.utc_today().year + 1
```

For cover_image_url, we'll add a regular expression to make sure user enters what *looks* like an image URL — either a fully-qualified URL, or a path to one of the sample album covers in the priv/static/images folder. This isn't comprehensive by any means — in a real-world app, we'd likely be implementing a file uploader, verifying that the uploaded files were valid images, etc. — but for our use case, it'll solve copy-paste mistakes, or entering nonsense.

02/lib/tunez/music/album.ex

```
validations do
  # ...
  validate match(:cover_image_url,
            ~r"(^https://|/images/).+(\.png|\.jpg)$"
          ),
          where: [changing(:cover_image_url)],
          message: "must start with https:// or /images/"
end
```

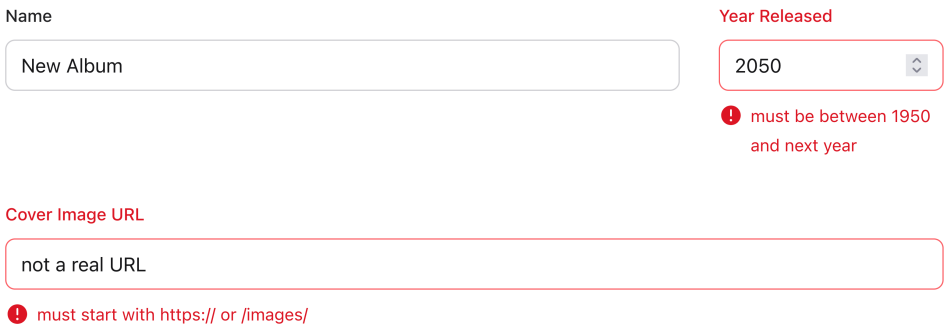
6. https://hexdocs.pm/ash/Ash.Resource.Validation.Builtins.html

For a little optimization, we'll also add a check that only runs the validation if the value is *changing*, using the `changing/1`[7] function in the `where` condition of the validation.

If you want to run a validation only for one specific action, you can put the validation directly in the action, instead of in the global `validations` block.

To run validations for all actions of a specific type, eg. all `create` actions, you can put them in the global `validations` block and use the `on` option[8] to specify the types of actions it should apply to.

We don't need to do anything to integrate these validations into Album actions, or into the forms in our views. Because they're global validations, they apply for every `create` and `update` action, and because the forms in our liveviews are built for actions, they'll automatically be included. Entering invalid data in the album form will now show validation errors to our users, letting them know what to fix:



Unique data with identities

There's one last feature we can add, for a better user experience on this form. Some artists have a *lot* of albums, and it would be good to ensure that duplicate albums don't accidentally get entered. Maintaining data integrity, especially with user-editable data, is important — sites like Wikipedia don't allow multiple pages with the exact same name, for example, they have to be disambiguated in some way.

Tunez will consider an album to be a duplicate if it has the same `name` as another album by the same artist, ie. the combination of `name` and `artist_id`

7. https://hexdocs.pm/ash/Ash.Resource.Validation.Builtins.html#changing/1
8. https://hexdocs.pm/ash/dsl-ash-resource.html#validations-validate-on

should be unique for every album in the database. (We'll assume that separate versions of albums with the same name get suffixes attached, like "Remastered" or "Live" or "Taylor's Version".) For ensuring this uniqueness, we can use an *identity* on our resource.

Ash defines an identity[9] as any attribute, or combination of attributes, that can uniquely identify a record. A primary key is a natural and automatically-generated identity, but our data may lend itself to other identities as well.

To add the new identity to our resource, add a new top-level identities block to the Album resource. An identity has a name, and a list of attributes that make up that identity. We can also specify a message to display on identity violations:

02/lib/tunez/music/album.ex

```
identities do
  identity :unique_album_names_per_artist, [:name, :artist_id],
    message: "already exists for this artist"
end
```

The way identities are handled depends on the data layer being used. Because we're using AshPostgres, the identity will be handled at the database level as a *unique index* on the two database fields, albums.name and albums.artist_id.

To create the index in the database, we can generate migrations after adding the identity to the Album resource:

```
$ mix ash.codegen add_unique_album_names_per_artist
```

This is the first time we've modified a resource and then generated migrations, so it's worth taking a bit of a closer look.

Like the previous times we've generated migrations, AshPostgres has generated a snapshot file representing the current state of the Album resource. It also created a new migration, which has all of the differences between the last snapshot from when we created the resource, and the brand-new snapshot:

02/priv/repo/migrations/[timestamp]_add_unique_album_names_per_artist.exs

```
def up do
  create unique_index(:albums, [:name, :artist_id],
           name: "albums_unique_album_names_per_artist_index"
         )
end

def down do
  drop_if_exists unique_index(:albums, [:name, :artist_id],
                   name: "albums_unique_album_names_per_artist_index"
```

9. https://hexdocs.pm/ash/identities.html

```
                    )
end
```

Ash correctly worked out that the only difference that required database changes was the new identity, so it created the correct migration to add and remove the unique index we need. Awesome!

Run the migration generated:

```
$ mix ash.migrate
```

And now we can test out the changes on the album form. Create an album with a specific name, and then try to create another one for the same artist with the same name — you should get a validation error on the `name` field, with the message we specified for the identity.

Deleting All of the Things

We'll round out the CRUD interface for Albums with the `destroy` action. We might not need to invoke it too much while using Tunez, but keeping our data clean and accurate is always an important priority.

While building the Album resource, we've also accidentally introduced a bug around Artist deletion, so we should address that as well.

Deleting album data

Deleting albums is done from the artist's profile page, `TunezWeb.Artists.ShowLive`, via a button next to the name of the album.

Clicking the icon will send the "destroy-album" event to the liveview. In the event handler, we'll fetch the album record from the list of albums we already have in memory, and then delete it. It's a little bit verbose, but it saves another round trip to the database to look up the album record. Like with artists, we also need to handle both the success and error cases:

02/lib/tunez_web/live/artists/show_live.ex

```
def handle_event("destroy-album", %{"id" => album_id}, socket) do
  case Tunez.Music.destroy_album(album_id) do
    :ok ->
      socket =
        socket
        |> update(:artist, fn artist ->
          Map.update!(artist, :albums, fn albums ->
            Enum.reject(albums, &(&1.id == album_id))
          end)
        end)
        |> put_flash(:info, "Album deleted successfully")
```

```
      {:noreply, socket}

    {:error, error} ->
      Logger.info("Could not delete album '#{album_id}': #{inspect(error)}")

      socket =
        socket
        |> put_flash(:error, "Could not delete album")

      {:noreply, socket}
  end
end
```

We've almost finished the initial implementation for albums! There's a bug in our Album implementation though — if you try to delete an artist that has albums, you'll see what we mean. This also affects our seed scripts: we can't reseed the database because we can't delete the seeded artists that have albums. We'll fix that now!

Cascading deletes with AshPostgres

When we defined our Album resource, we added a belongs_to relationship to relate it to Artists:

02/lib/tunez/music/album.ex

```
relationships do
  belongs_to :artist, Tunez.Music.Artist do
    allow_nil? false
  end
end
```

When we generated the migration for this resource in Defining Relationships, on page 34, it created a foreign key in the database, linking the artist_id field on the albums table to the id field on the artists table:

02/priv/repo/migrations/[timestamp]_create_albums.exs

```
def up do
  create table(:albums, primary_key: false) do
    # ...
    add :artist_id,
        references(:artists,
          column: :id,
          name: "albums_artist_id_fkey",
          type: :uuid,
          prefix: "public"
        )
  end
end
```

What we *didn't* define, however, was what should happen with this foreign key value when artists are deleted — if there are three albums with artist_id = "abc123", and artist abc123 is deleted, what happens to those albums?

The default behaviour, as we've seen, is to prevent the deletion from happening. This is verified by looking at the server logs when you try to delete one of the artists that this affects:

```
[info] Could not delete artist '«uuid»': %Ash.Error.Invalid{bread_crumbs:
["Error returned from: Tunez.Music.Artist.destroy"], changeset: "#Changeset<>",
errors: [%Ash.Error.Changes.InvalidAttribute{field: :id, message: "would leave
records behind", private_vars: [constraint: :foreign, constraint_name:
"albums_artist_id_fkey"], ...
```

Because an album doesn't make sense without an artist (we can say the albums are *dependent* on the artist), we should delete all of an artist's albums when we delete an artist. There are two ways we can go about this, each with its own pros and cons:

- We can delete the dependent records in code — in the destroy action for an artist, we can call the destroy action on all of the artist's albums as well. It's very explicit what's going on, but it can be *really* slow (relatively speaking). Sometimes it's a necessary evil though, if you need to run business logic in each of the dependent destroy actions.
- Or we can delete the dependent records in the database, by specifying the ON DELETE behaviour[10] of the foreign key that raised the error. This is super-fast, but it can be a little unexpected if you don't know it's happening. You don't get the chance to run any business logic in your app's code — but if you don't need to, this is easily the preferred option.

Which one you use really depends on the requirements of the app you're building, and as the requirements of your app change, you might need to change the behaviour. For now we'll go with the quick ON DELETE option, which is to delete the dependent records in the database (option 2).

AshPostgres lets us specify the ON DELETE behaviour for a foreign key by configuring the custom reference in the postgres block[11] of our resource. This goes on the resource that has the foreign key, ie. in this case, the Tunez.Music.Album resource:

02/lib/tunez/music/album.ex

```
postgres do
  # ...
```

10. https://www.postgresql.org/docs/current/sql-createtable.html#SQL-CREATETABLE-PARMS-REFERENCES
11. https://hexdocs.pm/ash_postgres/dsl-ashpostgres-datalayer.html#postgres-references

```
    references do
➤     reference :artist, index?: true, on_delete: :delete
    end
  end
```

This will make a structural change to our database, so we need to generate migrations and run them:

```
$ mix ash.codegen configure_reference_for_album_artist_id
$ mix ash.migrate
```

This will generate a migration that modifies the existing foreign key, setting on_delete: :delete_all. Running the migration sets the ON DELETE clause on the artist_id field:

```
tunez_dev=# \d albums
«definition of the columns and indexes of the table»
Foreign-key constraints:
    "albums_artist_id_fkey" FOREIGN KEY (artist_id) REFERENCES
      artists(id) ON DELETE CASCADE
```

And now we can delete artists again, even if they have albums; no error occurs and no data is left behind.

Our albums are really shaping up! They're not complete — we'll look at adding track listings in Chapter 8, Fun With Nested Forms, on page 179 — but for now they're pretty good, so we can step back and revisit our artist form.

What if we needed to make changes to the data we call an action with, before saving it into the data layer? The UI in our form might not *exactly* match the attributes we want to store, or we might need to format the data, or conditionally set attributes based on other attributes. We can look at making these kinds of modifications with *changes*.

Changing Data Within Actions

We've been using some built-in changes already in Tunez, without even realizing it, for inserted_at and updated_at timestamps on our resources. We didn't write any code for them, but Ash takes care of setting them to the current time. Both timestamps are set when calling any create action, and updated_at is set when calling any update action.

Like validations, changes can be defined at both the top-level of a resource, or at an individual action level. The implementation for timestamps *could* look like this:

```
changes do
  change set_attribute(:inserted_at, &DateTime.utc_now/0), on: [:create]
```

```
  change set_attribute(:updated_at, &DateTime.utc_now/0)
end
```

By default, global changes will run on any create or update action, which is why we wouldn't have to specify an action type for :updated_at above. They *can* be run on destroy actions, but only when opting-in by specifying on: [:destroy] on the change.

There's quite a few built-in changes[12] you can use in your resources, or you can add your own, either inline or with a custom module. We'll go through what it looks like to build one inline, and then how it can be extracted to a module for re-use.

Defining an inline change

Over time, artists go through phases, and sometimes change their names after re-branding, lawsuits, or lineup changes. Let's track updates to an artist's name over time, by keeping a list of all of the previous values that the name field has had, with a new change function.

This list will be stored in a new attribute, called previous_names, so we'll list it as an attribute in the Artist resource. It'll be a list, or *array*, of the previous names, and default to an empty list for new artists:

02/lib/tunez/music/artist.ex

```
attributes do
  # ...
  attribute :previous_names, {:array, :string} do
    default []
  end
  # ...
end
```

Generate a migration to add the new attribute to the database, and run it:

```
$ mix ash.codegen add_previous_names_to_artists
$ mix ash.migrate
```

We only need to run this change when the Artist form is submitted to update an Artist, so we'll add the change within the update action. (If your Artist resource is using defaults to define it's actions, you'll need to remove :update from that list and define the action separately.) The change macro can take a few different forms of arguments, the simplest being a two-argument anonymous function that takes and returns an Ash.Changeset:

12. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html

02/lib/tunez/music/artist.ex
```
actions do
  # ...
  update :update do
    accept [:name, :biography]

    change fn changeset, _context ->
      changeset
    end
  end
end
```

Inside this anonymous function we can make any changes to the changeset we want, including deleting data, changing relationships, adding errors, and more. If we set any errors in the changeset, they will stop the rest of the action from taking place and return the changeset to the user.

To implement the logic we want, we'll use some of the functions from Ash.Changeset[13] to read both the old and new name values from the changeset, and update the previous_names attribute where applicable:

02/lib/tunez/music/artist.ex
```
change fn changeset, _context ->
         new_name = Ash.Changeset.get_attribute(changeset, :name)
         previous_name = Ash.Changeset.get_data(changeset, :name)
         previous_names = Ash.Changeset.get_data(changeset, :previous_names)

         names =
           [previous_name | previous_names]
           |> Enum.uniq()
           |> Enum.reject(fn name -> name == new_name end)

         Ash.Changeset.change_attribute(changeset, :previous_names, names)
       end
```

Like calling actions, the change macro also accepts an optional second argument of options for the change. Because we only need to update previous_names if the name field is actually being modified, we'll add a changing/1[14] validation for the change function with a where check:

02/lib/tunez/music/artist.ex
```
change fn changeset, _context ->
         # ...
       end,
       where: [changing(:name)]
```

13. https://hexdocs.pm/ash/Ash.Changeset.html
14. https://hexdocs.pm/ash/Ash.Resource.Validation.Builtins.html#changing/1

If the validation fails, the change function is skipped, and the previous names won't be updated. That'll save a few CPU cycles!

There's one other small change we need to make, for this change function to work. By default, Ash will try to do as much work as possible in the data layer instead of in memory, via a concept called *atomics*. Because we've written our change functionality as imperative code, instead of in a data-layer-compatible way, we'll need to disable atomics for this `update` action with the `require_atomic?`[15] option.

02/lib/tunez/music/artist.ex
```
update :update do
  require_atomic? false

  # ...
end
```

We'll dig into atomics, and how to write changes atomically, later in chapter 10 on page 249.

Defining a change module

The inline version of the `previous_names` change works, but it's a bit long and imperative, smack-bang in the middle of our declarative resource. Imagine if we had a complex resource with a lot of attributes and changes, it'd be really hard to navigate and handle! And what if we wanted to apply this same record-previous-values logic to something else, like users who can change their usernames? Let's extract the logic out into a *change module*.

A change module is a standalone module that uses `Ash.Resource.Change`.[16] Its main access point is the `change/3` function, which has a similar function signature as the anonymous change function we defined earlier, but with an added second `opts` argument. We can move the content of the anonymous change function, and insert it directly into a new `change/3` function in a new change module:

02/lib/tunez/music/changes/update_previous_names.ex
```
defmodule Tunez.Music.Changes.UpdatePreviousNames do
  use Ash.Resource.Change

  @impl true
  def change(changeset, _opts, _context) do
    # The code previously in the body of the anonymous change function
  end
end
```

15. https://hexdocs.pm/ash/dsl-ash-resource.html#actions-update-require_atomic?
16. https://hexdocs.pm/ash/Ash.Resource.Change.html

And update the change call in the update action to point to the new module instead:

02/lib/tunez/music/artist.ex

```
update :update do
  require_atomic? false
  accept [:name, :biography]
  change Tunez.Music.Changes.UpdatePreviousNames, where: [changing(:name)]
end
```

A shorter and easier-to-read resource isn't the only reason to extract changes into their own modules. Change modules can define their own options and interface, and validate their usage at compile time. To reuse the current UpdatePreviousNames module, we might want to make the field names configurable instead of hardcoded to name and previous_names, and have a flag for allowing duplicate values or not. Change modules also have a performance benefit during development, by breaking compile-time dependencies between the resources and the code in the change functions. This makes it faster to recompile your app after modification!

Details on configuring and validating the interface for change modules using the Spark[17] library are a bit too much to go into here, but built-in changes like Ash.Resource.Change.SetAttribute[18] are a great way to see how they can be implemented.

Changes run more often than you might think!

Changes aren't *only* run when actions are called. When forms are tied to actions, like our update action is tied to the Artist edit form in the web interface, the pre-persistence steps like validations and changes are run multiple times:

- When building the initial form
- During any authorization checks (covered in Introducing Policies, on page 123)
- On every validation of the form
- And when actually submitting the form or calling the action.

Because of this, changes that are time-consuming, or have side effects such as calling external APIs, should be wrapped in *hooks* such as Ash.Changeset.before_action or Ash.Changeset.after_action — these will only be called immediately before or after the action is run.

17. https://hexdocs.pm/spark/
18. https://github.com/ash-project/ash/blob/main/lib/ash/resource/change/set_attribute.ex

If we wanted to do this for the UpdatePreviousNames change module, it would look like this:

```
def change(changeset, _opts, _context) do
  Ash.Changeset.before_action(changeset, fn changeset ->
    # The code previously in the body of the function
    # It can still use any `opts` or `context` passed in to the top-level
    # change function, as well.
  end)
end
```

The anonymous function set as the before_action would only run once — when the form is submitted — but it would still have the power to set errors on the changeset to prevent the changes from being saved, if necessary.

Setting attributes in a 'before_action' hook will bypass validations!

A function defined as a before_action will only run *right before save* — *after* validations of the action have been run — so it's possible to get your data into an invalid state in the database. If you validate that an album's year_released must be in the past, but then call Ash.Changeset.change_attribute(changeset, :year_released, 2050) in your before_action function, that year 2050 will happily be saved into the database. Ash will show a warning at runtime if you do this, which is helpful.

If you want to force any validation to run *after* before_action hooks, you can use the before_action?[19] option on the validation. Or if you simply want to silence the warning because you're fine with skipping the validation, replace your call to change_attribute with force_change_attribute instead.

Rendering the previous names in the UI

To finish this feature off, we'll show any previous names that an artist has had, on their profile page.

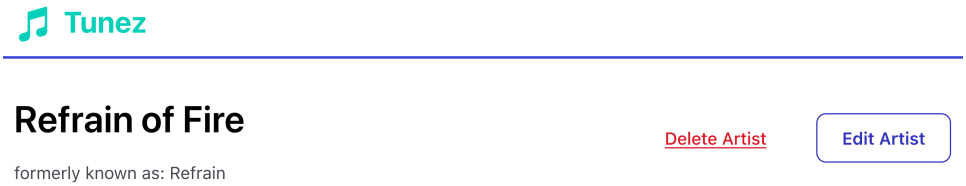
In TunezWeb.Artists.ShowLive, we'll add the names printed out, as part of the <header> block in the render/1 function:

02/lib/tunez_web/live/artists/show_live.ex

```
<.header>
  <.h1>...</.h1>
  <:subtitle :if={@artist.previous_names != []}>
    formerly known as: {Enum.join(@artist.previous_names, ", ")}
```

19. https://hexdocs.pm/ash/dsl-ash-resource.html#validations-validate-before_action?

```
➤   </:subtitle>
    ...
</.header>
```



And now our real Artist pages, complete with their real Album listings, are complete! We've learnt about the tools Ash provides for relating resources together, and how we can work with related data for efficent data loading, preparations, and data integrity. These are core building blocks, that you can use when building out your own applications and we'll be using more of in the future as well.

And we *still* haven't needed to write a lot of code — the small snippets we've written, like validations and changes, have been very targeted and specific, but have been usable throughout the whole app, from seeding data in the database to rendering errors in the UI.

We're only scratching the surface though — in the next chapter, we'll make the Artist catalog really useful, giving users the ability to search, sort, and page through artists, using more of Ash's built-in functionality. We'll also see how we can use calculations and aggregates to perform some sophisticated queries, without even breaking a sweat. This is where things will *really* get interesting!

CHAPTER 3

Creating a Better Search UI

In the previous chapter, we learned how we can link resources together with relationships, and use validations, preparations and changes to implement business logic within resource actions. With this new knowledge, we could build out a fairly comprehensive data model if we wished. We could make resources for everything from tracks on albums, band members, record labels, anything we wanted, and also make a reasonable CRUD interface for it all. We've covered a lot!

But we're definitely missing some business logic, and UI polish. If we had a whole lot of artists in the app, it would become difficult to use. The artist catalog is just one big list of cards — there's no way to search or sort data, and we definitely don't need the *whole* list at all times. Let's look at making this catalog a lot more user-friendly, using query filtering, sorting, and pagination.

Custom Actions with Arguments

To improve discoverability, we'll add search to the Artist catalog, to allow users to look up artists by name. What might it ideally look like, if we were designing the interface for this function? It'd be great to be able to call it like:

```
iex> Tunez.Music.search_artists("fur")
{:ok, [%Tunez.Music.Artist{name: "Valkyrie's Fury"}, ...]}
```

Can we do it? Yes we can!

Designing a search action

A search action will be reading existing data from the database, so we'll add a new `read` action to the Artist resource to perform this new search.

03/lib/tunez/music/artist.ex

```
actions do
```

```
    # ...
➤   read :search do
➤   end
  end
```

When we covered read actions in Defining a read action, on page 15, we mentioned that Ash will read data from the data layer based on parameters we provide, which can be done as part of the action definition. Our search action will support one such parameter, the text to match names on, via an *argument* to the action.

03/lib/tunez/music/artist.ex

```
read :search do
➤   argument :query, :ci_string do
➤     constraints allow_empty?: true
➤     default ""
➤   end
  end
```

Arguments can be anything from scalar values like integers or booleans, to maps, to resource structs. In our case, we'll be accepting a case-insensitive string (or *ci_string*), to allow for case-insensitive searching. This argument can then be used in a *filter*, to add conditions to the query, limiting the records returned to only those that match the condition.

03/lib/tunez/music/artist.ex

```
read :search do
  argument :query, :ci_string do
    # ...
  end

➤   filter expr(contains(name, ^arg(:query)))
  end
```

Whoa! There's a lot of new stuff in single line of code. Let's break it down a bit.

Filters with expressions

Filters are the where-clauses of our queries, allowing us to only fetch the records that match our query. They use a special SQL-like syntax, inspired by Ecto, but are much more expressive.

In iex you can test out different filters by running some queries inline. You'll need to run require Ash.Query first:

```
iex(1)> require Ash.Query
Ash.Query
iex(2)> Ash.Query.filter(Tunez.Music.Album, year_released == 2024)
#Ash.Query<resource: Tunez.Music.Album,
```

```
 filter: #Ash.Filter<year_released == 2024>>
iex(3)> |> Ash.read()
SELECT a0."id", a0."name", a0."inserted_at", a0."updated_at",
a0."year_released", a0."artist_id", a0."cover_image_url" FROM "albums" AS
a0 WHERE (a0."year_released"::bigint::bigint = $1::bigint::bigint) [2024]
{:ok, [%Tunez.Music.Album{year_released: 2024, ...}, ...]}
```

Filters aren't just limited to equality checking — they can use any of the expression syntax[1] including operators and functions. All of the expression syntax listed is data-layer-agnostic, and because we're using AshPostgres, is converted into SQL when we run the query.

Unlike running a filter with Ash.Query.filter, whenever we refer to expressions elsewhere, we need to wrap the body of the filter in a call to expr. The reasons for this are historical — the Ash.Query.filter function predates other usages of expressions, so this will likely be changed in a future version of Ash for consistency — so it's something to keep in mind.

Inside our filter, we'll use the contains/2 expression function, which is a substring checker. It checks to see if the first argument, in our case a reference to the name attribute of our resource, contains the second argument, which is a reference to the query argument to the action!

Because we're using AshPostgres, this filter will use the ilike[2] function in PostgreSQL:

```
iex(4)> Tunez.Music.Artist
Tunez.Music.Artist
iex(5)> |> Ash.Query.for_read(:search, %{query: "co"})
#Ash.Query<
  resource: Tunez.Music.Artist,
  arguments: %{query: #Ash.CiString<"co">},
  filter: #Ash.Filter<contains(name, #Ash.CiString<"co">)>
>
iex(6)> |> Ash.read()
SELECT a0."id", a0."name", a0."biography", a0."previous_names",
a0."inserted_at", a0."updated_at" FROM "artists" AS a0 WHERE
(a0."name"::text ILIKE $1) ["%co%"]
{:ok, [#Tunez.Music.Artist<name: "Crystal Cove", ...>, ...]}
```

Which does exactly what we want — a case-insensitive substring match on the column contents, based on the string we provide.

1. https://hexdocs.pm/ash/expressions.html
2. https://www.postgresql.org/docs/current/functions-matching.html#FUNCTIONS-LIKE

Speeding things up with custom database indexes

Using an ilike query naively over a massive data set isn't exactly performant — it'll run a sequential scan over every record in the table. As more and more artists get added, the search would get slower and slower. To make this query more efficient, we'll add a custom database index called a GIN index[3] on the name column.

AshPostgres supports the creation of custom indexes[4] like a GIN index. To create a GIN index specifically, we first need to enable the PostgreSQL pg_trgm extension.[5] AshPostgres handles enabling and disabling PostgreSQL extensions, via the installed_extensions[6] function in the Tunez.Repo module. By default it only includes ash-functions, so we can add pg_trgm to this list:

03/lib/tunez/repo.ex

```
defmodule Tunez.Repo do
  use AshPostgres.Repo, otp_app: :tunez

  @impl true
  def installed_extensions do
    # Add extensions here, and the migration generator will install them.
➤   ["ash-functions", "pg_trgm"]
  end
```

Then we can add the index to the postgres block of our Artist resource:

03/lib/tunez/music/artist.ex

```
postgres do
  table "artists"
  repo Tunez.Repo

➤   custom_indexes do
➤     index "name gin_trgm_ops", name: "artists_name_gin_index", using: "GIN"
➤   end
end
```

Generally, AshPostgres will generate the names of indexes by itself from the fields, but because we're creating a custom index we have to specify a valid name.

Lastly, generate and run the migration to update the database with the new extension and index.

```
$ mix ash.codegen add_gin_index_for_artist_name_search
$ mix ash.migrate
```

3. https://pganalyze.com/blog/gin-index#indexing-like-searches-with-trigrams-and-gin_trgm_ops
4. https://hexdocs.pm/ash_postgres/dsl-ashpostgres-datalayer.html#postgres-custom_indexes
5. https://www.postgresql.org/docs/current/pgtrgm.html
6. https://hexdocs.pm/ash_postgres/AshPostgres.Repo.html#module-installed-extensions

What kind of performance benefits do we actually get for this? We ran some tests, by inserting a million records with various names into our Artists table using the faker library. Without the index, running the SQL query to search for a word like "snow" (which returns 3,041 results in our data set) takes about 150ms.

After adding the index and re-running the generated SQL query with EXPLAIN ANALYZE,[7] however, the numbers look a *lot* different:

```
Bitmap Heap Scan on artists a0  (cost=118.28..15299.90 rows=10101
width=131) (actual time=1.571..13.443 rows=3041 loops=1)
  Recheck Cond: (name ~~* '%snow%'::text)
  Heap Blocks: exact=2759
  ->  Bitmap Index Scan on artists_name_idx  (cost=0.00..115.76 rows=10101
      width=0) (actual time=1.104..1.105 rows=3041 loops=1)
      Index Cond: (name ~~* '%snow%'::text)
Planning Time: 0.397 ms
Execution Time: 14.691 ms
```

That's a huge saving, the query now only takes about 10% of the time! It might not seem like such a big deal when we're talking about milliseconds, but for *every* artist query being run, it all adds up!

Integrating search into the UI

Now that we have our search action built, we can make the tidy interface we imagined and integrate it into the Artist catalog.

A code interface with arguments

We previously imagined a search function API like:

```
iex> Tunez.Music.search_artists("fur")
{:ok, [%Tunez.Music.Artist{name: "Valkyrie's Fury"}, ...]}
```

This will be a new code interface in our domain, one that supports passing arguments (args) to the action.

03/lib/tunez/music.ex

```
resource Tunez.Music.Artist do
  # ...
  define :search_artists, action: :search, args: [:query]
end
```

Defining a list of arguments with names that match the arguments defined in our function makes that link that we're after — the first parameter passed

7. https://www.postgresql.org/docs/current/using-explain.html#USING-EXPLAIN

when calling the Tunez.Music.search_artists function will now be assigned to the query argument in the action.

You can verify that this link has been made by checking out the function signature in iex:

```
iex(1)> h Tunez.Music.search_artists

          def search_artists(query, params \\ nil, opts \\ nil)

Calls the search action on Tunez.Music.Artist.
```

Any action arguments *not* listed in the args list on the code interface will be placed into the next argument, the map of params. If we didn't specify args: [:query], we would need to call the search function like this:

```
Tunez.Music.search_artists(%{query: "fur"})
```

Which works, but isn't anywhere near as nice!

Searching from the catalog

In the Artist catalog, searches should be repeatable and sharable, and we'll achieve this by making the searched-for text part of the query string, in the page URL. If a user visits a URL like http://localhost:4000/?q=test, Tunez should run a search for the string "test" and show only the matching results.

We currently read the list of artists to display in the handle_params/3 function definition, in Tunez.Artists.IndexLive:

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(_params, _url, socket) do
  artists = Tunez.Music.read_artists!()

  socket =
    socket
    |> assign(:artists, artists)
    # ...
```

Instead, we'll read the q value from the params to the page (from the page route/query string), and call our new search_artists function instead:

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
  query_text = Map.get(params, "q", "")
  artists = Tunez.Music.search_artists!(query_text)

  socket =
    socket
    |> assign(:query_text, query_text)
    |> assign(:artists, artists)
    # ...
```

Of course, users don't search by editing the URL — they search by typing text in a search box. We'll add another action slot to the .header component in the render/3 function of IndexLive, to render a search box function component.

03/lib/tunez_web/live/artists/index_live.ex

```
<.header responsive={false}>
  <.h1>Artists</.h1>
  <:action>
    <.search_box query={@query_text} method="get"
                 data-role="artist-search" phx-submit="search" />
  </:action>
  <:action>
    <.button_link
      # ...
```

When a user types something in the box and presses Enter to submit, the "search" event will be sent to the liveview. The event handler takes the entered text and patches the liveview — updating the URL with the new query string and calling handle_params/3 with the new params, which then re-runs the search and will re-render the catalog.

That's pretty neat! There's another big thing we can add, to make the artist catalog more awesome — the ability to sort the artists on the page. We'll start with some basic sorts, like sorting them alphabetically or by most recently updated, and then later in the chapter we'll look at some really *amazing* ones.

Dynamically Sorting Artists

Our searching functionality is fairly limited — Tunez doesn't have a concept of "best match" when searching text, artists either match or they don't. To help users potentially surface what they want to see more easily, we'll let them sort their search results. Maybe they want to see most recently added

artists listed first? Maybe they want to see artists that have released the most albums listed first? (oops that's a bit of a spoiler!) Let's dig in.

Letting users set a sort method

We'll start from the UI layer — how can users select a sort method? Usually it's by a dropdown of sort options at the top of the page, so we'll drop one next to the search box. In Tunez.Artists.IndexLive, we'll add another action to the actions list in the header function component:

03/lib/tunez_web/live/artists/index_live.ex

```
<.header responsive={false}>
  <.h1>Artists</.h1>
➤ <:action><.sort_changer selected={@sort_by} /></:action>
  <:action>
    # ...
```

The @sort_by assign doesn't yet exist, but it will store a string defining what kind of sort we want to perform. We'll add this to the list of assigns, in handle_params/3:

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
➤ sort_by = nil
  # ...

  socket =
    socket
➤   |> assign(:sort_by, sort_by)
    |> assign(:query_text, query_text)
    # ...
```

The actual sort_changer function component has already been defined further down in the liveview — it reads a set of option tuples for the sort methods we'll support, with internal and display representations, and embeds them into a form, with a phx-change event handler.

When the user selects a sort option, the "change-sort" event will be sent to the liveview. The handle_event/3 function head for this event looks pretty similar to the function head for the "search" event, right below it, except we now have an extra sort_by parameter in the query string. Let's add sort_by to the params list in "search" event handler as well, by reading it from the socket assigns. This will let users search *then* sort, or sort *then* search, and the result will be the same because both parameters will always be part of the URL.

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_event("search", %{"query" => query}, socket) do
➤ params = remove_empty(%{q: query, sort_by: socket.assigns.sort_by})
  {:noreply, push_patch(socket, to: ~p"/?#{params}")}
```

```
end
```

Test it out in your browser — now changing the sort dropdown should navigate to a URL with the sort method in the query string, like http://localhost:4000/?q=the&sort_by=name.

Now that we have the sort method in the query string, we can read it when the page loads just like we read pagination parameters, in handle_params/3. We'll do some validation to make sure that it's a valid option from the list of options, and then store it in the socket like before.

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
  sort_by = Map.get(params, "sort_by") |> validate_sort_by()
  # ...
```

That's the full loop of what we need to implement from a UI perspective — we have a default sort method defined, the user can change the selected value, that value gets reflected back to them in the URL and on the page. Now we can look at how to *use* that value, to change the way the data is returned when our user runs a search.

The base query for a read action

When we run any read action on a resource, we always have to start from some base, onto which we can build a query and start layering extras like filters, loads and so on. We've seen examples of this throughout the book, all the way back to our very first example of how to run a read action:

```
iex(2)> Tunez.Music.Artist
Tunez.Music.Artist
iex(3)> |> Ash.Query.for_read(:read)
#Ash.Query<resource: Tunez.Music.Artist>
iex(4)> |> Ash.Query.sort(name: :asc)
#Ash.Query<resource: Tunez.Music.Artist, sort: [name: :asc]>
iex(5)> |> Ash.Query.limit(1)
#Ash.Query<resource: Tunez.Music.Artist, sort: [name: :asc], limit: 1>
iex(6)> |> Ash.read()
SELECT a0."id", a0."name", a0."biography", a0."inserted_at", a0."updated_at"
FROM "artists" AS a0 ORDER BY a0."name" LIMIT $1 [1]
{:ok, [#Tunez.Music.Artist<...>]}
```

The core thing needed to create a query for a read action is knowing which resource needs to be read. By default this is what Ash does when we call a read action, or a code interface that points to a read action — it uses the resource module itself as the base and builds the query from there.

We *can* change this though. We can pass in our own hand-rolled query when calling a code interface for a read action, or pass a list of options to be used with the resource module when constructing the base query, and these will be used instead.

We mentioned earlier in Loading Related Resource Data, on page 42 that every action can take an optional set of arguments, but it's worth reiterating. These don't have to be defined as arguments to the action, they're added at the end, and they can radically change the behaviour of the action. For code interfaces for read actions, this list of options[8] includes the query option, and that's what we'll use to provide a query in the form of a keyword list.

The query keyword list can include any of the opts that Ash.Query.build/3[9] supports, and in our case we're interested in setting a sort order, so we'll pick sort_input.

Using sort_input for succinct yet expressive sorting

A few of you are probably already wondering, why sort_input, when sort is right there? What's the difference? Both could be used for our purposes, but one is much more useful than the other when sorts come from query string input.

sort is the traditional way of specifying a sort order with field names and sort directions, eg. to order records alphabetically by name, A to Z, you would specify [name: :asc]. To order alphabetically by name and then by newest created record first (to consistently sort artists that have the same name), you would specify [name: :asc, inserted_at: :desc]. Which is fine, this works, you can test it out with iex and our Tunez.Music.search_artists code interface function:

```
iex(6)> Tunez.Music.search_artists("the", [query: [sort: [name: :asc]]])
{:ok,
 [
   #Tunez.Music.Artist<name: "Nights in the Nullarbor", ...>,
   #Tunez.Music.Artist<name: "The Lost Keys", ...>
 ]}
```

sort_input is a bit different — instead of a keyword list, we can specify a single comma-separated string of fields to sort on. Sorting is ascending by default, but can be inverted by prefixing a field name with a -. So our example from before, sorting alphabetically by name and then newest first would be name,-inserted_at. Heaps better!

To use sort_input, we do need to make one change to our resource though — as it's intended to let users specify their own sort methods, it will only permit

8. https://hexdocs.pm/ash/code-interfaces.html#using-the-code-interface
9. https://hexdocs.pm/ash/Ash.Query.html#build/3

sorting on *public* attributes. We don't want users trying to hack our app in any way, after all. All attributes are *private* by default, for the highest level of security, so we'll have to explicitly mark those we want to be publicly accessible. This is done by adding public? true as an option on each of the attributes we want to be sortable:

03/lib/tunez/music/artist.ex

```
attributes do
  # ...

  attribute :name, :string do
    allow_nil? false
    public? true
  end

  # ...

  create_timestamp :inserted_at, public?: true
  update_timestamp :updated_at, public?: true
end
```

Once the attributes are marked public, then sort_input will be usable the way we want:

```
iex(6)> Tunez.Music.search_artists("the", [query: [sort_input: "-name"]])
{:ok,
 [
   #Tunez.Music.Artist<name: "The Lost Keys", ...>,
   #Tunez.Music.Artist<name: "Nights in the Nullarbor", ...>
 ]}
```

Because we've condensed sorting down to specifying a single string value at runtime, it's perfect for adding as an option when we run our search, in the handle_params/3 function in TunezWeb.Artists.IndexLive:

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
  # ...
  artists = Tunez.Music.search_artists!(query_text, query: [sort_input: sort_by])
  # ...
```

It's actually more powerful than our UI needs — it supports sorting on multiple columns, when we only have a single dropdown for one field, but that's okay. There's just one little tweak to make in our sort_options function — when we want recently added or updated records to be shown first, they should be sorted *descending*, so prefix those two field names with a -.

03/lib/tunez_web/live/artists/index_live.ex

```
defp sort_options do
  [
    {"recently updated", "-updated_at"},
```

```
    {"recently added", "-inserted_at"},
    {"name", "name"}
  ]
end
```

We can now search and sort, or sort and search, and everything works just as expected. There's still too much data to display on the page, though. Even if searching through All Of The Artists That Ever Was returns only a few hundred or thousand results, that would take too long to render. We'll split up the results with pagination, and let users browse artists at their own pace.

Pagination of Search Results

Pagination is the best way of limiting the amount of data on an initial page load, to the most important things a user would want to see. If they want more data they can request more data, either by scrolling to the bottom of the page and having more data load automatically (usually called *infinite scroll*), or by clicking a button to load more.

We'll implement the more traditional method of having distinct pages of results, and letting users go backwards/forwards between pages via buttons at the bottom of the catalog.

Adding pagination support to the search action

Our first step in implementing pagination is to update our `search` action to use it. Ash supports automatic pagination of read actions[10] using the `pagination`[11] macro, so we'll add that to our action definition.

03/lib/tunez/music/artist.ex

```
read :search do
  # ...
  pagination offset?: true, default_limit: 12
end
```

Ash supports both `offset` pagination (eg. "show the next 20 records after the 40th record") and `keyset` pagination (eg. "show the next 20 records after record ID=12345"). We've chosen to use offset pagination — it's a little easier to understand, and well-suited for when the data isn't frequently being updated. When the data *is* frequently being updated, such as for news feeds or timelines, or you want to implement infinite scrolling, then keyset pagination would be the better choice.

10. https://hexdocs.pm/ash/read-actions.html#pagination
11. https://hexdocs.pm/ash/dsl-ash-resource.html#actions-read-pagination

Adding the pagination macro immediately changes the return type of the search action. You can see this if you run a sample search in iex:

```
iex(1)> Tunez.Music.search_artists!("cove")
%Ash.Page.Offset{
  results: [#Tunez.Music.Artist<name: "Crystal Cove", ...>],
  limit: 12,
  offset: 0,
  count: nil,
  rerun: {#Ash.Query<...>, [«opts»]},
  more?: false
}
```

The list of artists resulting from running the text search is now wrapped up in an Ash.Page.Offset struct, which contains extra pagination-related information such how many results there are in total (if the countable option is specified), and whether there are more results to display. If we were using keyset pagination, you'd get back an Ash.Page.Keyset struct instead, but the data within would be similar.

This means we'll need to update the liveview, to support the new data structure.

Showing paginated data in the catalog

In TunezWeb.Artists.IndexLive, we load a list of artists and assign them under the artists key in the socket, for the template to iterate over. Now calling search_artists! will return a Page struct, so rename the variable and socket assign to better reflect what is being stored.

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
  # ...
  page = Tunez.Music.search_artists!(query_text, query: [sort_input: sort_by])

  socket =
    socket
    |> assign(:query_text, query_text)
    |> assign(:page, page)
    # ...
```

We also need to update the template code, to use the new @page assign and iterate through the page results.

03/lib/tunez_web/live/artists/index_live.ex

```
<div :if={@page.results == []} class="p-8 text-center">
  <.icon name="hero-face-frown" class="w-32 h-32 bg-gray-300" />
  <br /> No artist data to display!
</div>
```

```
<ul class="gap-6 lg:gap-12 grid grid-cols-2 sm:grid-cols-3 lg:grid-cols-4">
  <li :for={artist <- @page.results}>
    <.artist_card artist={artist} />
  </li>
</ul>
```

This works — now we only have one page worth of artists showing in the catalog, but no way of navigating to other pages.

We'll add some pre-prepared dummy pagination links to the bottom of the artist catalog template, with the pagination_links function component defined in the liveview. As pagination info will also be kept in the URL for easy sharing/reloading, the component will use the query text, the current sort and the page to construct URLs to link to, for changing pages.

03/lib/tunez_web/live/artists/index_live.ex

```
<Layouts.app {assigns}>
  <% # ... %>

  <.pagination_links page={@page} query_text={@query_text} sort_by={@sort_by} />
</Layouts.app>
```

To make the pagination links functional, we'll look at another of AshPhoenix's modules — AshPhoenix.LiveView.[12] It contains a handful of really useful helper functions for inspecting a Page struct to see if there's a previous page, next page, what the current page is, and so on. We can use these to add links to for the next/previous pages in the pagination_links function component, conditionally disabling them if there is no valid page to link to.

03/lib/tunez_web/live/artists/index_live.ex

```
<div
  :if={AshPhoenix.LiveView.prev_page?(@page) ||
       AshPhoenix.LiveView.next_page?(@page)}
  class="flex justify-center pt-8 space-x-4"
>
  <.button_link data-role="previous-page" kind="primary" inverse
    patch={~p"/?#{query_string(@page, @query_text, @sort_by, "prev")}"}
    disabled={!AshPhoenix.LiveView.prev_page?(@page)}
  >
    « Previous
  </.button_link>
  <.button_link data-role="next-page" kind="primary" inverse
    patch={~p"/?#{query_string(@page, @query_text, @sort_by, "next")}"}
    disabled={!AshPhoenix.LiveView.next_page?(@page)}
  >
    Next »
  </.button_link>
```

12. https://hexdocs.pm/ash_phoenix/AshPhoenix.LiveView.html

```
</div>
```

The query_string helper function doesn't yet exist, but we can quickly write it. It will take some pagination info from the @page struct, and use it to generate a keyword list of data to put in the query string:

03/lib/tunez_web/live/artists/index_live.ex

```
def query_string(page, query_text, sort_by, which) do
  case AshPhoenix.LiveView.page_link_params(page, which) do
    :invalid -> []
    list -> list
  end
  |> Keyword.put(:q, query_text)
  |> Keyword.put(:sort_by, sort_by)
  |> remove_empty()
end
```

We're using offset pagination, so when you call AshPhoenix.LiveView.page_link_params/2 it will generate limit and offset parameters.

```
iex(1)> page = Tunez.Music.search_artists!("a")
%Ash.Page.Offset{results: [#Tunez.Music.Artist<...>, ...], ...}
iex(2)> TunezWeb.Artists.IndexLive.query_string(page, "a", "name", "prev")
[sort_by: "name", q: "a"]
iex(3)> TunezWeb.Artists.IndexLive.query_string(page, "a", "-inserted_at", "next")
[sort_by: "-inserted_at", q: "a", limit: 12, offset: 12]
```

When interpolated by Phoenix into a URL on the "Next" button link, it will become http://localhost:4000/?q=a&sort_by=name&limit=12&offset=12.

« Previous Next »

The last step in the process is to use these limit/offset parameters, to make sure we load the right page of data. At the moment even if we click "Next", the URL changes but we still only see the first page of artists. To do that, we'll use another one of the helpers from AshPhoenix.LiveView, to parse the right data out of the params before we load artist data in handle_params/3.

03/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
  # ...
➤ page_params = AshPhoenix.LiveView.page_from_params(params, 12)

  page =
    Tunez.Music.search_artists!(query_text,
➤     page: page_params,
      query: [sort_input: sort_by]
    )
```

We *could* pluck out the limit and offset values from params ourselves, but by doing it this way, if we wanted to change the pagination type — from offset to keyset, or vice versa — we wouldn't have to touch this view code at all. We'd only have to change one line of code, the pagination definition in the search action, and everything else would still work. If you really want to be wild, you can even support *both* types of pagination in the action — URLs that include params for either type will work. Nifty!

And that's it! We've now got full sorting, searching, and pagination for our artist catalog. It was a lot to go through and understand, but not actually a lot of code. Concerns that belong entirely to our UI, like sorting, stayed in the UI layer of the app. Features that are more in-depth, like text searching, came into the resource layer to be analyzed and optimized.

Looking for even more dynamicness?

If you're imagining the majesty of a full "advanced search"-type form, where users can add their own boolean predicates and really narrow down what they're looking for, Ash has support for that by way of AshPhoenix.FilterForm.[13] Implementing one is a little out of the scope of this book, but the documentation should be able to get you started!

Now we'd love to talk about a really killer data modelling feature that Ash provides — calculations!

No DB field? No Problems, with Calculations

Calculations[14] are an awesome way of defining a special type of attribute that isn't stored in your database, but is *calculated* on-demand from other information, like a virtual field. You can use data from related resources, data from files on the filesystem, from external sources, or even just some way of tweaking, deriving or reformatting data you already store for that resource.

Calculations have to be specifically loaded when reading data, the same way you load a relationship; but once they're loaded, they can be treated like any other attribute of a resource.

Calculating data with style

Let's say we wanted to display how many years ago each album was released, on an artist's profile page. That'll make all Tunez's users feel really old! (We

13. https://hexdocs.pm/ash_phoenix/AshPhoenix.FilterForm.html
14. https://hexdocs.pm/ash/calculations.html

won't *actually* do this, because it's a terrible idea, but for demonstration purposes.)

Like a lot of the functionality we've seen before, we can add calculations to a resource by defining a top-level calculations block in the resource.

```
defmodule Tunez.Music.Album do
  # ...

➤  calculations do
➤  end
end
```

Inside the calculations block, we can use the calculate[15] macro to define individual calculations. A calculation needs three things — a name for the resulting attribute, the type of the resulting attribute, and some method of generating the value to store in the attribute.

```
calculations do
➤  calculate :years_ago, :integer, expr(2025 - year_released)
end
```

Calculations use the expression syntax[16] that we saw earlier with filters, to make for really terse code. These are SQL-ish, so we can't use arbitrary Elixir functions in them (hence we're hardcoding 2025 for the year), but we can write some very complex conditions. If we wanted to use some logic that can't easily be written as an expression, such as dynamically using the current year, or converting a string of minutes and seconds to a number of seconds, we could define a separate calculation module. (We'll see this exact example later in Calculating the seconds of a track, on page 201!)

Once you've added a calculation, you can test it out in iex by loading the calculation as part of the data for an album. We've seen load: [:albums] when loading artist data before, and to load nested data, each item in the load list can be a keyword list of nested things to load.

```
iex(1)> Tunez.Music.get_artist_by_id(«uuid», load: [albums: [:years_ago]])
{:ok, #Tunez.Music.Artist<
  albums: [
    #Tunez.Music.Album<year_released: 2022, years_ago: 3, ...>,
    #Tunez.Music.Album<year_released: 2012, years_ago: 13, ...>
  ],
  ...
>}
```

15. https://hexdocs.pm/ash/dsl-ash-resource.html#calculations-calculate
16. https://hexdocs.pm/ash/expressions.html

You could then use this years_ago attribute when rendering a view, or in an API response, like any other attribute. And because they *are* just like any other attribute, you can even use them within other calculations:

```
calculations do
  calculate :years_ago, :integer, expr(2025 - year_released)
  calculate :string_years_ago,
            :string,
            expr("wow, this was released " <> years_ago <> " years ago!")
end
```

If you load the string_years_ago calculation, you don't need to specify that it depends on another calculation so that should be loaded too — Ash can work that out for you.

```
iex(1)> Tunez.Music.get_artist_by_id(«uuid», load: [albums: [:string_years_ago]])
{:ok, #Tunez.Music.Artist<
  albums: [
    #Tunez.Music.Album<
      year_released: 2022,
      string_years_ago: "wow, this was released 3 years ago!",
      years_ago: #Ash.NotLoaded<:calculation, field: :years_ago>,
      ...
    >,
    #Tunez.Music.Album<
      year_released: 2012,
      string_years_ago: "wow, this was released 13 years ago!",
      years_ago: #Ash.NotLoaded<:calculation, field: :years_ago>,
      ...
    >
  ],
  ...
>}
```

You only get back what you request!

One important thing to note here is that Ash will only return the calculations you requested, even if some extra calculations are evaluated as a side-effect.

In the above example, Ash will calculate the years_ago field for each artist record, because it's needed to calculate string_years_ago — but years_ago *won't* be returned as part of the Artist data. This is to avoid accidentally relying on these implicit side-effects. If we changed how string_years_ago is calculated to not use years_ago, it would break any usage of years_ago in our views!

Calculations are a really, really powerful tool. They can be used for simple data formatting like our string_years_ago example, or complex tasks like building

tree data structures out of a flat data set, or pathfinding in a graph. Calculations can also work with resource relationships and their data, and here we get to what we actually want to build for Tunez.

Calculations with related records

Tunez is recording all this interesting album information for each artist, but not *showing* any of it in the artist catalog. So we'll use calculations to surface some of it as part of the loaded Artist data, and display it on the page.

There are three pieces of information we really want:

- The number of albums each artist has released,
- The year that each artist's latest album was released in, and
- The most recent album cover for each artist

Let's look at how we can build each of those with calculations!

Counting albums for an artist

Ash provides the `count/2` expression function,[17] also known as an inline *aggregate* function (we'll see why shortly), that we can use to count records in a relationship.

So to count each artist's albums as part of a calculation, we could add it as a calculation in the Artist resource:

```
defmodule Tunez.Music.Artist do
  # ...

  calculations do
➤   calculate :album_count, :integer, expr(count(albums))
  end
end
```

Testing this in `iex`, you can see it makes a pretty efficient query, even when querying multiple records. There's no n+1 query issues here, it's all handled in one query through a clever join:

```
iex(1)> Tunez.Music.search_artists("a", load: [:album_count])
SELECT a0."id", a0."name", a0."inserted_at", a0."updated_at", a0."biography",
a0."previous_names", coalesce(s1."aggregate_0", $1::bigint)::bigint::bigint
FROM "artists" AS a0 LEFT OUTER JOIN LATERAL (SELECT sa0."artist_id" AS
"artist_id", coalesce(count(*), $2::bigint)::bigint AS "aggregate_0" FROM
"public"."albums" AS sa0 WHERE (a0."id" = sa0."artist_id") GROUP BY
sa0."artist_id") AS s1 ON TRUE WHERE (a0."name"::text ILIKE $3) ORDER BY
a0."id" LIMIT $4 [0, 0, "%a%", 13]
{:ok, %Ash.Page.Offset{...}}
```

17. https://hexdocs.pm/ash/expressions.html#inline-aggregates

It's a little bit icky with some extra type-casting that doesn't need to be done, but we'll address that shortly (this isn't the final form of our calculation!)

Finding the most recent album release year for an artist

We're working with relationship data again, so we'll use another inline aggregate function. Because we've ensured that albums are always ordered according to release year in Loading Related Resource Data, on page 42, the first album in the list of related albums will always be the most recent.

The first aggregate function is used to fetch a specific attribute value from the first record in the relationship, so we can use it to pull out just the year_released value from the album, and store it in a new attribute on the Artist.

```
calculations do
  calculate :album_count, :integer, expr(count(albums))
➤   calculate :latest_album_year_released, :integer,
➤     expr(first(albums, field: :year_released))
end
```

Finding the most recent album cover for an artist

This is a slight twist on the previous calculation. Again we want the most recent album, but only out of albums that have the optional cover_image_url attribute specified. We *could* add this extra condition using the filter option on the base query, like we did when we set a sort order in The base query for a read action, on page 67, but we don't actually need to — for convenience, Ash will filter out nil values automatically. Note that the calculation can still return nil if an artist has no albums at all, or has no albums with album covers.

Everything combined, our calculation can look like this:

```
calculations do
  calculate :album_count, :integer, expr(count(albums))
  calculate :latest_album_year_released, :integer,
    expr(first(albums, field: :year_released))
➤   calculate :cover_image_url, :string,
➤     expr(first(albums, field: :cover_image_url))
end
```

If you don't want this convenience, if you really do want the cover for the most recent album even if it's nil, you can add the include_nil?: true option to the first inline-aggregate function call.

And this works! We can specify any or all of these three calculation names, :album_count, :latest_album_year_released, and :cover_image_url, when loading artist

data and get the attributes properly calculated and returned, with only a single SQL query. This is really powerful; and we've only scratched the surface of what you can do with calculations.

Our three calculations have one thing in common — they all use *inline aggregate* functions, to surface some attribute or derived value from relationships. Instead of defining the aggregates inline, we can look at extracting them into full aggregates, and see how that cleans up the code.

Relationship Calculations as Aggregates

Aggregates are a specialized type of calculation, as we've seen above. All aggregates are calculations; but a calculation like `years_ago` in our Album example was not an aggregate.

Aggregates perform some kind of calculation on records in a relationship — it could be a simple calculation like `first` or `count`, or a more complicated calculation like `min` or `avg` (average), or you can even provide a fully-custom implementation if the full list of aggregate types[18] doesn't have what you need.

To start adding aggregates to our Artist resource, we first need to add the `aggregates` block at the top level of the resource. (You might be sensing a pattern about this, by now.)

03/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  aggregates do
  end
end
```

Each of the three inline-aggregate calculations we defined can be rewritten to be an aggregate within this block. An aggregate needs at least three things — the type of aggregate, the name of the attribute to be used for the result value, and the relationship to be used for the aggregate.

So our example of the `album_count` calculation, could be written more appropriately as a `count` aggregate:

03/lib/tunez/music/artist.ex

```
aggregates do
  # calculate :album_count, :integer, expr(count(albums))
➤ count :album_count, :albums
end
```

18. https://hexdocs.pm/ash/aggregates.html#aggregate-types

We don't need to specify the type of the resulting attribute — Ash knows that a count is always an integer, it can't be anything else, even if it's zero. This also simplifies the generated SQL a little bit, there's no need for repeatedly casting things as bigints.

Our latest_album_year_released calculation can be rewritten similarly:

03/lib/tunez/music/artist.ex

```
aggregates do
  count :album_count, :albums

  # calculate :latest_album_year_released, :integer,
  #   expr(first(albums, field: :year_released))

➤ first :latest_album_year_released, :albums, :year_released
end
```

We've dropped a little bit of the messy syntax, and the result is a lot easier to read. We don't need to define that latest_album_year_released is an integer — that can be inferred because the Album resource already defines the year_released attribute as an integer. If the syntax seems a bit mysterious, the options available for each type of aggregate are fully laid out in the Ash.Resource DSL documentation.[19]

The final calculation, for cover_image_url, is the same as for latest_album_year_released. The include_nil?: true option can be used here too, if you really want a cover that might be nil, but we'll rely on the default value of false. If a given artist has *any* awesome album covers, we want the most recent one.

03/lib/tunez/music/artist.ex

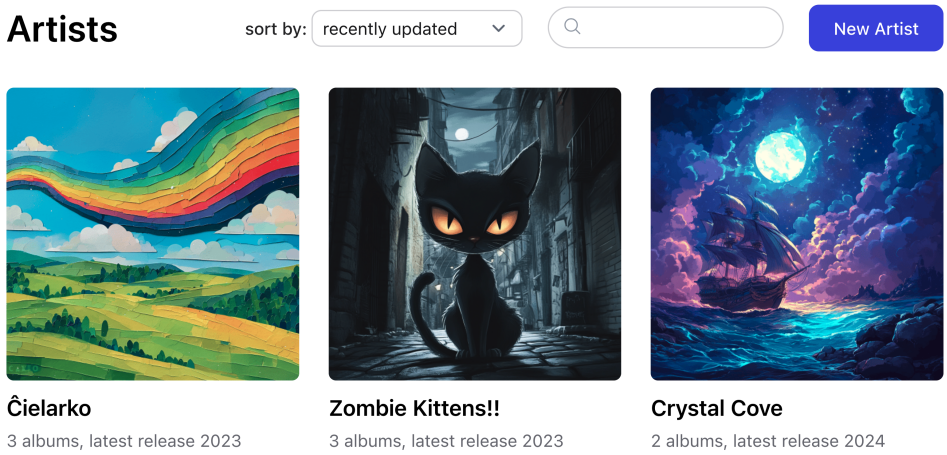
```
aggregates do
  count :album_count, :albums
  first :latest_album_year_released, :albums, :year_released

  # calculate :cover_image_url, :string,
  #   expr(first(albums, field: :cover_image_url))

➤ first :cover_image_url, :albums, :cover_image_url
end
```

In this way, we can put all the logic of how to calculate a latest_album_year_released or a cover_image_url for an artist where it belongs, in the domain layer of our application, and our front-end views don't have to worry about where it might come from. On that note, let's integrate these aggregates in our artist catalog.

19. https://hexdocs.pm/ash/dsl-ash-resource.html#aggregates

Using aggregates like any other attribute

It would be amazing if the artist catalog looked like a beautiful display, of album artwork and artist information.



In Tunez.Artists.IndexLive, the cover image display is handled by a call to the cover_image function component within artist_card. Because we can use and reference aggregate attributes like any other attributes on a resource, we'll add an image argument to the cover_image function component, to replace the default placeholder image with our cover_image_url calculation:

03/lib/tunez_web/live/artists/index_live.ex

```
<div id={"artist-#{@artist.id}"} data-role="artist-card" class="relative mb-2">
  <.link navigate={~p"/artists/#{@artist.id}"}>
➤   <.cover_image image={@artist.cover_image_url} />
  </.link>
</div>
```

Refreshing the artist catalog after making the change might not be what you expect — why aren't the covers displaying? Because we aren't loading them! Remember that we need to specifically load calculations/aggregates if we want to use them, they won't be generated automatically.

We *could* add the calculations to our Tunez.Music.search_artists function call using the load option, like we loaded albums for an artist on their profile page:

```
page =
  Tunez.Music.search_artists!(query_text,
    page: page_params,
    query: [sort_input: sort_by],
➤   load: [:album_count, :latest_album_year_released, :cover_image_url]
  )
```

And this works! This would be the easiest way. But if you ever wanted to reuse this artist card display, you would need to manually include all of the calculations when loading data there too, which isn't ideal. There are a few other ways we could load the data, such as via a preparation[20] in the Artist search action itself:

```
read :search do
  # ...

  prepare build(load: [:album_count, :latest_album_year_released,
    :cover_image_url])
end
```

Implementing it this way would mean that *every time* you call the action, the calculations would be loaded, even if they're not used. If the calculations were expensive, such as loading data from an external service, this would be costly!

Ultimately it depends on the needs of your application, but in this specific case a good middle ground would be to add the load statement to the *code interface*, using the default_options[21] option. This means that whenever we call the action via our Tunez.Music.search_artists code interface, the data will be loaded automatically, but if we call the action manually (such as by constructing a query for the action), it won't.

03/lib/tunez/music.ex

```
define :search_artists,
       action: :search,
       args: [:query],
       default_options: [
         load: [:album_count, :latest_album_year_released, :cover_image_url]
       ]
```

Reloading the artist catalog will now populate the data for all of the aggregates we listed, and look at the awesome artwork appear! Special thanks to Midjourney, for bringing our imagination to life!

Note that some of our sample bands don't have any albums, and some of the artists with albums, don't have any album covers. Our aggregates can account for these cases, and the site isn't broken in any way — we see the placeholder images that we saw before.

20. https://hexdocs.pm/ash/Ash.Resource.Preparation.Builtins.html#build/1
21. https://hexdocs.pm/ash/dsl-ash-domain.html#resources-resource-define-default_options

For the album count and latest album year released fields, we'll add those details to the end of the artist_card function, using the previously-unused artist_card_album_info component defined right below it:

03/lib/tunez_web/live/artists/index_live.ex
```
def artist_card(assigns) do
  ~H"""
  <% # ... %>

  <.artist_card_album_info artist={@artist} />
  """
end
```

And behold! The artist catalog is now in its full glory!

Earlier in the chapter, we looked at sorting artists in the catalog, via three different attributes - name, inserted_at and updated_at. We've explicitly said a few times now, that calculations and aggregates can be treated like any other attribute — does that mean we might be able to sort on them too???

You bet you can!

Sorting based on aggregate data

Around this point is where Ash really starts to shine, and you might start feeling a bit of a tingle with the power at your fingertips. Hold that thought, because it's going to get even better. Let's add some new sort options for our aggregate attributes, to our list of available sort options in Tunez.Artists.IndexLive:

03/lib/tunez_web/live/artists/index_live.ex
```
defp sort_options do
  [
    {"recently updated", "-updated_at"},
    {"recently added", "-inserted_at"},
    {"name", "name"},
➤   {"number of albums", "-album_count"},
➤   {"latest album release", "--latest_album_year_released"}
  ]
end
```

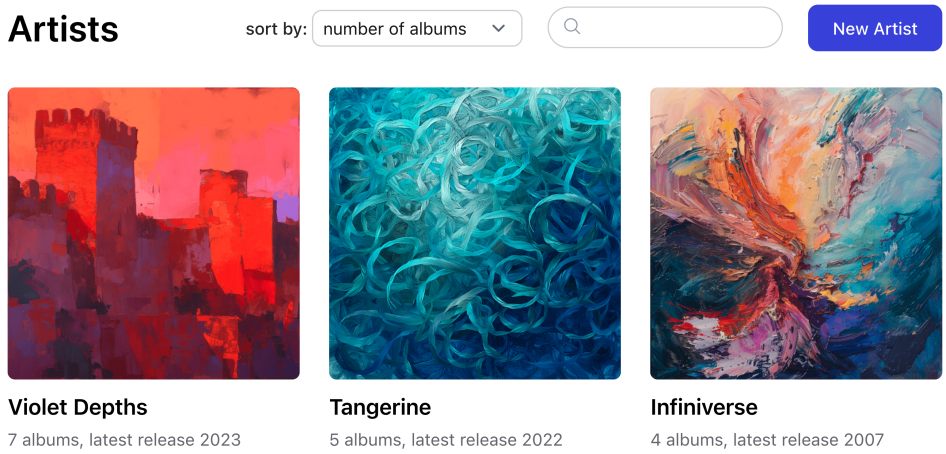
We want artists with the most albums and with the most recent albums listed first, so we'll sort them descending by prefixing the attribute name with a -. Using -- is a bit special — it'll put any nil values (if an artist hasn't released any albums!) at the end of the list.

To allow the aggregates to be sorted on, we do need to mark them as public? true, like we did with our initial set of sortable attributes in Using sort_input for succinct yet expressive sorting, on page 68:

03/lib/tunez/music/artist.ex

```
aggregates do
  count :album_count, :albums do
➤   public? true
  end

  first :latest_album_year_released, :albums, :year_released do
➤   public? true
  end

  # ...
end
```

And then we'll be able to sort in our artist catalog, to see which artists have the most albums, or have released albums most recently:



This is all *amazing*! We've built some really excellent functionality over the course of this chapter, to let users search, sort, and paginate through data.

And in the next one, we'll see how we can use the power of Ash to build some neat APIs for Tunez, using our existing resources and actions. Reduce, re-use and recycle code!

CHAPTER 4

Generating APIs Without Writing Code

In the previous chapter, we looked at making the artist catalog a lot more user-friendly by letting users search, sort and filter artist data. This was a big boost to Tunez's popularity, so much so that some users are asking how they can use the data from Tunez in their own apps.

We can give users access to a Tunez *application programming interface*, or *API* — a way of letting their apps talk to Tunez, to fetch or modify data. APIs are everywhere, whenever we build apps that can communicate with other apps, we're doing it via an API. If you're connecting to Facebook to read a user's friends list, or built an app that upload photos to a image-hosting service like Cloudinary, you're using those services's APIs.

Let's look at how we can build an API for Tunez to let other apps talk to *us*, using the resources and actions we've defined so far.

Model Your Domain, Derive The Rest

One of the core design principles[1] of Ash is its *declarative* and *derivable* nature. By themselves, resources are static configuration files that Ash can interpret and generate code from. We've seen examples of this with code interfaces for our actions — we *declared* that we should have an interface for our Artist :search action that accepts one argument for the query text, and Ash generated the function for us to call.

This can be taken further — Ash can generate a lot more than functions. It can generate entire APIs around the existing resources and actions in your app, hence the name of this chapter.

1. https://hexdocs.pm/ash/design-principles.html

It sounds wild, but what *is* an API, really? A set of functions, that map from an input URL to a response in a format like JSON.[2] Our web UI is an API, albeit a heavily customized one that returns HTML. An API using something like GraphQL[3] or REST[4] is a lot more standardized. Both the incoming requests and the outgoing responses have a strict format to adhere to, and that can be generated for us using Ash.

We'll build two APIs in this chapter, using both REST and GraphQL. In a real app you'd probably want one or the other, but we'll show off a little bit here and add both. Let's go!

Building a JSON REST Interface

A REST (or RESTful) API can be generated by Ash using the ash_json_api package. This will accept requests over HTTP, and return data formatted as JSON. APIs generated with ash_json_api are compatible with the JSON:API[5] specification, and can also generate OpenAPI[6] schemas, opening up a whole world of supporting tooling options.

Setup

You can add ash_json_api to Tunez using the igniter.install Mix task:

```
$ mix igniter.install ash_json_api
```

This will add a few new pieces to your app, that have a *lot* of power. The additions include:

- The ash_json_api Hex package, as well as its sibling dependency open_api_spex (in mix.exs and mix.lock)
- Code formatting and configuration to support a new application/vnd.api+json media type, needed for JSON:API compatibility[7] (in config/config.exs)
- A new TunezWeb.AshJsonApiRouter module that uses AshJsonApi.Router. This will process the web requests and return responses in the correct format (in lib/tunez_web/ash_json_api_router.ex)
- And a new scope in your Phoenix router to accept web requests for the /api/json/ URL (in lib/tunez_web/router.ex)

2. https://stackoverflow.blog/2022/06/02/a-beginners-guide-to-json-the-data-format-for-the-internet/
3. https://graphql.org/
4. https://www.ibm.com/topics/rest-apis
5. https://jsonapi.org/
6. https://www.openapis.org/
7. https://jsonapi.org/format/#introduction

This takes care of a lot of the boilerplate around a REST API, leaving us to handle the implementation of what our API should actually *do*.

Adding Artists to the API

What we primarily want to expose in our API is the CRUD interface for our resources, to let users manage artist and album data over the API. Each of our resources can be exposed as a type/schema/definition and each action on a resource exposed as an operation.

By default, the API is empty — we have to manually include each resource and action we want to make public. To add a resource to the API, we can use Ash's `ash.extend` Mix task to *extend* the resource with the `AshJsonApi.Resource` extension:

```
$ mix ash.extend Tunez.Music.Artist json_api
```

This will make some handy changes in our app:

- `AshJsonApi.Resource` will be added as an extension to the `Tunez.Music.Artist` resource
- A default API "type" will be added to the resource, in a new `json_api` block in the resource. Each record in a API response is identified by an `id` and a `type` field,[8] the type usually being a string version of the resource name.

And because this is the first resource in the `Tunez.Music` domain to be configured for AshJsonApi, the patch generator will also connect pieces in the domain:

- `AshJsonApi.Domain` will be added as an extension to the `Tunez.Music` domain
- And the `Tunez.Music` domain will be added to the list of domains configured in the `TunezWeb.AshJsonApiRouter` module.

You could make all the changes yourself manually, but there's a few moving parts there and it can be easy to miss a connection. The generators are a convenient way of making sure everything is set up as it should be.

Next, to make the actions on the Artist resource available in the API, we need to set up routes for them. Like code interfaces, this can be done either on the resource or the domain. However, to keep the domain as the solid boundary between our domain model and the outside world, we'll add them on the domain.

8. https://jsonapi.org/format/#document-resource-object-identification

In a new top-level json_api block in the Tunez.Music domain module, configure the routes using the DSL provided by AshJsonApi:[9]

04/lib/tunez/music.ex

```
defmodule Tunez.Music do
  # ...

  json_api do
    routes do
      base_route "/artists", Tunez.Music.Artist do
        get :read
        index :search
        post :create
        patch :update
        delete :destroy
      end
    end
  end
end
```

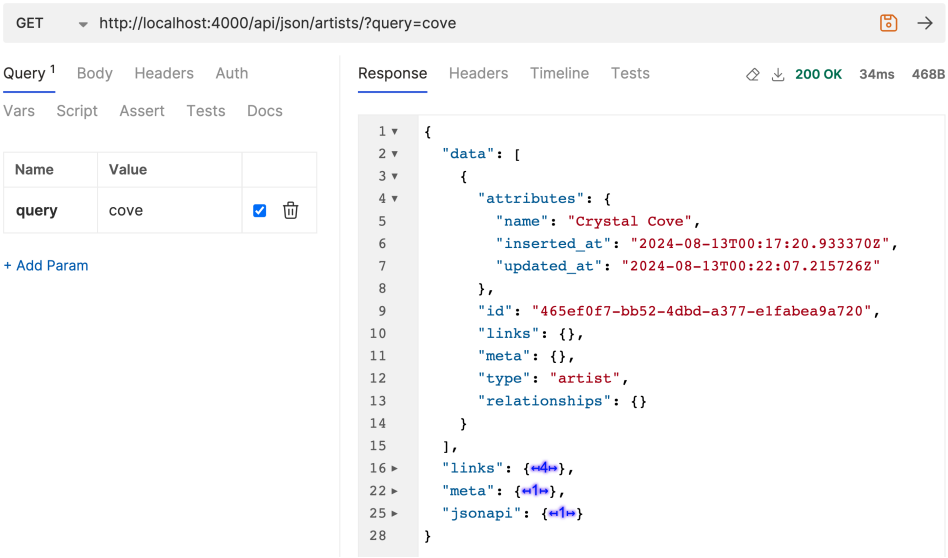
This code will connect a GET request to read a single artist by a given ID to the read action of the Tunez.Music.Artist resource, automatically applying the correct filter. A POST request will be connected to the create action, and so on.

You can see these generated routes included when running the phx.routes Mix task, to list all of the routes available in your application:

```
$ mix phx.routes
  *       /api/json/swaggerui       OpenApiSpex.Plug.SwaggerUI [...]
  GET     /api/json/artists/:id     ... Tunez.Music.Artist.read
  GET     /api/json/artists         ... Tunez.Music.Artist.search
  POST    /api/json/artists         ... Tunez.Music.Artist.create
  PATCH   /api/json/artists/:id     ... Tunez.Music.Artist.update
  DELETE  /api/json/artists/:id     ... Tunez.Music.Artist.destroy
  GET     /                         TunezWeb.Artists.IndexLive nil
  «the rest of the routes defined in the Phoenix router»
```

So how can we actually *use* the API? For GET requests, you can access the endpoints provided in a browser like any other URL. Alternatively, you could use a dedicated API client app such as Bruno,[10] shown here making a GET request to /api/json/artists:

9. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-domain.html#json_api-routes
10. https://www.usebruno.com/



Don't forget the custom headers!

While not strictly required for `GET` requests, you should configure your API client to add the correct `Content-Type` and `Accept` headers when making any requests to your API. The value for both headers should be `application/vnd.api+json`.

The `/api/json` section of the URL matches the scope that our AshJsonApi router is mounted in, in the Phoenix router, and `/artists` matches the base route for the `Tunez.Music.Artist` resource, meaning this request will connect to the `search` action of the resource.

The action accepts a `query` argument that can be passed in as a query string parameter, and the search results are returned in a neat JSON format. Links for pagination are automatically included, because the action supports pagination. And we barely needed to lift a finger!

What data gets included in API responses?

You might notice that some attributes are missing in the API response — the artist `name` is shown, but the `biography` and `previous_names` are missing, as are the aggregates for `album_count`, `cover_image_url` and `latest_album_year_released` that we added in the last chapter.

This is because only *public attributes* (attributes that are specifically marked `public?: true`) are returned in API queries, by default. This is for security reasons — if all attributes were included by default, it would be really easy to acciden-

tally leak information as you add more data to your resources, if you didn't also explicitly *remove* them from your API.

Some of the attributes are already public, such as those we used for sorting in the previous chapter. To add biography and previous_names to the API response, you can also mark them as public?: true in the Tunez.Music.Artist resource:

04/lib/tunez/music/artist.ex

```
attributes do
  # ...
  attribute :biography, :string do
➤   public? true
  end

  attribute :previous_names, {:array, :string} do
    default []
➤   public? true
  end
  # ...
end
```

Aggregates are a little different. For this usage, they are *not* treated like every other attribute, and aren't included by default even if they're public. This is because calculations and aggregates can be computationally expensive, and if they aren't specifically needed by users of the API, you can save time and effort by not calculating and returning them.

There are still two ways that you can make calculations and aggregates visible in your API:

- If you *do* want them to be calculated and returned by default, you can use the default_fields config option, eg. default_fields [:id, :name, :biography, :album_count]. This can be set at the resource level,[11] to apply any time an instance of the resource is returned in a response, or for any specific API route (either in the domain[12] or in a resource.)[13] This will replace the default "return all public attributes" behaviour, though, so you'll have to list *all* fields that should be returned by default, including any public attributes.
- Alternatively, part of the JSON:API spec[14] states that users can request which specific fields they want to fetch as part of their API request. Our API is JSON:API-compliant, so users can add the fields query string

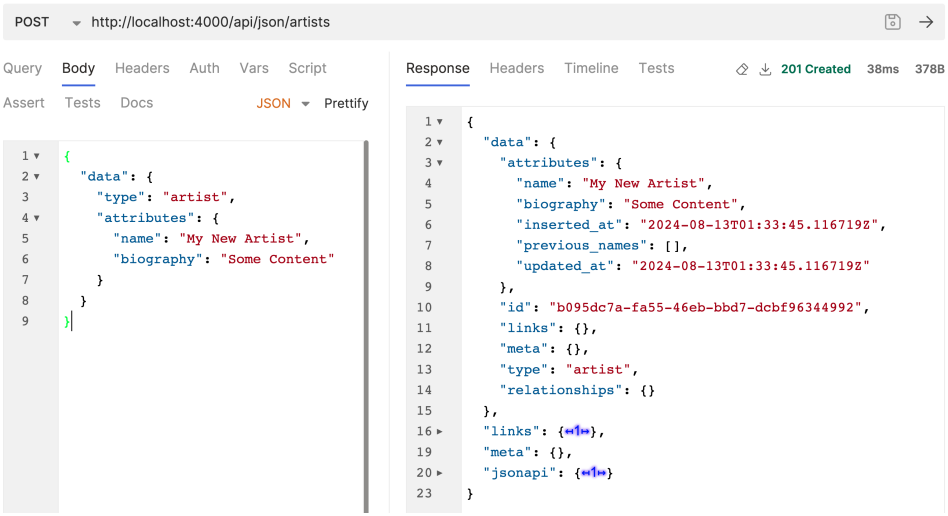
11. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-resource.html#json_api-default_fields
12. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-domain.html#json_api-routes-base_route-get-default_fields
13. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-resource.html#json_api-routes-get-default_fields
14. https://jsonapi.org/format/#fetching-sparse-fieldsets

parameter and list only the fields they need in a comma-separated list. The fields can be any public fields, including aggregates and calculations, so a URL like http://localhost:4000/api/json/artists?fields=name,album_count would return only names and the number of albums for each artist in the search results.

Creating artist records

We won't cover *every* endpoint we created, but it's worth a quick look at how data can be created, as well as how it can be read.

As our introspection showed earlier, we can make POST requests to the same URL we used for searching, to access the create action of our resource. We can post a JSON object in the format specified in the JSON:API specification,[15] containing the content for the artist record to be created.



In your Phoenix server logs, you can see the create request being handled by the AshJsonApiRouter module and processed:

```
[info] POST /api/json/artists
[debug] Processing with TunezWeb.AshJsonApiRouter
  Parameters: %{"data" => %{"attributes" => %{"biography" => "Some
  Content", "name" => "My New Artist"}, "type" => "artist"}}
  Pipelines: [:api]
INSERT INTO "artists" ("id","name","biography","inserted_at",
"previous_names","updated_at") VALUES ($1,$2,$3,$4,$5,$6) RETURNING
"updated_at","inserted_at","previous_names","biography","name","id"
[«uuid», "My New Artist", "Some Content", «timestamp», [], «timestamp»]
```

15. https://jsonapi.org/format/#crud-creating

Because this API endpoint connects to the create action in our Tunez.Music.Artist resource, it accepts all of the same data as the action does. Posting additional data (e.g., an attribute that the action doesn't accept or a non-existent attribute) or invalid data (e.g., a missing required field) will return an error message, and the record won't be created.

Other requests can be made in a similar way — a PATCH request to update an existing artist, and a DELETE request to delete an artist record.

Adding Albums to the API

We can add album management to the JSON API in much the same way we added artists, by extending the Tunez.Music.Album resource:

```
$ mix ash.extend Tunez.Music.Album json_api
```

And adding our routes in the json_api block in the domain:

04/lib/tunez/music.ex

```
json_api do
  routes do
    # ...

    base_route "/albums", Tunez.Music.Album do
      post :create
      patch :update
      delete :destroy
    end
  end
end
```

This closely resembles the web UI. We don't have an API endpoint to list all albums, but we do have endpoints to manage individual album records. This will create URLs like /api/json/albums and /api/json/albums/:album_id, with various HTTP methods to connect to the different actions in the resource.

Because we're not in the web UI, though, we don't have the nice pre-filled hidden artist ID when submitting a HTTP request to create an album. We need to provide a valid artist ID as part of the attributes of the album to be created, like this:

```
{
  "data": {
    "type": "album",
    "attributes": {
      "name": "New Album",
      "artist_id": [a-valid-uuid],
      "year_released": 2022
    }
  }
```

```
}
```

As part of this, we can also mark some of the attributes on the Tunez.Music.Album resource as public?: true, such as name, year_released, and cover_image_url, so they can be returned in API responses.

04/lib/tunez/music/album.ex

```
attributes do
  uuid_primary_key :id

  attribute :name, :string do
    allow_nil? false
➤   public? true
  end

  attribute :year_released, :integer do
    allow_nil? false
➤   public? true
  end

  attribute :cover_image_url, :string do
➤   public? true
  end

  # ...
```

There's just one part we're missing now — listing an artist's albums.

Showing albums for a given artist

The JSON:API spec allows for two methods of fetching related resources[16] for a given resource. We'll cover both methods; you can choose the one that suits you when building your own APIs.

Both methods require the relationship to be *public* to be accessible over the API, so you'll need to mark it as public?: true in Tunez.Music.Artist:

04/lib/tunez/music/artist.ex

```
relationships do
  has_many :albums, Tunez.Music.Album do
    sort year_released: :desc
➤   public? true
  end
end
```

Including related records

This is the easiest way to provide relationship data, and it mirrors what we see in the web UI when we view an Artist profile page. You can allow related records to be *included* when fetching the parent resource — returning the

16. https://jsonapi.org/format/#fetching-relationships

artist record, and their albums, in one request. This is convenient for consumers of the API as they only need to make a single request, but the responses can be large and they may overfetch data.

To enable this in our API, edit the json_api block in the Tunez.Music.Artist resource to list which relationships can be included from this resource:

04/lib/tunez/music/artist.ex

```
json_api do
  type "artist"
  includes [:albums]
end
```

You'll then be able to fetch album data by adding include=albums to the query string of any request for artist data, such as http://localhost:4000/api/json/artists?query=cove&include=albums. The response will have a list of record identifiers under the relationships key of the fetched data, and then a separate list of the full records under the top-level included key. The format is a little quirky, but it's the JSON:API way!

Linking to a list of related records

For a different approach, you can return a link to fetch relationship data via a separate related[17] route in your domain, and also specify which action should be used when fetching the related data.

04/lib/tunez/music.ex

```
base_route "/artists", Tunez.Music.Artist do
  # ...
  related :albums, :read, primary?: true
end
```

This will add a related relationship link to any artist API response, like http://localhost:4000/api/json/artists/[id]/albums. Accessing this related URL will then provide a list of the related albums in JSON format, which is great!

We've now got all the same functionality from our web UI, accessible over a JSON API. As we make further changes to our actions to add more functionality, they'll all automatically flow through to our API endpoints as well.

Now how can we get the word out about Tunez's fantastic new API?

We can auto-generate some documentation that we can share publicly to show people how to integrate with the API really easily!

17. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-domain.html#json_api-routes-base_route-related

Generating API documentation with OpenApiSpex

When we installed AshJsonApi, it also added a package called open_api_spex[18] to our mix.exs file, and this is how we can generate OpenAPI specifications automatically for our API. We don't have to do anything to set it up — the AshJsonApi installer did so when it created the JSON API router in lib/tunez_web/ash_json_api_router.ex:

04/lib/tunez_web/ash_json_api_router.ex

```
defmodule TunezWeb.AshJsonApiRouter do
  use AshJsonApi.Router,
    domains: [Tunez.Music],
➤   open_api: "/open_api"
end
```

This one line of code will give you a full OpenAPI specification document at the provided route, http://localhost:4000/api/json/open_api by default.

We can use this specification document with any tool or library that works with OpenAPI (and there are many!)[19] One that OpenApiSpex provides support for out of the box is Swagger UI,[20] to generate full documentation for our API and even let users try out endpoints directly from the docs.

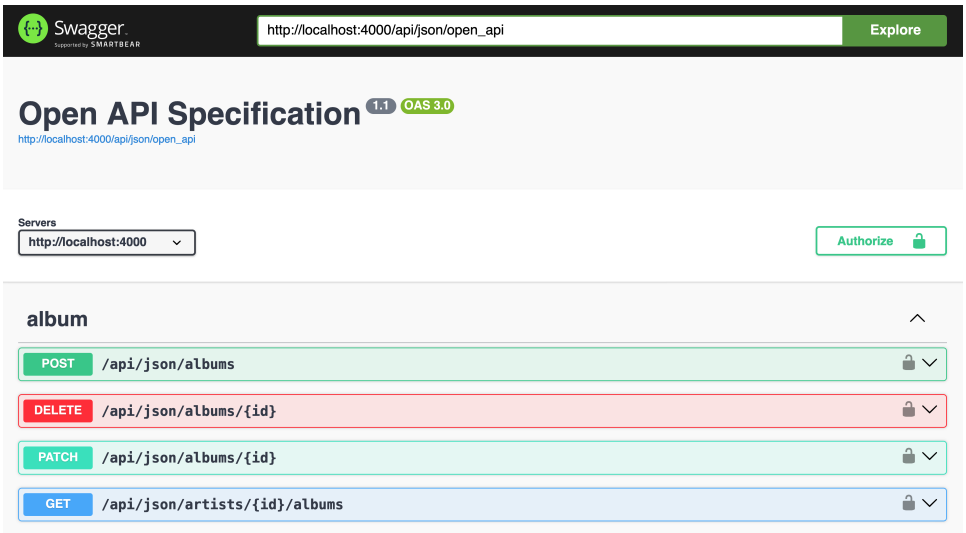
OpenApiSpex's SwaggerUI plug has already been set up in our router, in lib/tunez_web/router.ex:

04/lib/tunez_web/router.ex

```
scope "/api/json" do
  pipe_through [:api]

➤   forward "/swaggerui",
➤           OpenApiSpex.Plug.SwaggerUI,
➤           path: "/api/json/open_api",
➤           default_model_expand_depth: 4

  forward "/", TunezWeb.AshJsonApiRouter
end
```

This sets up the /api/json/swaggerui URL, with a full set of Swagger UI API documentation:

18. https://hexdocs.pm/open_api_spex/
19. https://tools.openapis.org/
20. https://swagger.io/tools/swagger-ui/



Totally for free! And it'll stay up to date as you update your API, adding or updating resources or actions.

If Swagger UI isn't to your liking, Redoc[21] is a good alternative. It can be installed in your app via the redoc_ui_plug[22] Hex package, and configured in your Phoenix router in a very similar way to Swagger UI.

And if you decide you don't want the documentation after all, you only need to remove the SwaggerUI plug from your router.

Customizing the generated API

Now that we have a great overview of our API, and we can see it the way a user would, we can see some places that it can be improved. These certainly aren't the *only* ways, but they're low-hanging fruit that will give quick wins.

Adding informative descriptions

Some of the defaults in the generated content can be a bit lacking. Our API shouldn't be called "Open API Specification", and AshJsonApi doesn't know what we really *mean* when we say "Get artists", so the default description of the search API endpoint is "/artists operation on artist resource". Not great.

Ash allows us to add description metadata in a few different places, that will be picked up by the OpenAPI schema generator and added to the documentation. This includes:

21. https://github.com/Redocly/redoc
22. https://hexdocs.pm/redoc_ui_plug/index.html

- A description for a resource as a whole. This can be added as part of a top-level resource block, such as this in Tunez.Music.Artist:

```
defmodule Tunez.Music.Artist do
  use Ash.Resource, ...

  resource do
    description "A person or group of people that makes and releases music."
  end
```

- A description for an action, or argument for an action, in a resource. These can be added in the action declaration itself, such as:

```
read :search do
  description "List Artists, optionally filtering by name."

  argument :query, :ci_string do
    description "Return only artists with names including the given value."
    # ...
```

As a nice bonus, these descriptions should be picked up by any Elixir-related language server packages in your text editor, such as ElixirLS or elixir-tools in VSCode.

Updating the API title and version

Basic information that is OpenAPI-specific, such as the name of the API, can be customized via options to use AshJsonApi.Router in your JSON API router module. If you need to make more specific changes, you can also add a modify_open_api hook function,[23] to be called when generating the OpenAPI spec. This function will have access to the whole generated spec, and there are a lot of things[24] that can be changed or overwritten, so be careful!

04/lib/tunez_web/ash_json_api_router.ex

```
defmodule TunezWeb.AshJsonApiRouter do
  use AshJsonApi.Router,
    domains: [Tunez.Music],
    open_api: "/open_api",
➤   open_api_title: "Tunez API Documentation",
➤   open_api_version: to_string(Application.spec(:tunez, :vsn))
end
```

Once you've made any changes like descriptions, refreshing the Swagger UI docs will immediately reflect the changes, and they're looking a lot better now.

23. https://hexdocs.pm/ash_json_api/open-api.html#customize-values-in-the-openapi-documentation
24. https://hexdocs.pm/open_api_spex/OpenApiSpex.OpenApi.html#t:t/0

Removing unwanted extras

Looking through the docs carefully shows that our API can actually do a little bit more than we thought. Expanding the section for GET /api/json/artists, our artist search, shows the endpoint will allow data to be filtered via a filter parameter in the URL. This is pretty cool, but we already have our own specific filtering set up, to search artists by name. So while it sounds like a waste, we'll disable the generated filtering for parity with the web interface.

AshJsonApi provides both generated filtering *and* sorting of data, for any index actions in our API router. These can be disabled either at the resource level or per-action. For Tunez, we want to keep the generated sorting of artists because we allow that via the web, but disable the generated filtering. We can do that with the derive_filter? config option[25] in the Tunez.Music.Artist resource:

04/lib/tunez/music/artist.ex

```
json_api do
  type "artist"
  includes [:albums]
  derive_filter? false
end
```

And that's our JSON REST API, fully complete! It packs a lot of punch, for not a lot of code. We didn't have to write any endpoints, generate any JSON, worry about error handling — everything is handled by AshJsonApi, which generates API endpoints and controllers to connect the actions in our resources to the outside world. It's pretty nifty.

If JSON and REST aren't to your liking, maybe you're in the GraphQL camp. We can build a GraphQL API for Tunez in a very similar way!

Building a GraphQL Interface

A GraphQL API can be generated by Ash using the ash_graphql package. It's built on top of the excellent absinthe[26] library, so it's rock-solid and ready for production use. This will create a standard GraphQL endpoint, accepting GET requests over HTTP using GraphQL syntax and returning JSON responses.

GraphQL APIs are a little more flexible than REST APIs — though with the JSON:API specification, the gap is smaller than you might think. We won't debate the pros and cons of each type of API here, but both approaches can create well-defined, well-structured, and well-documented interfaces for your users to work with.

25. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-resource.html#json_api-derive_filter?
26. https://hexdocs.pm/absinthe/

Setup

You can add ash_graphql to Tunez using the igniter.install Mix task:

```
$ mix igniter.install ash_graphql
```

This will add a few new pieces to your app, that have a *lot* of power. The changes include:

- Code formatting and configuration for AshGraphql and Absinthe (in .formatter.exs and config/config.exs)
- A new graphql pipeline and scope in your Phoenix router, to accept requests for both /gql, the GraphQL endpoint, and /gql/playground, a GraphiQL API client (in lib/tunez_web/router.ex)
- A new TunezWeb.GraphqlSchema module, that uses AshGraphql and Absinthe.Schema and is seeded with a sample runnable query (in lib/tunez_web/graphql_schema.ex)
- A new TunezWeb.GraphqlSocket module connected in TunezWeb.Endpoint, to support GraphQL subscriptions (in lib/tunez_web/graqhql_socket.ex and lib/tunez_web/endpoint.ex)

This takes care of all of the boilerplate around setting up a GraphQL API. After restarting your Phoenix server, you can test out the GraphiQL playground by accessing http://localhost:4000/gql/playground — there isn't a lot to see there at the moment, but we do have a generated schema with the sample query that AshGraphql provides when no other queries are present. Hello, AshGraphql!

Now we can look at what the API needs to actually *do*.

Adding Artists to the API

What we primarily want to expose in our GraphQL API is the CRUD interface for our resources, to let users manage artist and album data over the API. Each of our resources can be exposed as a type, and each action on a resource exposed as either a query or a mutation.

By default, the API is empty — we have to manually include each resource and action that we want to make public. To add a resource to the API, we can use Ash's ash.extend Mix task to *extend* the resource with the AshGraphql.Resource extension:

```
$ mix ash.extend Tunez.Music.Artist graphql
```

This will make some handy changes to our app:

- AshGraphql.Resource will be added as an extension to the Tunez.Music.Artist resource

- A default GraphQL type will be added to the resource, in a new graphql block in the resource. This is usually a simplified atom version of the resource name.

And because this is the first resource in the Tunez.Music domain to be configured for AshGraphql, the patch generator will also connect pieces in the domain:

- AshGraphql.Domain will be added as an extension to the Tunez.Music domain
- And the Tunez.Music domain will be added to the list of domains configured in the TunezWeb.GraphqlSchema module.

You could make all the changes yourself manually, but there's a few moving parts there and it can be easy to miss a connection. The generators are a convenient way of making sure everything is set up as it should be.

Next, to make the actions on the Artist resource available in the API, we need to create queries and mutations for them. Like code interfaces, this can be done either on the resource or the domain, but to keep the domain as the solid boundary between our domain model and the outside world, we'll add them on the domain.

In the top-level graphql block defined in the Tunez.Music domain model, we can add queries for read actions of our Artist resource. AshGraphql provides macros like get and list[27] for this, which describe what kind of responses we expect from the queries.

04/lib/tunez/music.ex

```
defmodule Tunez.Music do
  # ...

  graphql do
    queries do
      get Tunez.Music.Artist, :get_artist_by_id, :read
      list Tunez.Music.Artist, :search_artists, :search
    end
  end
end
```

This will create GraphQL queries named getArtistById, connecting to the read action of the Tunez.Music.Artist resource and automatically applying an ID filter; and searchArtists connecting to the search action.

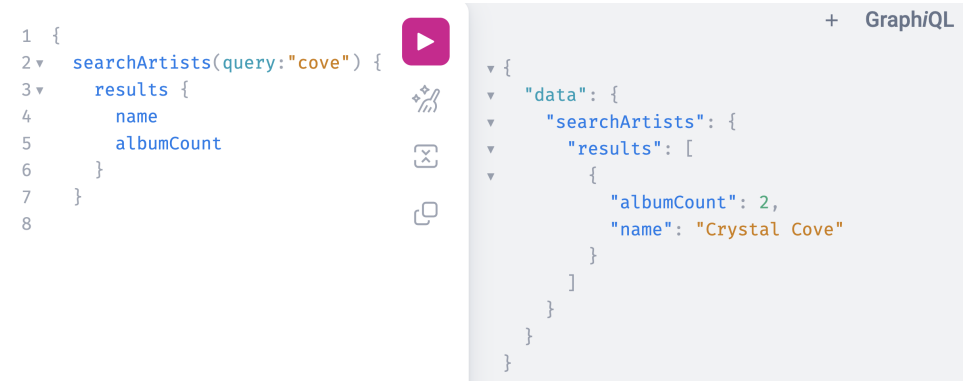
We can do the same for the non-read actions in our resource, which will all be mutations in the API:

27. https://hexdocs.pm/ash_graphql/dsl-ashgraphql-domain.html#graphql-queries

04/lib/tunez/music.ex

```
graphql do
  # ...

  mutations do
    create Tunez.Music.Artist, :create_artist, :create
    update Tunez.Music.Artist, :update_artist, :update
    destroy Tunez.Music.Artist, :destroy_artist, :destroy
  end
end
```

This gives us a lot out of the box. In the GraphiQL playground, expanding the Docs on the left now shows the queries and mutations we just defined, and they're fully-typed — a getArtistById query will return an Artist type, with all public attributes of the resource also typed and available to be requested. We can run any query, and fetch data in the shape we want.



The search action accepts a query argument, which means that the generated searchArtists query also accepts a query argument. Because the action also supports pagination, the request and response both support pagination, and it's all right there in the generated types. We barely needed to lift a finger!

What data gets included in API responses?

If you skipped over the JSON API section because GraphQL is much more interesting, you might be surprised to see that fields like biography and previousNames aren't defined in the GraphQL Artist type.

Only *public attributes* (attributes that are specifically marked public?: true) can be requested and returned in GraphQL responses, for security reasons — if all attributes were included by default, it would be really easy to accidentally leak information as you add more data to your resources, if you didn't also explicitly *remove* them from your API.

By adding public? true to those attributes in the Tunez.Music.Artist resource:

04/lib/tunez/music/artist.ex

```
attributes do
  # ...
  attribute :biography, :string do
    public? true
  end

  attribute :previous_names, {:array, :string} do
    default []
    public? true
  end
  # ...
end
```

They'll then be added to the GraphQL Artist type and can be requested like any other field. Aggregates and calculations must also be marked as public? true if you want to make them accessible in the API.

Creating artist records

We won't cover every operation we created, but it's worth a quick look at how data can be created, as well as how it can be read.

Expanding the Schema tab in the playground shows that we can call a mutation named createArtist for creating new Artist records. Because it connects to the create action in the Tunez.Music.Artist resource, the attributes the action accepts is matched by the typing of the input to the mutation.

```
mutation {
  createArtist(input:{
    name:"Unleash the Rangers",
    biography:"A great Canadian band"
  }) {
    errors { fields message }
    result { name albumCount }
  }
}
```

GraphiQL

```
{
  "data": {
    "createArtist": {
      "errors": [],
      "result": {
        "albumCount": 0,
        "name": "Unleash the Rangers"
      }
    }
  }
}
```

In your Phoenix server logs, you can see the mutation being handled by Absinthe using the TunezWeb.Schema module, and processed:

```
[debug] ABSINTHE schema=TunezWeb.Schema variables=%{}
---
mutation {
  createArtist(input: {
    name: "Unleash the Rangers",
    biography: "A great Canadian band"
  }) {
```

```
    errors { fields message }
    result { name albumCount }
  }
}
---
INSERT INTO "artists" ("id","name","biography","inserted_at","previous_names",
"updated_at") VALUES ($1,$2,$3,$4,$5,$6) RETURNING "updated_at","inserted_at",
"previous_names","biography","name","id" [«uuid», "Unleash the Rangers", "A
great Canadian band", «timestamp», [], «timestamp»]
```

If the submitted data passes the input type checking, but fails resource validation (such as an empty name value), the mutation will return a nicely typed error message and the record won't be created. And because the action and mutation will return the record being created, on success, we can use all the usual GraphQL ideas of requesting only the fields we need in the response.

Adding Albums to the API

We can add album management to the GraphQL API in much the same way we added artists, by extending the Tunez.Music.Album resource:

```
$ mix ash.extend Tunez.Music.Album graphql
```

And adding our mutations to the graphql block in the domain:

04/lib/tunez/music.ex

```
graphql do
  mutations do
    # ...
➤   create Tunez.Music.Album, :create_album, :create
➤   update Tunez.Music.Album, :update_album, :update
➤   destroy Tunez.Music.Album, :destroy_album, :destroy
  end
end
```

This closely resembles the web UI. We don't have a query to list all albums, but we do have mutations like createAlbum and updateAlbum to manage individual album records. Mutations for existing records have their arguments split into id, for the ID of the artist/album to be updated, and input for the data to update the record with.

Because we're not in the web UI, though, we don't have the nice pre-filled hidden artist ID when submitting a HTTP request to create an album — we need to provide a valid one with the attributes of the album to be created, like this:

```
mutation {
  createAlbum(input: { name: "New Album Name",
                       artistId: [an-artist-uuid],
```

```
                    yearReleased: 2022
                  }) {
    result { id }
  }
}
```

There's just one part we're missing — listing an artist's albums.

Showing albums for a given artist

If you've followed the JSON API section of this chapter, you may have already made the changes necessary to get this working.

By adding two resources with an existing relationship to our API, the flexible nature of GraphQL means that we'll automatically be able to load related records — *as long as the relationship is public*. We can do this by adding the option `public? true` to the relationship, in the `Tunez.Music.Artist` resource:

04/lib/tunez/music/artist.ex

```
relationships do
  has_many :albums, Tunez.Music.Album do
    sort year_released: :desc
➤   public? true
  end
end
```

This will add the `albums` field to the `Artist` type in the GraphQL API, letting you load related albums anywhere an artist is loaded. Super nifty!

Note that privacy settings on relationships are one way — to be able to load a related artist for an album in the API, you would also need to make the `artist` relationship in the `Tunez.Music.Album` resource public.

Customizing the generated API

With introspection in the GraphiQL playground, we now have a great overview of our API and we can see it the way a user might. We can also see that there are a few places it can be improved upon — a lot of it was covered in Customizing the generated API, on page 96 for the JSON API, so we won't reiterate it, but it applies equally for GraphQL.

Removing unwanted extras

Looking through the schema carefully, shows that our API can actually do a little bit more than we thought. Expanding the section for the `searchArtists` query shows that it also accepts arguments named `filter`, for filtering data, and `sort` for sorting data. The `sort` option will let us sort on any public attribute, either ascending or descending. And the `filter` will let us write complex conditions

using greaterThan, lessThanOrEqualTo, ilike, notEq comparisons and so on, and then combine them with and, or and not clauses. And AshGraphql will generate this for *any* list action in our API, for free.

Our searchArtists query already has a query argument to filter by name, so we don't want that to be used in the filter too. Some fields also don't make much sense to filter on, like biography. To customize the list of fields that can be filtered on, use the filterable_fields config option[28] in the graphql block, in the Tunez.Music.Artist resource:

04/lib/tunez/music/artist.ex

```
graphql do
  type :artist
➤ filterable_fields [:album_count, :cover_image_url, :inserted_at,
➤   :latest_album_year_released, :updated_at]
end
```

We've removed the values that don't make sense, but we'd still allow users to search by name and also apply filters like artists that haven't released an album since 2010 ({ latestAlbumYearReleased: { lessThan: 2010 } }) or artists that were added to Tunez in the last week ({ insertedAt: { greaterThan: [timestamp] } }).

You can also disable the automatic filtering or sorting entirely, with the derive_filter? and derive_sort? options in your resource — set them to false.

And that's the our GraphQL API, fully done up to match the functionality of our web UI. It's pretty powerful, given how little code we needed to write to support it. We didn't have to define our own GraphQL resolvers, or types, or worry about error handling — everything is handled by AshGraphql. Awesome!

We'll be revisiting our two APIs over the rest of this book, as we add more functionality to Tunez — we want to keep full feature parity with the web, and we also want to see if growing the API organically over time will be difficult to do. In the meantime, we'll look at something a bit different.

Some bad actors have started polluting Tunez with bad data, oh no! This won't do! Tunez has to be the best, most accurate source of high-quality information — and that means locking down certain types of access to only people that we trust to not do anything dodgy. Before we can start limiting access, though, we need to know who people are, and that means some kind of authentication process. Onward march!

28. https://hexdocs.pm/ash_graphql/dsl-ashgraphql-resource.html#graphql-filterable_fields

CHAPTER 5

Authentication: Who Are You?

In chapter 4, we expanded Tunez with APIs — we now have HTML in the browser, REST JSON, and GraphQL. It was fun seeing how Ash's declarative nature could be used to generate everything for us, using the existing domains, resources and actions in our app.

But now it's time to get down to serious business. The world is a scary place, and unfortunately we can't trust everyone in it to have free rein over the data in Tunez. We need to start locking down access to critical functionality, to only trusted users — but we don't yet have any way of knowing who those users *are*.

We can solve this by adding authentication to our app, and requiring users to log in before they can create or modify any data. Ash has a library that can help with this, called…

Introducing AshAuthentication

There are two parts to AshAuthentication — the core `ash_authentication` package, and the `ash_authentication_phoenix` Phoenix extension to provide things like signup and registration forms. We'll start with the basic library, to get a feel for how it works, and then add the web layer afterwards.

This chapter will be a little different than everything we've covered so far, because we really won't have to write much code until the later stages. The AshAuthentication installer will generate most of the necessary code into our app for us, and while we won't have to modify a lot of it, it's important to understand it. (And it's there if we *do* need to modify it!)

Install AshAuthentication with Igniter:

```
$ mix igniter.install ash_authentication
```

This will generate a *lot* of code in several stages — so let's break it down bit by bit.

You may get an error here, about SAT solver installation. Ash requires a SAT solver[1] to run authorization policies — by default it will attempt to install picosat_elixir on non-Windows machines, but this can be fiddly to set up. If you get an error, follow the prompts to uninstall picosat_elixir, and install simple_sat instead.

New domain, who's this?

We're now working with a whole different section of our domain model. Previously we were building music-related resources, so we created a domain named Tunez.Music. Authentication is part of a separate system, an account management system, and so the generator will create a new domain called Tunez.Accounts. This domain will be populated with two new resources — Tunez.Accounts.User and Tunez.Accounts.Token.

The Tunez.Accounts.User resource, in lib/tunez/accounts/user.ex, is what will represent, well, *users* of your app. It comes pre-configured with AshPostgres as its data layer, so each user record will be stored in a row of the users database table.

By itself, the user resource doesn't do much yet. It doesn't even have any attributes, except an id. It does have some authentication-related configuration, in the top-level authentication block, like linking the resource with *tokens*. This is what makes up most of the rest of the generated code.

Tokens and secrets and config, oh my!

Tokens, via the Tunez.Accounts.Token resource and the surrounding config, are the secret sauce to an AshAuthentication installation. Tokens are how we securely identify users — from an authentication token provided on every request ("I am logged in as rebecca"), to password reset tokens appended to links in emails, and more.

This is the part you *really* don't want to get wrong when building a web app, because the consequences could be pretty bad. If tokens are insecure, they could be spoofed by malicious users to impersonate other users and gain access to things they shouldn't. So AshAuthentication generates all of the token-related code we need right up front, before we do anything. For basic uses, we shouldn't need to touch anything in the generated token code, but it's there if we need to.

1. https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/

So how do we actually use all this code? We need to set up at least one authentication *strategy*.

Setting Up Password Authentication

AshAuthentication supports a number of authentication strategies[2] — ways we can identify users in our app. Traditionally, we think of logging in to an app via entering an email address and password, which is one of the supported strategies (the password strategy), but there are several more. We can authenticate via different types of OAuth, or even via magic links sent to a user's email address.

Let's set the password strategy up and get a feel for how it works. AshAuthentication comes with igniters to add strategies to our existing app, so you can run the following command:

```
$ mix ash_authentication.add_strategy password
```

This will add a lot *more* code to our app. We now have:

- Two new attributes for the Tunez.Accounts.User resource — email and hashed_password. The email attribute is also marked as an identity, so it must be unique.
- A strategies block added to the authentication configuration in the Tunez.Accounts.User resource. This lists the email attribute as the identity field for the strategy, and it also sets up the resettable option to allow users to reset their passwords.
- The confirmation add-on added to the add_ons block, as part of the authentication configuration in the Tunez.Accounts.User resource. This will require users to confirm their email addresses by clicking on a link in their email, when registering for an account or changing their email address.
- A whole set of actions in our Tunez.Accounts.User resource, around signing in, registering, and resetting passwords.
- And lastly, some email modules that will be used when it comes to actually sending password reset/email confirmation emails.

That's a lot of goodies!

Because the tasks have created a few new migrations, run ash.migrate to get our database up to date:

```
$ mix ash.migrate
```

2. https://hexdocs.pm/ash_authentication/get-started.html#choose-your-strategies-and-add-ons

There will be a few warnings from the email modules about the routes for password reset/email confirmation not existing yet — that's okay, we haven't looked at setting up AshAuthenticationPhoenix yet! But we can still test out our code with the new password strategy in an iex session, to see how it works.

Don't try this in a real app!

Note that we'll skip AshAuthentication's built-in authorization policies for this testing, by passing the authorize?: false option to Ash.create. This is only for testing purposes — the real code in our app won't do this.

Testing authentication actions in iex

One of the generated actions in the Tunez.Accounts.User resource is a create :register_with_password action, which takes email, password, and password_confirmation arguments and creates a user record in the database. It doesn't have a code interface defined, but you can still run it by generating a changeset for the action, and submitting it.

```
iex(1)> Tunez.Accounts.User
Tunez.Accounts.User
iex(2)> |> Ash.Changeset.for_create(:register_with_password, %{email: «email»,
           password: "supersecret", password_confirmation: "supersecret"})
#Ash.Changeset<
  domain: Tunez.Accounts,
  action_type: :create,
  action: :register_with_password,
  ...
>
iex(3)> |> Ash.create!(authorize?: false)
INSERT INTO "users" ("id","email","hashed_password") VALUES ($1,$2,$3)
RETURNING "hashed_password","email","id","confirmed_at" [«uuid»,
#Ash.CiString<«email»>, «hashed password»]
«several queries to generate tokens»
#Tunez.Accounts.User<
  __meta__: #Ecto.Schema.Metadata<:loaded, "users">,
  confirmed_at: nil,
  id: «uuid»,
  email: #Ash.CiString<«email»>,
  ...
>
```

Calling this action has done a few things:

- Inserted the new user record into the database, including securely hashing the provided password;

- Created tokens for the user to authenticate and also confirm their email address, and
- Generated an email to send to the user, to actually confirm their email address. In development, it *won't* send a real email, but all of the plumbing is in place for the app to do so.

What can we do with our new user record? We can try to authenticate them, using the created sign_in_with_password action. This mimics what a user would do on a login form, by entering their email address and password:

```
iex(9)> Tunez.Accounts.User
Tunez.Accounts.User
iex(10)> |> Ash.Query.for_read(:sign_in_with_password, %{email: «email»,
...(10)> password: "supersecret"})
#Ash.Query<
  resource: Tunez.Accounts.User,
  arguments: %{password: "**redacted**", email: #Ash.CiString<«email»>},
  filter: #Ash.Filter<email == #Ash.CiString<«email»>>
>
iex(11)> |> Ash.read(authorize?: false)
SELECT u0."id", u0."confirmed_at", u0."email", u0."hashed_password" FROM
"users" AS u0 WHERE (u0."email"::citext = ($1::citext)) [«email»]
{:ok, [#Tunez.Accounts.User<...>]}
```

And it works! AshAuthentication has validated that the credentials are correct, by fetching any user records with the provided email, hashing the provided password, and verifying that it matches what is stored in the database. You can try it with different credentials like an invalid password; AshAuthentication will properly return an error.

Calling sign_in_with_password with correct credentials has also generated an authentication token in the returned user's metadata, to be stored in the browser and used to authenticate the user in future.

```
iex(12)> {:ok, [user]} = v()
{:ok, [#Tunez.Accounts.User<...>]}
iex(13)> user.__metadata__.token
"eyJhbGciOi..."
```

This token is a JSON Web Token, or JWT.[3] It's cryptographically signed by our app to prevent tampering — if a malicious user has a token and edits it to attempt to impersonate another user, the token will no longer verify. To test out the verification, we can use some of the built-in AshAuthentication functions like AshAuthentication.Jwt.verify/2 and AshAuthentication.subject_to_user/2:

```
iex(14)> AshAuthentication.Jwt.verify(user.__metadata__.token, :tunez)
```

3. https://jwt.io/

```
{:ok,
 %{
   "aud" => "~> 4.5",
   "exp" => 1742123290,
   "iat" => 1740913690,
   "iss" => "AshAuthentication v4.7.6",
   "jti" => «string»,
   "nbf" => 1740913690,
   "purpose" => "user",
   "sub" => "user?id=«uuid»"
 }, Tunez.Accounts.User}
```

The interesting parts of the decoded token here are the `sub` (subject) and the `purpose`. JWTs can be created for all kinds of purposes, and this one is for user authentication, hence the purpose "user". The subject is a specially-formatted string with a user ID in it, which we can verify belongs to a real user:

```
iex(15)> {:ok, claims, resource} = v()
{:ok, %{...}, Tunez.Accounts.User}
iex(16)> AshAuthentication.subject_to_user(claims["sub"], resource)
SELECT u0."id", u0."confirmed_at", u0."hashed_password", u0."email" FROM
"users" AS u0 WHERE (u0."id"::uuid::uuid = $1::uuid::uuid) [«uuid»]
{:ok, #Tunez.Accounts.User<email: #Ash.CiString<«your email»>, ...>}
```

So when a user logs in, they'll receive an authentication token. On subsequent requests, the user can provide this token as part of a header or a cookie, which our app will decode and verify — and hey presto, we now know who they are. They're logged in!

We don't really need to muck around with all of this, though. It's good to know how AshAuthentication works, and how to verify that it works, but we're building a web app — we want forms that users can fill out to register or sign-in. For that, we'll use AshAuthentication's sister library, AshAuthentication-Phoenix.

Automatic UIs With AshAuthenticationPhoenix

As the name suggests, AshAuthenticationPhoenix is a library that connects AshAuthentication with Phoenix, providing a great LiveView-powered UI that we can tweak a little bit to fit our site look and feel, but otherwise don't need to touch. Like other libraries, install it with Igniter:

```
$ mix igniter.install ash_authentication_phoenix
```

Ignoring the same warnings about some routes not existing (this will be the last time we see them!), the AshAuthenticationPhoenix installer will set up:

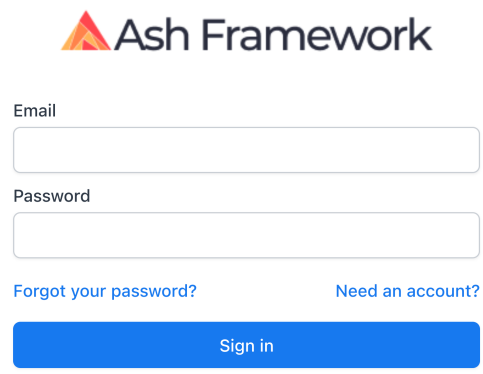
- A basic Igniter config file in .igniter.exs — this is the first generator we've run that needs specific configuration (for Igniter.Extensions.Phoenix), so it gets written to a file
- A TunezWeb.AuthOverrides module, that we can use to customize the look and feel of the generated liveviews (in lib/tunez_web/auth_overrides.ex)
- A TunezWeb.AuthController module, to securely process sign-in requests (in lib/tunez_web/controllers/auth_controller.ex). This is due to a bit of a quirk in how LiveView works; it doesn't have access to the user session to store data on successful authentication.
- A TunezWeb.LiveUserAuth module providing a set of hooks we can use in liveviews (in lib/tunez_web/live_user_auth.ex)
- And lastly, updating our web app router in lib/tunez_web/router.ex to add plugs and routes for all of our authentication-related functionality.

Before we can test it out, there's one manual change we need to make, as Igniter doesn't (yet) know how to patch JavaScript or CSS files. AshAuthenticationPhoenix's liveviews are styled with Tailwind CSS, so we need to add its liveview paths to Tailwind's content lookup paths. Tunez is using Tailwind 4, configured via CSS, so you need to add the @source line under the list of other @source lines in assets/css/app.css.

05/assets/css/app.css

```
/* ... */
@source "../../lib/tunez_web";
@source "../../deps/ash_authentication_phoenix";

@plugin "@tailwindcss/forms";
/* ... */
```

Restart your mix phx.server and then we can see what kind of UI we get, by visiting the sign-in page at http://localhost:4000/sign-in:



It's pretty good! Out of the box we can sign-in, register for new accounts, and request password resets.

After signing in, we get redirected back to the Tunez homepage — but there's no indication that we're now logged in, and no link to log out. We'll fix that now.

Showing the currently-authenticated user

It's a common pattern for web apps to show current user information in the top-right corner of the page, so that's what we'll implement as well. The main Tunez navigation is part of the TunezWeb.Layouts.app function component, in lib/tunez_web/components/layouts.ex, so we can edit to add a new rendered user_info component:

05/lib/tunez_web/components/layouts.ex

```
<div class="flex items-center w-full p-4 pb-2 border-b-2 border-primary-600">
  <div class="flex-1 mr-4">
    <% # ... %>
  </div>
➤ <.user_info current_user={@current_user} socket={@socket} />
</div>
```

This is an existing function component located in the same TunezWeb.Layouts module, and shows sign in/register buttons if there's no user logged in, and a dropdown of user-related things if there is. Refreshing the app after making this change shows a big error, however:

```
key :current_user not found in: %{
  socket: #Phoenix.LiveView.Socket<...>,
  __changed__: %{...},
  page_title: "Artists",
  inner_content: %Phoenix.LiveView.Rendered{...},
  ...
```

Fixing this will require looking into how the new router code works.

Digging into AshAuthenticationPhoenix's generated router code

We didn't really go over the changes to our router in lib/tunez_web/router.ex, after installing AshAuthenticationPhoenix — we just assumed everything was all good. For the most part it is, but there are one or two things we need to tweak.

The igniter added plugs to our pipelines to load the current user — load_from_bearer for our API pipelines, and load_from_session for our browser pipeline. These are what will decode the user's JWT token, load the authenticated user's record, and store it in the request conn for us to use. This works for

traditional non-liveview controller-based web requests, that receive a request and send the response in the same process.

LiveView works differently, though. When a new request is made to a liveview, it spawns a new process and keeps that active websocket connection open for that realtime data transfer. This new process doesn't have access to the session, so although our base request knows who the user is, the spawned process does not.

Enter live_session, and how its wrapped by AshAuthentication, ash_authentication_live_session. This macro will ensure that when new processes are spawned, they get copies of the data in the session so the app will continue working as expected.

What does this mean for Tunez? It means that all our liveview routes that are expected to have access to the current user, need to be moved into the ash_authentication_live_session block in the router.

05/lib/tunez_web/router.ex

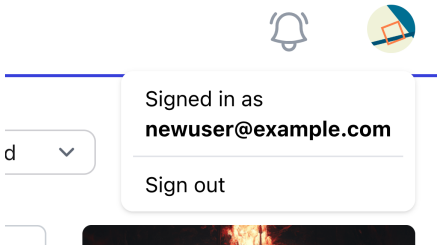
```
scope "/", TunezWeb do
  pipe_through :browser

  # This is the block of routes to move
  live "/", Artists.IndexLive
  # ...
  live "/albums/:id/edit", Albums.FormLive, :edit

  auth_routes AuthController, Tunez.Accounts.User, path: "/auth"
  # ...
```

The ash_authentication_live_session helper is in a separate scope block in the router, earlier on in the file:

05/lib/tunez_web/router.ex

```
scope "/", TunezWeb do
  pipe_through :browser

  ash_authentication_live_session :authenticated_routes do
    # This is the location that the block of routes should be moved to
    live "/", Artists.IndexLive
    # ...
    live "/albums/:id/edit", Albums.FormLive, :edit
  end
end
```

With this change, our app should be renderable, and we should see information about the currently logged in user in the top-right corner of the main navigation.



Now we can turn our attention to the generated liveviews themselves. We want them to look totally seamless in our app, like we wrote and styled them ourselves. While we don't have control over the HTML that gets generated, we can customize a lot of the styling and some of the content, using *overrides*.

Stylin' and profilin' with overrides

Each liveview component in AshAuthenticationPhoenix's generated views has a set of overrides configured, that we can use to change things like component class names and image URLs.

When we installed AshAuthenticationPhoenix, a base TunezWeb.AuthOverrides module was created in lib/tunez_web/auth_overrides.ex. This shows the syntax that we can use to set different attributes, that will then be used when the liveview is rendered:

05/lib/tunez_web/auth_overrides.ex

```
# override AshAuthentication.Phoenix.Components.Banner do
#   set :image_url, "https://media.giphy.com/media/g7GKcSzwQfugw/giphy.gif"
#   set :text_class, "bg-red-500"
# end
```

As well a link to the complete list of overrides[4] you can use, in the documentation.

Let's test it out, by changing the "Sign In" button on the sign in page. It can be a bit tricky to find exactly which override will do what you want, but in this case, the submit button is an *input*, and under AshAuthentication.Phoenix.Components.Password.Input is an override for submit_class. Perfect.

In the overrides file, set a new override for that Input component:

05/lib/tunez_web/auth_overrides.ex

```
defmodule TunezWeb.AuthOverrides do
  use AshAuthentication.Phoenix.Overrides

➤  override AshAuthentication.Phoenix.Components.Password.Input do
```

4. https://hexdocs.pm/ash_authentication_phoenix/ui-overrides.html#reference

```
    set :submit_class, "bg-primary-600 text-white my-4 py-3 px-5 text-sm"
  end
```

Log out and return to the sign-in page, and the sign-in button will now be purple!

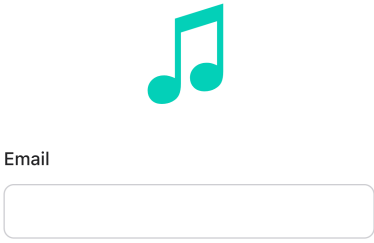
As any overrides we set will completely override the default styles, there may be more of a change than you expect. If you're curious about what the default values for each override are, or you want to copy and paste them so you can only change what you need, you can see them in the AshAuthentication-Phoenix source code.[5]

We won't bore you with every single class change to make, to turn a default AshAuthenticationPhoenix form into one matching the rest of the site theme, so we've provided a set of overrides to use in lib/tunez_web/auth_overrides_sample.txt. Take the full contents of that file and replace the contents of the TunezWeb.AuthOverrides module, like so:

05/lib/tunez_web/auth_overrides.ex

```
defmodule TunezWeb.AuthOverrides do
  use AshAuthentication.Phoenix.Overrides
  alias AshAuthentication.Phoenix.Components

  override Components.Banner do
    set :image_url, nil
    # ...
```

And it should look like this:



Feel free to tweak the styles the way you like - Tunez is your app, after all!

5. https://github.com/team-alembic/ash_authentication_phoenix/blob/main/lib/ash_authentication_phoenix/overrides/default.ex

Why do users always forget their passwords!?

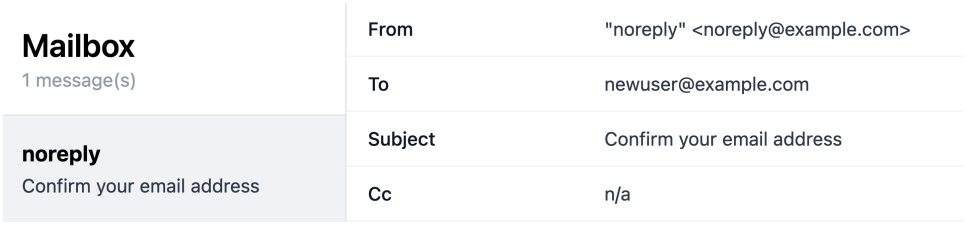
Earlier we mentioned that the app was automatically generating an email to send to users after registration, to confirm their accounts. Let's see what that looks like!

When we added the `password` authentication to Tunez, AshAuthentication generated two modules responsible for generating emails, *senders* in AshAuthentication jargon. These live in `lib/tunez/accounts/user/senders` — there's one for `SendNewUserConfirmationEmail`, and one for `SendPasswordResetEmail`.

Phoenix apps come with a Swoosh[6] integration built in for sending email, and the generated senders have used that. Each sender module defines two critical functions: a `body/1` private function that generates the content for the email, and a `send/3` that is responsible for constructing and sending the email using Swoosh.

We don't need to set up an email provider to send real emails, while working in development. Swoosh provides a "mailbox" we can use — any emails sent, no matter the target email address, will be delivered to the dev mailbox (instead of actually being sent!). This dev mailbox is added to our router in dev mode only, and can be accessed at http://localhost:4000/dev/mailbox.

The mailbox is empty by default, but if you register for a new account via the web app and then refresh the mailbox:



The email contains a link to confirm the email address, which, sure, that's totally my email address and I did sign up for the account, so visit the URL. You'll be redirected to a Tunez confirmation screen, with a button to click to ensure that you do want to verify your account. If you click it, you'll be back on the app homepage, with a flash message letting us know that our email address is confirmed. Success!

6. https://hexdocs.pm/swoosh/

Setting Up Magic Link Authentication

Some users nowadays think that passwords are just *so* passé, and they'd much prefer to be able to log in using magic links instead — enter their email address, click the login link that gets sent straight to their inbox, and they're in. That's no problem!

AshAuthentication doesn't limit our apps to just *one* method of authentication, we can add as many as we like from the supported strategies[7] or even write our own. So there's no problem with adding the magic link strategy to our existing password-strategy-using app, and users can even log in with either strategy depending on their mood. Let's go.

To add the strategy, run the ash_authentication.add_strategy Mix task:

```
$ mix ash_authentication.add_strategy magic_link
```

This will:

- Add a new magic_link authentication strategy block to our Tunez.Accounts.User resource, in lib/tunez/accounts/user.ex
- Add two new actions named sign_in_with_magic_link and request_magic_link, also in our Tunez.Accounts.User resource
- Remove the allow_nil? false on the hashed_password attribute in the Tunez.Accounts.User resource (users that sign in with magic links won't necessarily have passwords!)
- And finally, add a new sender module responsible for generating the magic link email, in lib/tunez/accounts/user/senders/send_magic_link_email.ex.

The magic_link config block in the Tunez.Accounts.User resource lists some sensible values for the strategy configuration, such as the identity attribute (email by default). There are more options[8] that can be set, such as how long generated magic links are valid for (token_lifetime), but we won't need to add anything extra to what is generated here.

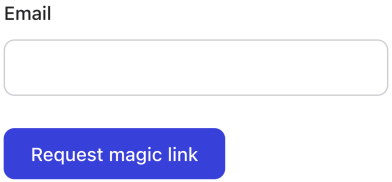
A migration was generated for the allow_nil? false change on the users table, so you'll need to run that:

```
$ mix ash.migrate
```

Wait… that's it? Yep, that's it. The initial setup of AshAuthentication generates a lot of code for the initial resources, but adding subsequent strategies typically only needs a little bit.

7. https://hexdocs.pm/ash_authentication/get-started.html#choose-your-strategies-and-add-ons
8. https://hexdocs.pm/ash_authentication/dsl-ashauthentication-strategy-magiclink.html#options

Once you've added the strategy, visiting the sign-in page will have a nice surprise:



Is it really that simple?? If we fill out the magic link sign-in form with the same email address we confirmed earlier, an email will be delivered to our dev mailbox, with a sign-in link to click. Click the link, and after confirming the login, you *should* be back on the Artist catalog, with a flash message saying that you're now signed in. Awesome!

But you might *not* be signed in automatically. You might be back on the sign in page, with a generic "incorrect email or password" message that doesn't give away any secrets. If not, you can force an error by logging out and visiting the magic link a second time. *Now* you'll get an error! How can we tell what's happening behind the scenes?

Debugging when authentication goes wrong

While showing a generic failure message is good in production for security reasons (we don't want to give away information like if an email address definitely belonged to an account, to potentially-bad actors), it's not good in development while you're trying to make things work, or debugging issues.

To get more information about what's going on, enable authentication debugging for our development environment *only*, by placing the following at the bottom of config/dev.exs:

05/config/dev.exs

```
config :ash_authentication, debug_authentication_failures?: true
```

Restart your mix phx.server to apply the new config change, and visit the same magic link URL again. You should see a big yellow warning in your server logs:

```
[warning] Authentication failed:
Bread Crumbs:
  > Error returned from: Tunez.Accounts.User.sign_in_with_magic_link

Invalid Error

* Invalid magic_link token
  (ash 3.x.xx) lib/ash/error/changes/required.ex:4: Ash.Error.Changes...
```

...

Ah hah, now we know! The magic link token has either already been used, or has expired. Either way, it's not valid anymore.

Without turning the AshAuthentication debugging on, these kinds of issues would be near-impossible to fix. It's safe to leave it enabled in development, as long as you don't mind the warning about it during server start. If the warning is too annoying, feel free to turn debugging off, but don't forget that it's available to you!

And that's all we need to do, to implement magic link authentication in our apps. Users will be able to create accounts via magic links, and also log into their existing accounts that were created with an email and password. Our future users will thank us!

Can we allow authentication over our APIs?

In the previous chapter, we built two shiny APIs that users can use to programmatically access Tunez and its data. In order to make sure the APIs have full feature parity with the web UI, we need to make sure they can register and sign in via the API as well. When we start locking down access to critical parts of the app, we don't want API users to be left out!

Let's give it a try, and see how far we can get. We'll start with adding registration support, in our JSON API.

To add JSON API support to our Tunez.Accounts.User resource, we can extend it using Ash's extend patcher:

```
$ mix ash.extend Tunez.Accounts.User json_api
```

This will configure our JSON API router, domain module, and resource with everything we need to start connecting routes to actions. To create a POST request to our register_with_password action, we can add a new route to the domain,[9] like we did with Adding Albums to the API, on page 92. We've customized the actual URL with the route option, to create a full URL like /api/json/users/register.

05/lib/tunez/accounts.ex

```
defmodule Tunez.Accounts do
  use Ash.Domain, extensions: [AshJsonApi.Domain]

➤   json_api do
➤     routes do
➤       base_route "/users", Tunez.Accounts.User do
```

9. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-domain.html#json_api-routes-base_route-post

```
        post :register_with_password, route: "/register"
      end
    end
  end

  # ...
end
```

Looks good so far! But if you try it in an API client, or using cURL, correctly suppling all the arguments that the action expects… it won't work, it always returns a `forbidden` error. Drat.

This is because at the moment, the `Tunez.Accounts.User` resource is really tightly secured. All of the actions are restricted, to only be accessible via AshAuthenticationPhoenix's form components. (Or if we skip authorization checks, like we did earlier on page 109. That was for test purposes only!)

This is good for security reasons — we don't want *any* old code to be able to do things like change people's passwords! It makes our development lives a little bit harder, though, because to understand how to allow the functionality we want, we need to dive into our next topic, *authorization*. Buckle up, this may be a long one…

CHAPTER 6

Authorization: What Can You Do?

We left Tunez in a good place at the end of the previous chapter. Visitors to the app can now register accounts, or login with either an email and password or a magic link. Now we can identify *who* is using Tunez.

However, we couldn't allow users to authenticate or register an account for Tunez via either of our APIs. The app also doesn't behave any differently depending on whether a user is logged in or not. Anyone can still create, edit and delete data. This is what we want to prevent, for better data integrity — unauthenticated users should have a total read-only view of the app, and authenticated users should be able to perform only the actions they are granted access to. We can enforce this by implementing *access control* in the app, using a component of Ash called *policies*.

Introducing Policies

Policies define who has access to resources within our app, and what actions they can run. Each resource can have its own set of policies, and each policy can apply to one or more actions defined in that resource.

Policies are checked internally by Ash *before* any action is run — if *all* policies that apply to a given action *pass* (return authorized), then the action is run. If one or more of the policies *fail* (return unauthorized), then the action is *not* run and an error is returned.

Because policies are part of resource definitions, they're automatically be checked on all calls to actions in those resources. Write them once, and they'll apply everywhere — in our web UI, our REST and GraphQL APIs, an iex REPL, and any other interfaces we add in future. You don't have to worry about the *when* or *how* of policy checking, you're freed up to focus on the actual business

logic of who can access what. This makes access control simple, straightforward, and fast to implement.

A lot of policy checks will naturally depend on the entity calling the action, or the *actor*. This is usually (but not always!) the person using the app, clicking buttons and links in their browser. This is why it's a pre-requisite to know who our users are!

At its core, a policy is made up of two things:

- One or more *policy conditions*, to determine whether or not the policy applies to a specific action request
- A set of *policy checks*, that are run to see if a policy passes or fails. Each policy check is itself made up of a check condition, and an action to take if the check condition matches the action condition.

There's a lot of conditions to consider, so an example will hopefully make it clearer. If we were building a Blog application, and wanted to add policies for a Post in our blog, a policy might look like the following:

```
defmodule Blog.Post do
  use Ash.Resource, authorizers: [Ash.Policy.Authorizer]

  policies do
    policy action(:publish) do
      forbid_if expr(published == true)
      authorize_if actor_attribute_equals(:role, :admin)
    end
  end
end
```

The single condition for this policy is action(:publish), meaning the policy will apply to any calls to the hypothetical publish action in this Blog.Post resource. It has two policy checks — one that will *forbid* the action *if* the published attribute on the resource is true, and one that will *authorize* the action *if* the actor calling the action has the attribute role set to :admin.

Or in human terms, *admin* users can *publish* a blog post if it's not already *published*. Any other cases would return an error.

As mentioned previously in Loading Related Resource Data, on page 42, all actions (including code interfaces) support an extra argument of options to customize the behaviour of the action. One of the options that all actions support is actor (as seen in the list of options for each action type),[1] so we can modify each action call to add the actor option:

1. https://hexdocs.pm/ash/Ash.html

```
# Publish a blog post while identifying the actor
Blog.Post.publish(post, actor: current_user)
```

Decisions, decisions

When evaluating policy checks, the first check that successfully *makes a decision* determines the overall result of the policy. Policies could be thought of like a cond statement — checks are evaluated in order until one makes a decision, and the rest are ignored — so the order of checks within a policy is really important, perhaps more than it first appears.

In our Blog.Post example, if we attempt to publish a post that has the attribute published equal to true, the first check will make a decision because the check condition is true (the attribute published is indeed equal to true), and the action is to forbid_if the check condition is true.

The second check for the admin role is irrelevant — a decision has already been made, so no-one can publish an already-published blog post, period.

If the order of the checks in our policy was reversed:

```
policy action(:publish) do
  authorize_if actor_attribute_equals(:role, :admin)
  forbid_if expr(published == true)
end
```

Then the logic is actually a bit different. Now we only check the published attribute if the first check *doesn't* make a decision. To not make a decision, the actor would have to have role attribute that *doesn't* equal :admin. In other words, this policy would mean that admin users can publish *any* blog post. Subtly different, enough to possibly cause unintended behaviour in your app.

If none of the checks in a policy make a decision, the default behaviour of a policy is to *forbid* access, as if each policy had a hidden forbid_if always() at the bottom. And once an authorizer is added to a resource, if no policies apply to a call to an action, then the request will also be forbidden. This is perfect for security purposes!

Authorizing API Access for Authentication

Now that we have a bit of context around policies and how they can restrict access to actions, we can check out the policies that were automatically generated in the Tunez.Accounts.User resource when we installed AshAuthentication:

06/lib/tunez/accounts/user.ex

```
defmodule Tunez.Accounts.User do
```

```
  # ...
  policies do
    bypass AshAuthentication.Checks.AshAuthenticationInteraction do
      authorize_if always()
    end

    policy always() do
      forbid_if always()
    end
  end
  # ...
end
```

There are two policies defined in the resource — one *bypass* (more on bypasses on page 137) and one standard policy. The AshAuthenticationInteraction module, used in the the bypass, contains a custom policy condition that applies to any actions called from AshAuthenticationPhoenix's liveviews, and it will *always* authorize them. This allows the web UI to work out of the box, and allows actions like register_with_password or sign_in_with_magic_link to be run.

The second policy *always* applies to any other action call (the policy condition), and it will *forbid if*, well, *always*! This includes action calls from any of our generated API endpoints, and explains why we always got a forbidden error when we tried to register an account via the API. The User resource is open for AshAuthentication's action calls, and firmly closed for absolutely everything else. Let's update this and write some new policies, to allow access to the actions we want.

Writing our first user policy

If you test running any actions on the Tunez.Accounts.User resource in iex (without using with authorize?: false, like we did earlier on page 109), you'll see the forbidden errors that were getting processed by AshJsonApi, and returned in our API responses.

```
iex(1)> Tunez.Accounts.User
Tunez.Accounts.User
iex(2)> |> Ash.Changeset.for_create(:register_with_password, %{email:
...(2)> «email», password: "password", password_confirmation: "password"})
#Ash.Changeset<...>
iex(3)> |> Ash.create()
{:error, %Ash.Error.Forbidden{...}}
```

It wouldn't hurt to have *some* of the actions on the resource, such register_with_password and sign_in_with_password, accessible over the API. To do that, you can remove the policy always() policy, and replace it with a new policy that will *authorize* calls to those specific actions:

06/lib/tunez/accounts/user.ex

```
policies do
  bypass AshAuthentication.Checks.AshAuthenticationInteraction do
    authorize_if always()
  end

  policy action([:register_with_password, :sign_in_with_password]) do
    authorize_if always()
  end
end
```

This is the most permissive type of policy check. As it says on the tin, it *always* authorizes any action that meets the policy condition — any action that has one of those two names.

Authorizing a sign-in action does *not* mean the sign-in will be successful!

There's a difference between an action being *authorized* ("this user is allowed to run this action"), and an action returning a *successful result* ("this user provided valid credentials and is now signed in").

In our example, if a user attempts to sign-in with an incorrect password, the action will be authorized, so the sign-in action will run… and return an authentication failure because of the incorrect password.

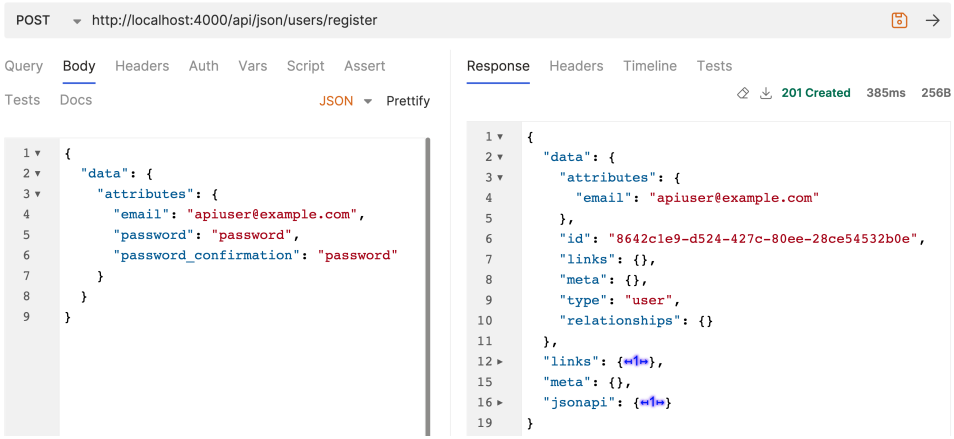
With that policy in place, if you recompile in iex and then re-run the action to register a user, it will now have a very different result:

```
iex(1)> Tunez.Accounts.User
Tunez.Accounts.User
iex(2)> |> Ash.Changeset.for_create(:register_with_password, %{email:
...(2)> «email», password: "password", password_confirmation: "password"})
#Ash.Changeset<...>
iex(3)> |> Ash.create()
{:ok, #Tunez.Accounts.User<...>}
```

This is what we want! Is that all we needed to do?

Authenticating via JSON

When we left the JSON API at the end of the previous chapter, we'd configured a route in our Tunez.Accounts domain for the register_with_password action of the User resource, but it always returned a forbidden error. With the new policies we've added, we can test it with an API client again:



It does create a user, but the response isn't quite right. When we tested out authentication earlier in Testing authentication actions in iex, on page 110, the user data included an authentication token as part of its metadata, to be used to authenticate future requests.

```
iex(11)> |> Ash.read(authorize?: false)
{:ok, [#Tunez.Accounts.User<...>]
iex(12)> user.__metadata__.token
"eyJhbGciOi..."
```

That was for sign-in, not registration, but the same principle applies — when you register an account, you're typically automatically signed into it.

This token needs to be included as part of the response, so API clients can store it and send it as a request header with future requests to the API. This is handled for us with AshAuthenticationPhoenix, with cookies, sessions and plugs, but for an API the clients must handle it themselves.

In AshJsonApi, we can attach extra *metadata* to an API response.[2] The user's authentication token sounds like a good fit! The `metadata` option for a route takes a three-argument function that includes the data being returned in the response (the created user), and returns a map of data to include, so we can extract the token and return it:

06/lib/tunez/accounts.ex

```
post :register_with_password do
  route "/register"

  metadata fn _subject, user, _request ->
    %{token: user.__metadata__.token}
  end
```

2. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-resource.html#json_api-routes-post-metadata

```
end
```

The same process can be used for creating the sign-in route — give it a nice URL, and add the token to the response:

06/lib/tunez/accounts.ex

```
base_route "/users", Tunez.Accounts.User do
  # ...

  post :sign_in_with_password do
    route "/sign-in"

    metadata fn _subject, user, _request ->
      %{token: user.__metadata__.token}
    end
  end
end
```

Other authenticated-related actions, such as magic links or password resetting, don't make as much sense to perform over an API — they all require following links in emails that go to the app to complete. It *can* be done, but it's a much less common use case so we'll leave it as an exercise for the reader!

Authenticating via GraphQL

Adding support for authentication to our GraphQL API is easier than for the JSON API, now that we've granted access to the actions via policies — some of the abstractions around metadata are already built in.

To get started, extend the Tunez.Accounts.User resource with AshGraphql, using Ash's extend patcher:

```
$ mix ash.extend Tunez.Accounts.User graphql
```

This will configure our GraphQL schema, domain module, and resource with everything we need to start creating queries and mutations for actions.

AshGraphql is strict in regards to which types of actions can be defined as mutations, and which as queries. Read actions must be queries; and creates, updates and destroys must be mutations. So for the two actions we want to make accessible via GraphQL, create :register_with_password must be a mutation, and read :sign_in_with_password must be a query.

To create a mutation for the register_with_password action, define a new graphql block in the Tunez.Accounts domain module and populate it using the create macro:[3]

3. https://hexdocs.pm/ash_graphql/dsl-ashgraphql-domain.html#graphql-mutations-create

06/lib/tunez/accounts.ex

```
defmodule Tunez.Accounts do
  use Ash.Domain, extensions: [AshGraphql.Domain, AshJsonApi.Domain]

  graphql do
    mutations do
      create Tunez.Accounts.User, :register_user, :register_with_password
    end
  end
  # ...
```

This is exactly the same as mutations we added for creating or updating artist and album records — the type of action, the name of the resource module, the name to use for the generated mutation, and the name of the action.

And that's all that needs to be done! AshGraphql can tell that the action has metadata, and that metadata should be exposed as part of the response. You can test it out in the GraphQL playground, by calling the mutation with an email, password, and password confirmation, the same way as you would on the web registration form, and reading back the token as part of the metadata:

```
mutation {
  registerUser(input: {email: "test@example.com", password: "mypassword",
                       passwordConfirmation: "mypassword"}) {
    result { id email }
    metadata { token }
  }
}
```

To create a query for the sign_in_with_password action, update the graphql block in the Tunez.Accounts domain module and add the query using the get macro:[4]

06/lib/tunez/accounts.ex

```
defmodule Tunez.Accounts do
  use Ash.Domain, extensions: [AshGraphql.Domain, AshJsonApi.Domain]

  graphql do
    queries do
      get Tunez.Accounts.User, :sign_in_user, :sign_in_with_password
    end

    # ...
```

But instead of seeing the new query reflected in the GraphQL playground schema, this change will raise a compilation error!

```
** (Spark.Error.DslError) [Tunez.Accounts]
Queries for actions with metadata must have a type configured on the query.

The sign_in_with_password action on Tunez.Accounts has the following metadata
```

4. https://hexdocs.pm/ash_graphql/dsl-ashgraphql-domain.html#graphql-queries-get

```
fields:

* token

To generate a new type and include the metadata in that type, provide a new
type name, for example `type :user_with_token`.
```

AshGraphql doesn't know what to do with the authentication token, returned as metadata on the user record. GraphQL mutations return data wrapped in a result field, leaving space for other fields like errors and metadata, but queries return plain data — nowhere to add the metadata in the response. We need either a new GraphQL return type for this action, to combine the token with the user record, or we can explicitly ignore the metadata.

The error suggests a name to use if we want to include the metadata — :user_with_token — so add that to the query definition in the Tunez.Accounts module.

06/lib/tunez/accounts.ex

```
queries do
  get Tunez.Accounts.User, :sign_in_user, :sign_in_with_password do
➤   type_name :user_with_token
  end
end
```

Now our app will compile, and we can see the query in the GraphQL playground schema. It takes an email and password as inputs, but also… an id input? This is because the get macro is typically used for fetching records by their primary key, or id. Not what we want in this case! Disable this id input by adding identity false[5] to the query definition as well.

06/lib/tunez/accounts.ex

```
queries do
  get Tunez.Accounts.User, :sign_in_user, :sign_in_with_password do
➤   identity false
    type_name :user_with_token
  end
end
```

We can now call the query in the playground, and get either user data (and a token) back, or an authentication failure if we provided invalid credentials. The user token can then be used to authenticate subsequent API requests, by setting the token in an Authorization HTTP header. And now users can successfully sign in, register, and authenticate via both of our APIs.

Now we can look at more general authorization, for the rest of Tunez. This data isn't going to secure itself!

5. https://hexdocs.pm/ash_graphql/dsl-ashgraphql-domain.html#graphql-queries-get-identity

Assigning Roles to Users

Access control — who can access what — is a massive topic. Systems to control access to data can range from the simple (users can perform actions based on a single role), to the more complex (users can be assigned roles in one or more groups, each with its own permissions) to the very very fine-grained (users can be granted specific permissions per piece of data, on top of everything else).

What level of access control you need heavily depends on the app you're building, and it may change over time. For Tunez, we don't need anything complicated, we only want to make sure data doesn't get vandalized, so we'll implement a more simple system with *roles*.

Each user will have an assigned *role*, that determines which actions they can run and what data they can modify. We'll have three different roles:

- Basic users won't be able to modify any artist/album data
- Editors will be able to create/update a limited set of data
- Admins will be able to perform any action across the app

This role will be stored in a new attribute in the Tunez.Accounts.User resource, named role. This attribute could be an atom, and we could add a constraint[6] to specify that it must be one of our list of valid role atoms:

```
attributes do
  # ...

  attribute :role, :atom do
    allow_nil? false
    default :user
    constraints [one_of: [:admin, :editor, :user]]
  end
end
```

Ash provides a better way to handle enum-type values, though, with the Ash.Type.Enum[7] behaviour. We can define our roles in a separate Tunez.Accounts.Role module:

06/lib/tunez/accounts/role.ex

```
defmodule Tunez.Accounts.Role do
  use Ash.Type.Enum, values: [:admin, :editor, :user]
end
```

And then specify that the role attribute is actually a Tunez.Accounts.Role:

6. https://hexdocs.pm/ash/dsl-ash-resource.html#attributes-attribute-constraints
7. https://hexdocs.pm/ash/Ash.Type.Enum.html

06/lib/tunez/accounts/user.ex

```
attributes do
  # ...

➤  attribute :role, Tunez.Accounts.Role do
➤    allow_nil? false
➤    default :user
➤  end
end
```

This has a couple of neat benefits. We can fetch a list of all of the valid roles, with Tunez.Accounts.Role.values/0, and we can also specify human-readable names and descriptions for each role, useful if you wanted a page where you could select a role from a dropdown list.

Generate a migration for the new role attribute, and run it:

```
$ mix ash.codegen add_role_to_users
$ mix ash.migrate
```

The attribute is created as a text column in the database, and the default option[8] we used is also passed through to be a default in the database. This means that we don't need to manually set the role for any existing users, or any new users that sign up for accounts, they'll all automatically have the role user.

What about custom logic for assigning roles?

Maybe you don't want to hardcode a default role for all users — maybe you want to assign the 'editor' role to users who register with a given email domain, for example.

You can implement this with a custom change module, similar to UpdatePreviousNames on page 55. The wrinkle here is that users can be created either from signing up with a password, or signing in with a magic link (which creates a new account if one doesn't exist for that email address). The change code would need to be used in both actions, and support magic link sign-ins for both new and existing accounts.

We do need some way of changing a user's role, though, otherwise Tunez can never have any editors or admins! We'll add a utility action to the Tunez.Accounts.User resource — a new update action that *only* allows setting the role attribute for a given user record:

06/lib/tunez/accounts/user.ex

```
actions do
  defaults [:read]
```

8. https://hexdocs.pm/ash/dsl-ash-resource.html#attributes-attribute-default

```
    update :set_role do
      accept [:role]
    end

    # ...
```

Adding a code interface for the new action will make it easier to run, so add it (and another helper for reading users) within the Tunez.Accounts.User resource definition in the Tunez.Accounts domain module.

06/lib/tunez/accounts.ex

```
defmodule Tunez.Accounts do
  # ...

  resources do
    # ...
    resource Tunez.Accounts.User do
      define :set_user_role, action: :set_role, args: [:role]
      define :get_user_by_id, action: :read, get_by: [:id]
    end
```

As we'll never be calling these functions from code, only running them manually in an iex console, we don't need to add a policy that will authorize them — we can skip authorization using the authorize?: false option when we call them.

Once you've registered an account in your development Tunez app, you can then change it to be an admin in iex:

```
iex(1)> user = Tunez.Accounts.get_user_by_id!(«uuid», authorize?: false)
#Tunez.Accounts.User<...>
iex(2)> Tunez.Accounts.set_user_role(user, :admin, authorize?: false)
{:ok, #Tunez.Accounts.User<role: :admin, ...>}
```

Now that we have users with roles, we can define which roles can perform which actions. We can do this by writing some more policies, to cover actions in our Artist and Album resources.

Writing Policies for Artists

There are two parts we need to complete, when implementing access control in an Ash web app:

- Creating the policies for our resources, and
- Updating our web interface to specify the actor when calling actions, as well as niceties like hiding buttons to perform actions if the current user doesn't have permission to run them.

Which order you do them in is up to you, but both will need to be done. Writing the policies first will break the web interface completely (everything will be forbidden, without knowing who is trying to load or modify data!) but

doing it first will ensure we catch all permission-related issues when tweaking our liveviews. We'll write the policies first, while they're fresh in our minds.

Creating our first artist policy

To start adding policies to a resource for the first time, we first need to configure it with the Ash policy authorizer. In the Tunez.Music.Artist resource, that looks like this:

06/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  use Ash.Resource,
    otp_app: :tunez,
    domain: Tunez.Music,
    data_layer: AshPostgres.DataLayer,
    extensions: [AshGraphql.Resource, AshJsonApi.Resource],
    authorizers: [Ash.Policy.Authorizer]
```

Testing a create action

The default behaviour in Ash is to authorize (run policy checks for) any action in a resource that has an authorizer configured, so straight away we get forbidden errors when attempting to run any actions on the Tunez.Music.Artist resource:

```
iex(1)> Tunez.Music.create_artist(%{name: "New Artist"})
{:error,
  %Ash.Error.Forbidden{
    bread_crumbs: ["Error returned from: Tunez.Music.Artist.create"],
    changeset: "#Changeset<>",
    errors: [%Ash.Error.Forbidden.Policy{...}]
  }
}
```

There are no policies for the create action, so it is automatically forbidden.

We can add a policy for the action by adding a new policies block at the top level of the Tunez.Music.Artist resource, and adding a sample policy that applies to the action:

06/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  policies do
    policy action(:create) do
      authorize_if always()
    end
  end
end
```

This is the most permissive type of policy check — it *always* authorizes the given policy. With this policy in our resource, after recompiling we can now create artists again:

```
iex(1)> Tunez.Music.create_artist(%{name: "New Artist"})
{:ok, #Tunez.Music.Artist<...>}
```

There are a whole set of policy checks built into Ash,[9] but the most common ones for policy conditions are action and action_type. Our policy, as written, only applies to the action *named* create, but if we wanted to use it for any action of *type* create, we could use action_type(:create) instead.

Of course, we don't always want to blanket authorize actions, as that would defeat the purpose of authorization entirely. Let's update the policy to only allow admin users to create artist records, ie. *actors* who have their *role* attribute set to :admin:

06/lib/tunez/music/artist.ex

```
policies do
  policy action(:create) do
    authorize_if actor_attribute_equals(:role, :admin)
  end
end
```

Because our policy refers to an actor, we have to pass in the actor when calling the action. If we don't, or pass in nil instead, the check won't make a decision. If none of the checks specifically authorize the policy, it safely defaults to being unauthorized. We can test what the action does in iex, by creating different types of Tunez.Accounts.User structs and running the create action:

```
iex(2)> Tunez.Music.create_artist(%{name: "New Artist"})
{:error, %Ash.Error.Forbidden{...}}

iex(3)> Tunez.Music.create_artist(%{name: "New Artist"}, actor: nil)
{:error, %Ash.Error.Forbidden{...}}

iex(4)> editor = %Tunez.Accounts.User{role: :editor}
#Tunez.Accounts.User<role: :editor, ...>
iex(5)> Tunez.Music.create_artist(%{name: "New Artist"}, actor: editor)
{:error, %Ash.Error.Forbidden{...}}

iex(6)> admin = %Tunez.Accounts.User{role: :admin}
#Tunez.Accounts.User<role: :admin, ...>
iex(7)> Tunez.Music.create_artist(%{name: "New Artist"}, actor: admin)
{:ok, #Tunez.Music.Artist<...>}
```

The only actor that was authorized to create the artist record was the admin user — just as we intended.

9. https://hexdocs.pm/ash/Ash.Policy.Check.Builtins.html

Filling out update and destroy policies

In a similar way, we can write policies for the update and destroy actions in the Tunez.Music.Artist resource. Admin users should be able to perform both actions; and we'll also allow editors (users with role: :editor) to update records.

06/lib/tunez/music/artist.ex

```
policies do
  # ...

  policy action(:update) do
    authorize_if actor_attribute_equals(:role, :admin)
    authorize_if actor_attribute_equals(:role, :editor)
  end

  policy action(:destroy) do
    authorize_if actor_attribute_equals(:role, :admin)
  end
end
```

If an editor attempts to update an artist record, the first check won't make a decision — their role isn't :admin, so the next check is looked at. This one makes a decision to authorize, so the policy is authorized and the action will run.

Cutting out repetitiveness with bypasses

When we have some all-powerful role like admin, it can be really repetitive to write authorize_if actor_attribute_equals(:role, :admin) in every. Single. Policy. It'd be much nicer to be like, oh, these users? They're special, just let 'em on through. We can do that by using a *bypass*.[10]

With standard policies defined using policy, *all* applicable policies for an action must apply. This allows for cases like the following:

```
policy action_type(:update) do
  authorize_if actor_present()
end

policy action(:force_update) do
  authorize_if actor_attribute_equals(:role, "admin")
end
```

Our hypothetical resource has a few update-type actions that any authenticated user can run (using the built-in actor_present policy check),[11] but also a special force_update that only admin users can run. It's not that the action(:force_update) policy takes precedence over the action_type(:update) policy, it's that *both* policies

10. https://hexdocs.pm/ash/dsl-ash-policy-authorizer.html#policies-bypass
11. https://hexdocs.pm/ash/Ash.Policy.Check.Builtins.html#actor_present/0

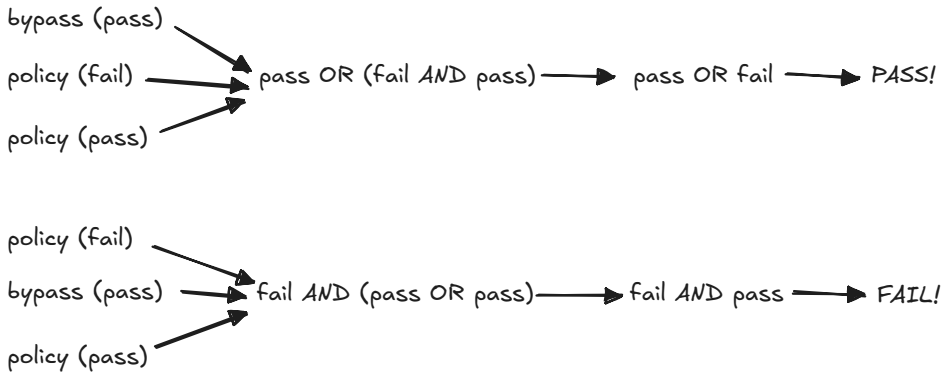
apply and have to pass when calling the force_update action, but only one applies to other update actions.

Bypass policies are different. If a bypass policy *authorizes* an action, it can skip all other policies that apply to that action.

```
bypass actor_attribute_equals(:role, :admin) do
  authorize_if always()
end

# If the bypass passes, then this policy doesn't matter!
policy action_type(:update) do
  forbid_if always()
end
```

We say that a bypass *can* skip other policies, not that it *will*, because it's a little more complicated than that — order matters when it comes to mixing bypass policies with standard policies.

Internally, Ash converts the results of running each policy into a big boolean expression with passing authorization being true and failing authorization being false, which is evaluated by the SAT solver we installed earlier. Standard policies are AND-ed into the expression, so all need to be authorized for the action to be authorized. Bypass policies are OR-ed into the expression, so changing the order of policies within a resource can drastically affect the result. See the following example, where the policy results are the same, but the order is different, leading to a different overall result:



Bypasses are really powerful, and allow abstracting common authorization logic into one place. It's possible to encode very complicated logic, but it's also pretty easy to make a mess or have unintended results. We'd recommend the following guidelines:

- Keep all bypass policies together at the start of the policies block, and don't intermingle them with standard policies, and
- Write naive tests for your policies that test as many combinations of permissions as possible, to verify that the behaviour is what you expect. More on testing in the next chapter on page 157.

Debugging when policies fail

Policies can get complex, with multiple conditions in each policy check, multiple policy checks of different types within a policy, and as we've just seen, multiple policies that apply to a single action request, including bypasses.

We can tell if authorization fails, because we get an Ash.Error.Forbidden struct back from the action request, but we can't necessarily see *why* it might fail.

Ash can display *breakdowns* of how policies were applied to an action request. Similar to how we set a config option to debug authentication failures in the last chapter, we can do the same thing for policies.[12]

In your dev config in config/dev.exs, turn on policy breakdowns:

06/config/dev.exs

```
config :ash, :policies, show_policy_breakdowns?: true
```

After restarting your iex session, repeat the previous experiment by trying to create an artist with an actor that isn't an admin:

```
iex(1)> editor = %Tunez.Accounts.User{role: :editor}
#Tunez.Accounts.User<...>
iex(2)> Tunez.Music.create_artist!(%{name: "Oh no!"}, actor: editor)
** (Ash.Error.Forbidden)
Bread Crumbs:
  > Error returned from: Tunez.Music.Artist.create

Forbidden Error

* forbidden:

Tunez.Music.Artist.create

Policy Breakdown
  user: %{id: nil}

  Policy | [M]:

    condition: action == :create

    authorize if: actor.role == :admin | ✘ | [M]

SAT Solver statement:

 "action == :create" and
```

12. https://hexdocs.pm/ash/policies.html#policy-breakdowns

```
  (("action == :create" and "actor.role == :admin")
    or not "action == :create")
```

Note that the [M]s are actually magnifying glass emojis in the terminal.

Similar output can be seen in a browser, if you attempt to visit the app (because it's all broken right now!) It's a little bit verbose, but it clearly states which policies have applied to this action call, and what the results were — this user was not an admin, so they get a big ✘.

Filtering results in read action policies

The last actions we have to address in the Artist resource are our two read actions — the default read action, and our custom search action.

So far we've only looked at checks for single records, with yes/no answers — can the actor run this action on this record, yes or no? Read actions are built a little differently, as they don't start with an initial record to operate on and they don't modify data. Policies for read actions behave as *filters* — given all of the records that the action would fetch, which ones are the actor allowed to see?

If, for example, we had the following policy that uses a secret attribute:

```
policy action_type(:read) do
  authorize_if expr(secret == false)
end
```

You can call the action as normal, via code like MyResource.read(actor: user), and the results will only include records where the secret attribute has the value false.

If a policy check would have different answers depending on the record being checked (ie. it checks some property of the record, like the value of the secret attribute) we say this is a *filter* check. If it depends only on the actor, or a static value like always(), then we say it's a *simple* check.

Filter checks and simple checks can be included in the same policy, eg. to allow admins to read all records, but non-admins can only read non-secret records:

```
policy action_type(:read) do
  authorize_if expr(secret == false)
  authorize_if actor_attribute_equals(:role, :admin)
end
```

Trust, but verify!

One quirk of read policies is distinguishing between "the actor can't run the action" and "the actor *can* run the action, but all of the results are filtered out".

By default, all `read` actions are runnable and all checks are applied as filters. If you want the whole action to be forbidden on authorization failure, this can be configured in the policy using the `access_type` option.[13]

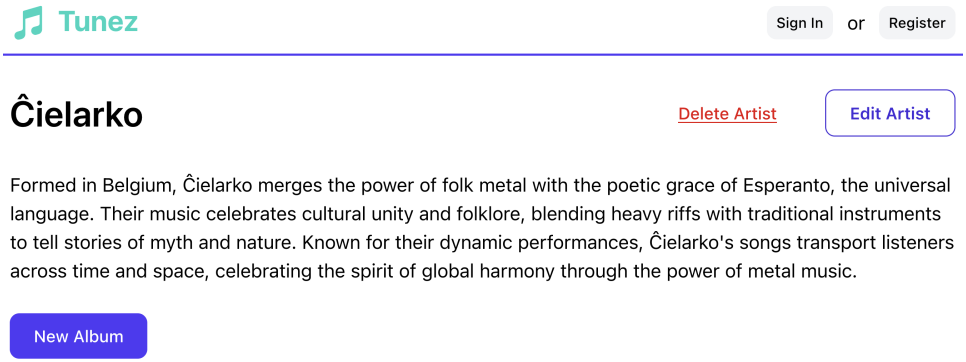
Tunez won't have any restrictions on reading artists, but we *do* need to have policies for all actions in a resource once we start adding them, so we can add a blanket `authorize_if always()` policy:

06/lib/tunez/music/artist.ex

```
policies do
  # ...

  policy action_type(:read) do
    authorize_if always()
  end
end
```

Removing Forbidden Actions from the UI

At the moment, the Artist resource in Tunez is secure — actions that modify data can only be called if a) we pass in a user record as the actor and b) that actor is authorized to run that action. The web UI doesn't reflect these changes, though. Even when not logged in to the app, we can still see buttons and forms inviting us to create, edit, or delete data.



13. https://hexdocs.pm/ash/policies.html#access-type

We can't actually *run* the actions, so clicking the buttons and submitting the forms will return an error, but it's not a good user experience to see them at all. And if we *are* logged in and we *should* have access to manage data, we *still* get an error! Oops.

There are a few things we need to do, to make the UI behave correctly for any kind of user viewing it:

- Update all of our action calls to pass the current user as the actor
- Update our forms to ensure we only let the current user see them if they can submit them
- And lastly, update our templates to only show buttons if the current user is able to use them.

It sounds like a lot, but it's only a few changes to make, spread across a few different files. Let's dig in!

Identifying the actor when calling actions

For a more complex app, this would be the biggest change from a functionality perspective — allowing actions to be called by users who *are* authorized to do things. Tunez is a lot simpler, and most of the data management is done via forms, so this isn't a massive change for us. The only actions we call directly are read and destroy actions:

- Tunez.Music.search_artists/2, in Tunez.Artists.IndexLive. We don't *need* to pass the actor in here, as all of our policies for read actions will always authorize the action, but that could change in the future!

06/lib/tunez_web/live/artists/index_live.ex

```
def handle_params(params, _url, socket) do
  # ...

  page =
    Tunez.Music.search_artists!(query_text,
      page: page_params,
      query: [sort_input: sort_by],
      actor: socket.assigns.current_user
    )
```

- Tunez.Music.get_artist_by_id/2, in Tunez.Artists.ShowLive. Same as above. It does no harm to set the actor either, so we'll add it.

06/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(%{"id" => artist_id}, _session, socket) do
  artist =
    Tunez.Music.get_artist_by_id!(artist_id,
      load: [:albums],
```

```
      actor: socket.assigns.current_user
    )
```

- Tunez.Music.get_artist_by_id/2, in Tunez.Artists.FormLive. Same as above!

06/lib/tunez_web/live/artists/form_live.ex

```
def mount(%{"id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id,
    actor: socket.assigns.current_user
  )

  # ...
```

- Tunez.Music.destroy_artist/2, in Tunez.Artists.ShowLive. We *need* to pass the actor in here to make it work, as only specific types of users can delete artists.

06/lib/tunez_web/live/artists/show_live.ex

```
def handle_event("destroy-artist", _params, socket) do
  case Tunez.Music.destroy_artist(
    socket.assigns.artist,
    actor: socket.assigns.current_user
  ) do
    # ...
```

- Tunez.Music.destroy_album/2, in Tunez.Artists.ShowLive. We haven't added policies for albums yet, but it doesn't hurt to start updating our templates to support them.

06/lib/tunez_web/live/artists/show_live.ex

```
def handle_event("destroy-album", %{"id" => album_id}, socket) do
  case Tunez.Music.destroy_album(
    album_id,
    actor: socket.assigns.current_user
  ) do
    # ...
```

- Tunez.Music.get_album_by_id/2, in Tunez.Albums.FormLive. Same as above.

06/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"id" => album_id}, _session, socket) do
  album = Tunez.Music.get_album_by_id!(album_id,
    load: [:artist],
    actor: socket.assigns.current_user
  )

  # ...
```

- Tunez.Music.get_artist_by_id/2, in Tunez.Albums.FormLive. Same as above!

06/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"artist_id" => artist_id}, _session, socket) do
  artist = Tunez.Music.get_artist_by_id!(artist_id,
    actor: socket.assigns.current_user
```

```
  )
  # ...
```

Not *too* onerous! Moving forward, we'll add the actor to every action we call, to avoid this kind of rework.

Updating forms to identify the actor

We also need to add authorization checks to forms, as we create and edit both artists and albums via forms. There are two parts to this: setting the actor when building the forms, and ensuring that the form is submittable.

We don't want to show the form at all if the user wouldn't be able to submit it, so we need to run the submittable check before rendering, in the mount/3 functions of Tunez.Artists.FormLive:

06/lib/tunez_web/live/artists/form_live.ex

```
def mount(%{"id" => artist_id}, _session, socket) do
  # ...
  form =
    Tunez.Music.form_to_update_artist(
      artist,
➤     actor: socket.assigns.current_user
    )
➤   |> AshPhoenix.Form.ensure_can_submit!()
  # ...
def mount(_params, _session, socket) do
  form =
    Tunez.Music.form_to_create_artist(
➤     actor: socket.assigns.current_user
    )
➤   |> AshPhoenix.Form.ensure_can_submit!()
```

AshPhoenix.Form.ensure_can_submit!/1[14] is a neat little helper function that authorizes the configured action and data in the form using our defined policies, to make sure the actor can submit it. If the authorization fails, then an exception will be raised.

We can make the same changes to the mount/3 functions in Tunez.Albums.FormLive:

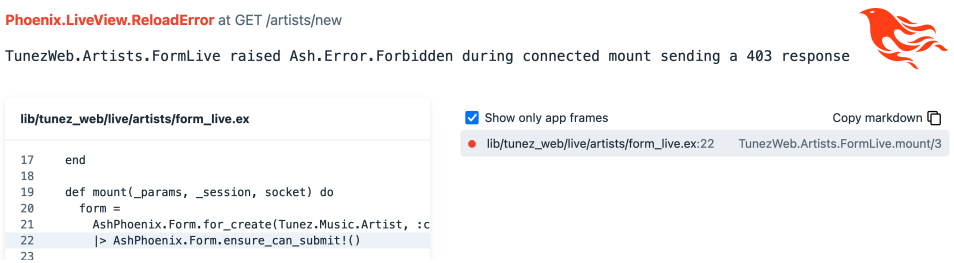
06/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"id" => album_id}, _session, socket) do
  # ...
  form =
    Tunez.Music.form_to_update_album(
```

14. https://hexdocs.pm/ash_phoenix/AshPhoenix.Form.html#ensure_can_submit!/1

```
        album,
➤       actor: socket.assigns.current_user
      )
➤     |> AshPhoenix.Form.ensure_can_submit!()

  # ...

def mount(%{"artist_id" => artist_id}, _session, socket) do
  # ...

  form =
    Tunez.Music.form_to_create_album(artist.id,
➤     actor: socket.assigns.current_user
    )
➤   |> AshPhoenix.Form.ensure_can_submit!()
```

Now if you click any of the buttons that link to pages focussed on forms, when not logged in as a user with the correct role, an exception will be raised and you'll get a standard Phoenix error page:



That works well for forms — but what about entire pages? Maybe we've built an admin-only area, or we've added an Artist version history page that only editors can see. We can't use the same form helpers to ensure access, but we can prevent users from accessing what they shouldn't.

Blocking pages from unauthorized access

When we installed AshAuthenticationPhoenix, one file that the installer created was the TunezWeb.LiveUserAuth module, in lib/tunez_web/live_user_auth.ex. It contains several on_mount function definitions, that do different things based on the authenticated user (or lack of).

The live_user_optional function head will make sure there's always a current_user set in the socket assigns, even if it's nil; the live_user_required function head will redirect away if there's no user logged in, and the live_no_user function head will redirect away if there *is* a user logged in!

These are LiveView-specific helper functions,[15] that can be called at the root level of any liveview like so:

```
defmodule Tunez.Accounts.ForAuthenticatedUsersOnly do
  use TunezWeb, :live_view

  # or :live_user_optional, or :live_no_user
  on_mount {TunezWeb.LiveUserAuth, :live_user_required}

  # ...
```

So to block a liveview from unauthenticated users, we could drop that on_mount call with :live_user_required in that module, and the job would be done!

We can add more function heads to the TunezWeb.LiveUserAuth module for custom behaviour, such as role checking.

06/lib/tunez_web/live_user_auth.ex

```
defmodule TunezWeb.LiveUserAuth do
  # ...

  def on_mount([role_required: role_required], _, _, socket) do
    current_user = socket.assigns[:current_user]

    if current_user && current_user.role == role_required do
      {:cont, socket}
    else
      socket =
        socket
        |> Phoenix.LiveView.put_flash(:error, "Unauthorized!")
        |> Phoenix.LiveView.redirect(to: ~p"/")

      {:halt, socket}
    end
  end
end
```

This would allow us to write on_mount calls in a liveview like:

```
defmodule Tunez.Accounts.ForAdminsOnly do
  use TunezWeb, :live_view

  on_mount {TunezWeb.LiveUserAuth, role_required: :admin}

  # ...
```

Now we can secure all of our pages really neatly, both those that are form-based, and those that aren't. We still shouldn't see any shiny tempting buttons for things we can't access, though, so let's hide them if the user can't perform the actions.

15. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html#on_mount/1

Hiding calls to action that the actor can't perform

There are buttons sprinkled throughout our liveviews and components — buttons for creating, editing, and deleting artists; and for creating, updating and deleting albums. We can use Ash's built-in helpers to add general authorization checks to each of them, meaning we don't have to duplicate any policy logic and we won't need to update any templates if our policy rules change.

Ash.can?

Ash.can?[16] is a pretty low-level function. It takes a tuple representing the action to call and an actor, runs the authorization checks for the action, and returns a boolean representing whether or not the action is authorized:

```
iex(1)> Ash.can?({Tunez.Music.Artist, :create}, nil)
false
iex(2)> Ash.can?({Tunez.Music.Artist, :create}, %{role: :admin})
true
iex(3) artist = Tunez.Music.get_artist_by_id!(«uuid»)
#Tunez.Music.Artist<id: «uuid», ...>
iex(4)> Ash.can?({artist, :update}, %{role: :user})
false
iex(5)> Ash.can?({artist, :update}, %{role: :editor})
true
```

The format of the action tuple looks a lot like how you would run the action manually, as we covered in Running actions, on page 12 — building a changeset for a create action with Ash.Changeset.for_create(Tunez.Music.Artist, :create, ...), or for an update action with Ash.Changeset.for_update(artist, :update, ...).

Our liveviews and components don't call actions like this though, we use code interfaces because they're a lot cleaner. Ash also defines some helper functions around authorization for code interfaces, that are much nicer to read.

can_*? code interface functions

We call these can_*? functions because the names are dynamically generated based on the name of the code interface. For our Tunez.Music domain, for example, iex shows a whole set of functions with the can_ prefix:

```
iex(1)> Tunez.Music.can_
can_create_album/1        can_create_album/2        can_create_album/3
can_create_album?/1       can_create_album?/2       can_create_album?/3
can_create_artist/1       can_create_artist/2       can_create_artist/3
...
```

16. https://hexdocs.pm/ash/Ash.html#can?/3

This list includes can_*? functions for *all* code interfaces, even ones that don't have policies applied yet like Tunez.Music.destroy_album. If authorization isn't configured for a resource, both Ash.can? and the can_*? functions will simply return true, so we can safely add authorization checks to our templates without fear of breaking anything.

One important thing to note is the order of arguments to the code interface helpers. Whereas Ash.can? always takes an action tuple and an actor (plus options), the first argument to a can_*? function is always the actor. Some of the action tuple information is now in the function name itself, and if the code interface needs extra information like a record to operate on or params, those come *after* the actor argument.

```
iex(4)> h Tunez.Music.can_create_artist?

        def can_create_artist?(actor, params_or_opts \\ %{}, opts \\ [])

Runs authorization checks for Tunez.Music.Artist.create, returning a boolean.
See Ash.can?/3 for more information

iex(5)> h Tunez.Music.can_update_artist?

    def can_update_artist?(actor, record, params_or_opts \\ %{}, opts \\ [])

Runs authorization checks for Tunez.Music.Artist.update, returning a boolean.
See Ash.can?/3 for more information
```

Armed with this new knowledge, we can now update the buttons in our templates, to wrap them in HEEx conditionals to only show the buttons if the relevant can_*? function returns true. There's one button in Tunez.Artists.IndexLive, for creating an artist:

06/lib/tunez_web/live/artists/index_live.ex

```
<.header responsive={false}>
  <% # ... %>
➤ <:action :if={Tunez.Music.can_create_artist?(@current_user)}>
    <.button_link [opts]>New Artist</.button_link>
  </:action>
</.header>
```

Two buttons in the header of Tunez.Artists.ShowLive, for editing/deleting an artist:

06/lib/tunez_web/live/artists/show_live.ex

```
<.header>
  <% # ... %>
➤ <:action :if={Tunez.Music.can_destroy_artist?(@current_user, @artist)}>
    <.button_link [opts]>Delete Artist</.button_link>
  </:action>
➤ <:action :if={Tunez.Music.can_update_artist?(@current_user, @artist)}>
    <.button_link [opts]>Edit Artist</.button_link>
  </:action>
```

```
</.header>
```

One above the album list in Tunez.Artists.ShowLive, for creating an album:

06/lib/tunez_web/live/artists/show_live.ex

```
<.button_link navigate={~p"/artists/#{@artist.id}/albums/new"} kind="primary"
  :if={Tunez.Music.can_create_album?(@current_user)}>
  New Album
</.button_link>
```

And two in the album_details function component in Tunez.Artists.ShowLive, for editing/deleting an album:

06/lib/tunez_web/live/artists/show_live.ex

```
<.header class="pl-4 pr-2 !m-0">
  <% # ... %>
  <:action :if={Tunez.Music.can_destroy_album?(@current_user, @album)}>
    <.button_link [opts]]>Delete</.button_link>
  </:action>
  <:action :if={Tunez.Music.can_update_album?(@current_user, @album)}>
    <.button_link [opts]>Edit</.button_link>
  </:action>
</.header>
```

For these to work, you also need to update the call to the album_details function component, to pass the current_user in:

06/lib/tunez_web/live/artists/show_live.ex

```
<ul class="mt-10 space-y-6 md:space-y-10">
  <li :for={album <- @artist.albums}>
    <.album_details album={album} current_user={@current_user} />
  </li>
</ul>
```

Whew, that was fiddly. Moving forward, we'll wrap everything in authorization checks as we write it in our templates, so we don't have to do this kind of rework again.

Beware the policy check that performs queries!

Some policy checks can require database queries to figure out if an action is authorized or not. They might reference the actor's group membership, or a count of associated records, or data other than what you've loaded for the page to render.

For a LiveView app, if that related data isn't pre-loaded and stored in memory, it'll be re-fetched to recalculate the authorization on every page render, which would be *disastrous* for performance!

Beware the policy check that performs queries!

Ash make a best guess about authorization using data already loaded, if you use `run_queries?: false`[17] when calling `Ash.can?/can_*?`. If a decision can't be made definitively, Ash will use the value of the `maybe_is` option — by default this is `true`, but you can fail closed by setting it to `false`.

```
# Authorize based on data in memory, defaulting to unauthorized
Tunez.Music.can_run_complicated_action?(@current_user,
  run_queries?: false, maybe_is: false)
```

Everything is now in place for artist authorization — you should be able to log in and out of your Tunez dev app as users with different roles, and the app should behave as expected around managing artist data. We've also added authorization checks around album management in our templates — but don't have any policies to go with them. We'll add those now.

Writing Policies for Albums

The rules we want to implement for album management are a little different to those for artist management. Our rules for artists could be summarized as:

- Everyone can read all artist data
- Editors can update (but not create or delete) artists
- Admins can perform any action on artist data

For albums, our rules for reading and admins will be the same, but rules for editors will be different — they can create album records, or update/delete album records *that they created*.

It's only a small change, but a common use case. In a issue tracker/help desk app, users might be assigned as owners of tickets and thus have extra permissions for those tickets. Or a user might be recorded as the owner of an organization, and have permissions to invite members to the organization.

The key piece of information we need that we're not currently storing, is *who* is creating each album in Tunez. Once we know that, we can write the policies that we want.

17. https://hexdocs.pm/ash/Ash.html#can?/3

Recording who created and last modified a resource

To meet our requirements, we only need to record who created each album. But while we're here, we'll implement recording who created *and* last modified records, for both artists and albums.

To record this data for albums, we'll add two new relationships to the Tunez.Music.Album resource, both pointing at the Tunez.Accounts.User resource — one named created_by, and one named updated_by.

06/lib/tunez/music/album.ex

```
relationships do
  # ...
➤  belongs_to :created_by, Tunez.Accounts.User
➤  belongs_to :updated_by, Tunez.Accounts.User
end
```

You can add the exact same thing to the Tunez.Music.Artist resource.

Adding these relationships means an update to the database structure, so we need to generate a migration for the changes, and run it:

```
$ mix ash.codegen add_user_links_to_artists_and_albums
$ mix ash.migrate
```

We're now identifying the actor, on page 144 every time we submit a form to create or modify data, so we can use the built-in relate_actor[18] change to our actions, to store that actor information in our new relationships.

We'll do this a bit differently than previous changes like UpdatePreviousNames, though. Storing the actor isn't really related to the business logic of what we want the action to do, it's more of a side effect. We *really* want to implement something like "by the way, whenever you create or update a record, can you also store who made the change? Cheers." So the logic shouldn't be restricted to only the actions *named* :create and :update, it should apply to *all* actions of *type* create and update.

We can do this with a resource-level changes block. Like validations, changes can be added either to individual actions or to the resource as a whole. In a resource-level changes block, we can also choose one or more action types that the change should apply to, using the on option.[19]

In the Tunez.Music.Album resource, add a new top-level changes block and add the changes we want to store.

18. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html#relate_actor/2
19. https://hexdocs.pm/ash/dsl-ash-resource.html#changes-change-on

06/lib/tunez/music/album.ex

```
defmodule Tunez.Music.Album do
  # ...

  changes do
    change relate_actor(:created_by, allow_nil?: true), on: [:create]
    change relate_actor(:updated_by, allow_nil?: true)
  end
end
```

And repeat the process for the Tunez.Music.Artist resource.

These changes mean that if we add more create or update actions to the Artist or Album resources in future, they'll automatically have created_by and updated_by tracked — and we won't have to do a thing!

Why allow_nil?: true?

So that if you want to run or re-run the seed data scripts we've provided with Tunez, they'll successfully run both before and after adding these changes!

Depending on your app, you may also want to have nil values representing some kind of "system" action, if data may be created or updated by means other than a user specifically submitting a form.

There's one other change we need to make — as we discovered earlier, the User resource is really locked-down, permission-wise. To relate the actor to a record with relate_actor, we need to be able to *read* the actor from the database, and at the moment we can't. All reading of user data is forbidden, unless being called internally by AshAuthentication.

To solve this issue, we'll add another policy to the Tunez.Accounts.User resource, to allow a user to read their *own* record:

06/lib/tunez/accounts/user.ex

```
policies do
  # ...
➤ policy action(:read) do
➤   authorize_if expr(id == ^actor(:id))
➤ end
end
```

This uses the ^actor expression template[20] to reference the actor calling the action, as part of the policy's check condition. Like pinning a variable in a match or an Ecto query, this is how we can reference outside data that isn't

20. https://hexdocs.pm/ash/expressions.html#templates

a literal value (like true or :admin) and isn't an attribute or calculation on the resource (like created_by_id).

And that's all we need to do! Now whenever we call create or update (or their code interfaces) on either an artist or an album, the user ID of the actor will be stored in the created_by_id and/or the updated_by_id fields of the resource. You can test it out in iex, to make sure you've connected the pieces properly.

```
iex(1)> user = Tunez.Accounts.get_user_by_id!(«uuid», authorize?: false)
#Tunez.Accounts.User<id: «uuid», role: :admin, ...>
iex(2)> Tunez.Music.create_artist(%{name: "Who Made Me?"}, actor: user)
{:ok,
 #Tunez.Music.Artist<
   name: "Who Made Me?",
   updated_by: #Tunez.Accounts.User<id: «uuid», ...>,
   created_by: #Tunez.Accounts.User<id: «uuid», ...>,
   ...
 >}
```

Filling out policies

All of the pre-requisite work has been done, the only thing left to do is write the actual policies for album management. As with artists, the first step is enabling Ash.Policy.Authorizer in the Tunez.Music.Album resource:

06/lib/tunez/music/album.ex

```
defmodule Tunez.Music.Album do
  use Ash.Resource,
    otp_app: :tunez,
    domain: Tunez.Music,
    data_layer: AshPostgres.DataLayer,
    extensions: [AshGraphql.Resource, AshJsonApi.Resource],
➤   authorizers: [Ash.Policy.Authorizer]
```

All of our action calls, including auto-loading albums when loading an artist record (which uses a read action!), will now automatically run authorization checks. Because we haven't yet defined any policies, they will all be forbidden by default.

We can reuse some of the policies that we wrote for artists, as the bulk of the rules are the same. In a new policies block in the Tunez.Music.Album resource, write a bypass for users with the role :admin, as they're allowed to run every action. As this will be the first policy in the policies block, if it passes, all other policies will be skipped.

06/lib/tunez/music/album.ex

```
defmodule Tunez.Music.Album do
  # ...
```

```
  policies do
    bypass actor_attribute_equals(:role, :admin) do
      authorize_if always()
    end
  end
end
```

We'll also add an allow-all rule for *reading* album data — any user, authenticated or not, should be able to see the full list of albums for any artist.

06/lib/tunez/music/album.ex

```
policies do
  # ...

  policy action_type(:read) do
    authorize_if always()
  end
end
```

The main rules we want to look at are for editors. They'll have limited functionality — we want them to be able to create albums, and update/delete albums *if* they are related to those records via the created_by relationship.

The policy for create actions is pretty straightforward, it will use the same actor_attribute_equals built-in policy check we've used a few times now:

06/lib/tunez/music/album.ex

```
policies do
  # ...

  policy action(:create) do
    authorize_if actor_attribute_equals(:role, :editor)
  end
end
```

If the actor calling the create action doesn't have the role :admin (which would be authorized by the bypass) or :editor (which would be authorized by this create policy), then the action will be forbidden.

Lastly, we'll write one policy that covers both the update and destroy actions, as the rules are identical for both. If we only wanted to verify the created_by relationship link, we could use the built-in relates_to_actor_via policy check,[21] like this:

```
policy action([:update, :destroy]) do
  authorize_if relates_to_actor_via(:created_by)
end
```

21. https://hexdocs.pm/ash/Ash.Policy.Check.Builtins.html#relates_to_actor_via/2

We *could* still technically use this! As only editors can create albums (ignoring the admin bypass), then if the album was created by the actor, the actor *must* be an editor! Right?? But… rules can change. Maybe in some months, a new :creator role will be added that can *only* create records. But by the checks in this policy, they would also be authorized to update and destroy records, if they created them. Not good. Let's make the editor check in this policy explicit.

We can't combine built-in policy checks, so we'll have to fall back to writing an expression, like expr(published == true), to verify both conditions in the same policy check. We end up with a policy like the following:

06/lib/tunez/music/album.ex

```
policies do
  # ...

  policy action_type([:update, :destroy]) do
    authorize_if expr(^actor(:role) == :editor and created_by_id == ^actor(:id))
  end
end
```

It's a little verbose, but it clearly captures our requirements — updates and deletes should be authorized if the actor's role is :editor, and the record was created by the actor.

Test it out in your app! Register a new user, make it an editor with Tunez.Accounts.set_user_role/2, and see how it behaves! As we already edited all of the templates to add authorization checks, we don't need to make any other changes. Note that your editor doesn't have access to edit any existing albums, but if they create a new one, they can then edit *that* one. Perfect!

All of our authorization policies have also automatically flowed through to our APIs. Trying to create albums or artists when not being authenticated will now be forbidden, but when an authentication token for a valid editor or admin is provided in the request headers, creating an album will succeed. And we didn't need to do anything for that! We defined our policies once, in a central place, and they apply everywhere.

All this manual testing is getting a bit tiresome, though. We're starting to get more complicated logic in our app, and we can't keep manually testing everything. In the next chapter, we'll dive into testing — what to test, how to test it, and how Ash can help you get the best bang for your testing buck!

CHAPTER 7

All About Testing

While working on Tunez, we've been doing lots of manual testing of our code. We've called functions in iex and verified the results, and loaded the web app in a browser to click around. This is fine while we figure things out, but won't scale as our app grows. For that, we can look at automated testing.

There are two main reasons to write automated tests:

- *To confirm our current understanding of our code.* When we write tests, we're asserting that our code behaves in a certain way. This is what we've been doing so far.
- *To protect against unintentional change.* When we make changes to our code, it's critical to understand the impact of those changes. The tests now serve as a safety net to prevent regressions in functionality or bugs being introduced.

A common misconception about testing Ash applications is that you don't need to write as many tests as you would if you had handwritten all of the features that Ash provides for you. This isn't the case: it's important to confirm our understanding and to protect against unintentional change when building with Ash. Just because it's much easier to build our apps, doesn't mitigate the necessity for testing.

In this chapter, we won't cover how to use ExUnit[1] to write unit tests in Elixir. There are entire books written on testing, such as *Testing Elixir [LM21]*. For LiveView-specific advice, there's also a great section in *Programming Phoenix LiveView [TD24]*, and libraries like PhoenixTest[2] to make it smoother. What we *will* focus on is as follows:

1. https://hexdocs.pm/ex_unit
2. https://hexdocs.pm/phoenix_test/

- How to set up and execute tests against Ash resources
- What helpers Ash provides to assist in testing
- What kinds of things you should test in applications built with Ash

There's no code for you to write in this chapter — Tunez comes with a full set of tests pre-prepared, but they're all skipped and commented out (to prevent compilation failures). As we go through this chapter, you can check them out, and un-skip and un-comment the tests that cover features we've written so far.

For the remaining chapters in this book, we'll point out the tests that cover the functionality we're going to build.

What Should We Test?

"What do we test?" is a question that Ash can help answer. Ultimately, every interface in Ash stems from our action definitions. This means that the vast majority of our testing should center around calling our actions, making assertions about the behavior and effects of those actions. We should still write tests for our API interfaces, but they don't necessarily need to be comprehensive. One caveat to this is that if you're developing a public API, you may want to be more rigorous in your testing. We'll cover this in more detail shortly.

Additionally, Ash comes with tools and patterns that allow you to unit test various elements of your resource. Since an example is worth a thousand words, let's use some of these tools.

The basic first test

One of the best first tests to write for a resource is the empty read case — when there is no stored data, nothing is returned. This test may be kind of obvious, but it can detect problems in your test setup, such as leftover data that isn't being deleted between tests. It can also help identify when something is broken about your action that has nothing to do with the data in your data layer.

```
defmodule Tunez.Music.ArtistTest do
  use Tunez.DataCase, async: true

  describe "Tunez.Music.read_artists!/0-2" do
    test "when there is no data, nothing is returned" do
      assert Tunez.Music.read_artists!() == []
    end
  end
end
```

We can call the code interface functions defined for our actions and directly assert on the result. Provide inputs, verify outputs. It sounds so simple, when written like that!

While our code interfaces are on the Tunez.Music domain module, and not the Tunez.Music.Artist resource module, it would make for a *very* long and hard-to-navigate test file to include all the tests for the domain in one test module.

It's generally better to split up tests into smaller groups. Here we're testing actions on the Tunez.Music.Artist resource, so we have one module just for those. This isn't a requirement, but it leads to better test organization.

For more complicated actions (i.e. nearly all of them), we'll need a way of setting up the data and state required.

Setting Up Data

For artist actions like search or update, we'll need some records to exist in the data layer before we can run our actions and check the results. There are two approaches for this:

- Setting up test data using your resource actions
- Seeding data directly via the data layer, bypassing actions

Using actions to set up test data

The first approach is to do what we've already been doing all throughout this book: calling resource actions. These tests can be seen as a *series of events*.

```
# Demonstration test only
# There are tests for this action in Tunez, but not written like this!
defmodule Tunez.Music.ArtistTest do
  # ...

  describe "Tunez.Music.search_artists!/1-3" do
    test "can find artists by partial name match" do
      artist = Tunez.Music.create_artist!(%{
        name: "The Froody Dudes",
        biography: "42 musicians all playing the same instrument (a towel)"
      }, authorize?: false)

      assert [match] = Tunez.Music.search_artists!("Frood")
      assert match.id == artist.id
    end
  end
end
```

First we create an artist, and then we assert that we get that same artist back when we search for it. When in doubt, start with these kinds of tests.

We're testing our application's behaviour in the same way that it actually gets *used*. And because we're building with Ash, and our APIs and web UIs go through the same actions, we don't need to write extensive tests covering each different interface — we can test the action thoroughly, and then write simpler smoke tests for each of the interfaces that uses it.

(Writing out action calls with full data can be tedious and prone to breakage, though. We'll cover ways of addressing this in Consolidating Test Setup Logic, on page 163.)

Pro: We are testing real sequences of events

If something changes in the way that our users create artists that affects whether or not they show up in the search results, our test will reflect that. This is more akin to testing a "user story" than a unit test (albeit a very small user story).

This *can* also be a con: if something breaks in the Tunez.Music.Artist create action, every test that creates artist records as part of their setup will suddenly start failing. If this happens, though, all tests that *aren't* specifically for that action should point directly to it as the cause.

Con: Real application code has rules and dependencies

Let's imagine that we have a new app requirement that new artists could only be created on Tuesdays. If we wrote a custom validation module named IsTuesday and called it in the Artist create action, suddenly our test suite would only pass on Tuesdays!

There are ways around this, such as using a private argument to determine whether to run the validation or not. This can then be specifically disabled in tests, by passing in the option private_arguments: %{validate_tuesday: false} when building a changeset or calling a code interface function.

```
create :create do
  argument :validate_tuesday, :boolean, default: true, public?: false
  validate IsTuesday, where: argument_equals(:validate_tuesday, true)
end
```

You could also introduce a test double in the form of a mock with an explicit contract,[3] with different implementations based on the environment. This is also commonly used for replacing external dependencies in either dev or test.

3. https://dashbit.co/blog/mocks-and-explicit-contracts

We've already used an example of this with the Swoosh mailer, in Why do users always forget their passwords!?, on page 118. In production, it will send real emails (if we connected a suitable adapter)[4] but in dev/test it uses an in-memory adapter instead.

If all else fails, you can fall back to a library like mimic,[5] that performs more traditional mocking (mocking-as-a-verb).

Pro: Your application is end-to-end testable

If you have the time and resources to go through the above steps to ensure that actions with complex validations or external dependencies are testable, then this strategy is the best approach. Our tests are all doing only real, valid action calls, and we can have much more confidence in them.

With all of that said, there are still cases where we would want to set up our tests by working directly against the data layer.

Seeding data

The other method of setting up our tests is to use *seeds*. Seeds bypass action logic, going straight to the data layer. When using AshPostgres, this essentially means performing an INSERT statement directly. The only thing that Ash can validate when using seeds are attribute types, and the allow_nil? option, because they're implemented at the database level. If you've used libraries like ex_machina,[6] this is the strategy they use.

When should you reach for seeds to set up test data, instead of calling resource actions? Imagine that we've realized that a lot of Tunez users are creating artists with incomplete biographies, like just the word "Hi". To fix this, we've decided that all biographies must have at least 3 sentences.

So we write another custom validation module called SentenceCount, and add it the validations block of our Artist resource like validate {SentenceCount, field: :biography, min: 3}, so it applies to all actions. Ship it! Oops, we've just introduced a subtle bug. Can you spot it?

In this hypothetical, when a user tries to update the name of an artist that has a too-short biography saved, they'll get an error about the biography. That's not a great user experience. Luckily, it's an easy fix. We can tweak the validation to only apply when the biography is being updated:

4. https://hexdocs.pm/swoosh/Swoosh.html#module-adapters
5. https://hexdocs.pm/mimic/
6. https://hexdocs.pm/ex_machina/

```
validations do
  validate {SentenceCount, field: :biography, min: 3} do
➤   where changing(:biography)
  end
end
```

To write a test for this fix, we need a record with a short biography in the database to make sure the validation doesn't get triggered if it's not being changed. We don't want to add a new action just to allow for the creation of *bad data*. This is a perfect case for inserting data directly into the data layer using seeds.

In this example, we use Ash.Seed to create an artist that would not normally be allowed to be created.

```
# Demonstration test only - this validation doesn't exist in Tunez!
describe "Tunez.Music.update_artist!/1-3" do
  test "when an artist's name is updated, the biography length does
        not cause a validation error" do
    artist =
      Ash.Seed.seed!(
        %Tunez.Music.Artist{
          name: "The Froody Dudes",
          biography: "42 musicians all playing the same instrument (a towel)."
        }
      )

    updated_artist = Tunez.Music.update_artist!(artist, %{name: "New Name"})
    assert updated_artist.name == "New Name"
  end
end
```

Pro: Your tests are faster and simpler

Ash.Seed goes directly to the data layer, so any action logic, policies, or notifiers will be skipped. It can be easier to reason about what your test setup actually does. You can think more simply in terms of the data you need, and not the steps required to create it. If a call to Ash.Seed.seed! succeeds, you know you've written exactly that data to the data layer.

For the same reason, this will always be at least a *little* faster than calling actions to create data. For actions that do a lot of validation or contain hooks to call other actions, using seeds can be *much* faster.

Con: Your tests are not as realistic

While writing test setup using real actions makes setup more complicated, it also makes them more *valuable* and more *correct*. When testing with seed data, it's easy to accidentally create data that has no value to test against

because it's not possible to create under normal app execution. In Tunez, we could seed artists that were created by users with role :user or :editor, which definitely violates our authorization rules. Or we could set a user role that doesn't even *exist*! (This has actually happened.) What is testing the validity of the test data?

Depending on the situation, this can be worse than just wasted code: It can mislead you into believing that you've tested a part of your application that you haven't. It can also be difficult to know when you've changed something in your actions that *should* be reflected in your tests because your test setup bypasses actions.

How do I choose between seeds and calling actions?

When both will do what you need, consider what you're trying to test. Are you testing a *data condition*, such as the validation example, or are you testing an *event*, such as running a search query? If the former, then use seeds. If the latter, use your resource actions. When in doubt, use actions.

Consolidating Test Setup Logic

Ash.Generator[7] provides tools for dynamically generating various kinds of data. You can generate action inputs, queries, and even complete resource records, without having to specify values for every single attribute. We can use Ash.Generator to clean up our test setup, and to clearly distinguish our setup code from our test code.

The core functionality of Ash.Generator is built using the StreamData[8] library, and the generator/1 callback on Ash.Type. You can test out any of Ash's built-in types,[9] using Ash.Type.generator/2:

```
iex(1)> Ash.Type.generator(:integer, min: 1, max: 100)
#StreamData<66.1229758/2 in StreamData.integer/1>
iex(2)> Ash.Type.generator(:integer, min: 1, max: 100) |> Enum.take(10)
[21, 79, 33, 16, 15, 95, 53, 27, 69, 31]
```

The generator returns an instance of StreamData, which is a lazily-evaluated stream[10] of random data that matches the type and constraints specified. To get generated data *out* of the stream, we can evaluate it using functions from the Enum module.

7. https://hexdocs.pm/ash/Ash.Generator.html
8. https://elixir-lang.org/blog/2017/10/31/stream-data-property-based-testing-and-data-generation-for-elixir/
9. https://hexdocs.pm/ash/Ash.Type.html
10. https://hexdocs.pm/elixir/Stream.html

Ash.Generator also works for more complex types, such as maps with a set format:

```
iex(1)> Ash.Type.generator(:map, fields: [
          hello: [
            type: {:array, :integer},
            constraints: [min_length: 2, items: [min: -1000, max: 1000]]
          ],
          world: [type: :uuid]
        ]) |> Enum.take(1)
[%{hello: [-98, 290], world: "2368cc8d-c5b6-46d8-97ab-1fe1d9e5178c"}]
```

Ash.Generator.action_input/3 can be used to generate sets of valid inputs for actions, and Ash.Generator.changeset_generator/3 builds on top of that to generate whole changesets for calling actions. That sounds like an idea…

Creating test data using Ash.Generator

We can use the tools provided by Ash.Generator to build a Tunez.Generator module for test data. Using changeset_generator/3,[11] we can write functions that generate streams of changesets for a specific action, which can then be modified further if necessary or submitted to insert the records into the data layer.

Let's start with a user generator. To create different types of users, we would need to create changesets for the register_with_password action of the Tunez.Accounts.User resource, submit them, and then maybe update their roles afterwards with Tunez.Accounts.set_user_role. We can follow a very similar pattern using options for changeset_generator/3.

The key point to keep in mind is that our custom generators should *always* return a stream: The test calling the generator should always be able to decide if it needs one record or one hundred.

```
defmodule Tunez.Generator do
  use Ash.Generator

  def user(opts \\ []) do
    changeset_generator(
      Tunez.Accounts.User,
      :register_with_password,
      defaults: [
        # Generates unique values using an auto-incrementing sequence
        # eg. `user1@example.com`, `user2@example.com`, etc.
        email: sequence(:user_email, &"user#{&1}@example.com"),
        password: "password",
        password_confirmation: "password"
      ],
      overrides: opts,
```

11. https://hexdocs.pm/ash/Ash.Generator.html#changeset_generator/3

```
      after_action: fn user ->
        role = opts[:role] || :user
        Tunez.Accounts.set_user_role!(user, role, authorize?: false)
      end
    )
  end
end
```

To use our shiny new generator in a test, the test module can import Tunez.Generator and then we can use the provided generate[12] or generate_many functions:

```
# Demonstration test - this is only to show how to call generators!
defmodule Tunez.Accounts.UserTest do
  import Tunez.Generator

  test "can create user records" do
    # Generate a user with all default data
    user = generate(user())

    # Or generate more than one user, with some specific data
    two_admins = generate_many(user(role: :admin), 2)
  end
end
```

As we're forwarding the generator's opts directly to changeset_generator/3 as overrides for the default data, we could also include a specific email address or password, if we wanted. The generate functions use Ash.create! to process the changeset, so if something goes wrong, we'll know immediately. This is pretty clean!

We can write a generator for artists similarly. Creating an artist needs some additional data to exist in the data layer: an actor to create the record. We can pass an actor in via opts, or we can call our user generator within the artist generator.

One pitfall of calling the user generator directly is that we would get a user created for each artist we create. That *might* be what you want, but most of the time, it's unnecessary. To solve this, Ash.Generator provides the once/2 helper function: it will call the supplied function (in which we can generate a user) exactly once, and then re-use the value for subsequent calls in the same generator.

```
def artist(opts \\ []) do
  actor = opts[:actor] || once(:default_actor, fn ->
    generate(user(role: :admin))
  end)
```

12. https://hexdocs.pm/ash/Ash.Generator.html#generate/1

```
  changeset_generator(
    Tunez.Music.Artist,
    :create,
    defaults: [name: sequence(:artist_name, &"Artist #{&1}")],
    actor: actor,
    overrides: opts
  )
end
```

If we don't pass in an actor when generating artists, even if we generate a million artists, they'll all have the same actor. Efficient!

Now we can tie it all together to create an `album` factory. We can follow the same patterns as before, accepting options to allow customizing the generator, and massaging the generated inputs to be acceptable by the action.

```
def album(opts \\ []) do
  actor = opts[:actor] || once(:default_actor, fn ->
    generate(user(role: opts[:actor_role] || :editor))
  end)

  artist_id = opts[:artist_id] || once(:default_artist_id, fn ->
    generate(artist()).id
  end)

  changeset_generator(
    Tunez.Music.Album,
    :create,
    defaults: [
      name: sequence(:album_name, &"Album #{&1}"),
      year_released: StreamData.integer(1951..2024),
      artist_id: artist_id,
      cover_image_url: nil
    ],
    overrides: opts,
    actor: actor
  )
end
```

If we need to seed data instead of using changesets with actions, `Ash.Generator` also provides `seed_generator/2`.[13] This can be used in a very similar way, except instead of providing a resource/action, you provide a resource struct:

```
def seeded_artist(opts \\ []) do
  actor = opts[:actor] || once(:default_actor, fn ->
    generate(user(role: :admin))
  end)

  seed_generator(
```

13. https://hexdocs.pm/ash/Ash.Generator.html#seed_generator/2

```
    %Tunez.Music.Artist{name: sequence(:artist_name, &"Artist #{&1}")},
    actor: actor,
    overrides: opts
  )
end
```

This is a drop-in replacement for the artist generator, so you can still call functions like generate_many(seeded_artist(), 3). You could even put both seed and changeset generators in the same function, and switch between them based on an input option. It's a flexible pattern that allows you to generate exactly the data you need, in an explicit yet succinct way, and with the most confidence that what you're generating is *real*.

Armed with our generator, we're ready to start writing more tests!

Testing Resources

As we discussed earlier, the interfaces to our app all stem from our resource definitions. The code interfaces we define are the only thing external sources know about our app and how it works, so it makes sense that most of our tests will revolve around calling actions and verifying what they do. We've already seen a brief example when we wrote our first empty-case test on page 158, and now we'll write some more.

Testing actions

Our tests will follow a few guidelines:

- Prefer to use code interfaces when calling actions
- Use the raising "bang" versions of code interfaces in tests
- Avoid using pattern matching to assert success or failure of actions
- For asserting errors, use Ash.Test.assert_has_error or assert_raise
- Test policies, calculations, aggregates and relationships, changesets, and queries separately if necessary

The reasons for using code interfaces in tests are the same as in our application code, and they'll help us detect when changes to our resources require changes in our tests. Using the bang versions of functions that support it will keep our tests simple and give us better error messages when something goes wrong. Avoiding pattern matching helps with error messages and also increases the readability of our tests.

Some of the more interesting actions we might want to test are the Artist search action (including filtering and sorting), and the Artist update action (for storing

previous names and recording who made the change). What might those look like with our new generators?

```
# This can also be added to the `using` block in `Tunez.DataCase`
use Tunez.Generator

describe "Tunez.Music.search_artists/1-2" do
  defp names(page), do: Enum.map(page.results, & &1.name)

  test "can filter by partial name matches" do
    ["hello", "goodbye", "what?"]
    |> Enum.each(&generate(artist(name: &1)))

    assert Enum.sort(names(Music.search_artists!("o"))) == ["goodbye", "hello"]
    assert names(Music.search_artists!("oo")) == ["goodbye"]
    assert names(Music.search_artists!("he")) == ["hello"]
  end
```

The test uses the generators we just wrote, so we're assured that we're looking at real (albeit trivial) data. What about something a bit more complex, like testing one of the aggregate sorts we added?

```
test "can sort by number of album releases" do
  generate(artist(name: "two", album_count: 2))
  generate(artist(name: "none"))
  generate(artist(name: "one", album_count: 1))
  generate(artist(name: "three", album_count: 3))

  actual =
    names(Music.search_artists!("", query: [sort_input: "-album_count"]))

  assert actual == ["three", "two", "one", "none"]
end
```

The artist generator we wrote doesn't currently have an album_count option (it won't raise an error, but it won't do anything). For something like this, though, that feels like common behaviour, we can always add one. We can add an after_action to the call to changeset_generator to generate the number of albums we want for the artist.

```
def artist(opts \\ []) do
  # ...

  after_action =
    if opts[:album_count] do
      fn artist ->
        generate_many(album(artist_id: artist.id), opts[:album_count])
        Ash.load!(artist, :albums)
      end
    end

  # ...
  changeset_generator(
```

```
    Tunez.Music.Artist, :create,
    defaults: [name: sequence(:artist_name, &"Artist #{&1}")],
    actor: actor, overrides: opts,
➤   after_action: after_action
  )
end
```

We haven't specified any overrides for the albums to be generated. If you want to do that (e.g., specify that the albums were released in a specific year), we recommend not using this option and generating the albums separately in your test.

If your generators become complex enough, you may even want to write tests for them, to ensure that if we pass in something like album_count, the generated artist really has the related data that we expect.

Testing errors

Testing errors is a critical part of testing your application, but can also be kind of inconvenient. Actions can produce many different kinds of errors, and sometimes even multiple errors at once.

ExUnit comes with assert_raise[14] built-in for testing raised errors, and Ash also provides a helper function named Ash.Test.assert_has_error.[15] assert_raise is good for quick testing to say "when I do X, it fails for Y reason", while assert_has_error allows for more granular verification of the generated error.

The most common errors in Tunez right now are data validation errors, and we can write tests for those:

```
test "year_released must be between 1950 and next year" do
  admin = generate(user(role: :admin))
  artist = generate(artist())

  # The assertion isn't really needed here, but we want to signal to
  # our future selves that this is part of the test, not the setup.
  assert %{artist_id: artist.id, name: "test 2024", year_released: 2024}
         |> Music.create_album!(actor: admin)

  # Using `assert_raise`
  assert_raise Ash.Error.Invalid, ~r/must be between 1950 and next year/, fn ->
    %{artist_id: artist.id, name: "test 1925", year_released: 1925}
    |> Music.create_album!(actor: admin)
  end

  # Using `assert_has_error` - note the lack of bang to return the error
  %{artist_id: artist.id, name: "test 1950", year_released: 1950}
```

14. https://hexdocs.pm/ex_unit/ExUnit.Assertions.html#assert_raise/2
15. https://hexdocs.pm/ash/Ash.Test.html

```
  |> Music.create_album(actor: admin)
  |> Ash.Test.assert_has_error(Ash.Error.Invalid, fn error ->
    match?(%{message: "must be between 1950 and next year"}, error)
  end)
end
```

There are a few more examples of validation testing in the Tunez.Music.AlbumTest module — including how to use Ash.Generator.action_input[16] to generate valid action inputs (according to the constraints defined.) Check them out!

Testing policies

If you test *anything* at all while building an app, test your *policies*. Policies typically define the most critical rules in your application, and should be tested *rigorously*.

We can use the same tools for testing policies as we did in our liveview templates for showing/hiding buttons and other content — 'Ash.can?', on page 147 and the helper functions generated for code interfaces, can_*?. These run the policy checks for the actions, and return a boolean. Can the supplied actor run the actions according to the policy checks, or not? For testing policies for create, update and destroy actions, these make for simple and expressive tests.

Note that we're using refute for the last three assertions in the test. These users *can't* create artists!

```
test "only admins can create artists" do
  admin = generate(user(role: :admin))
  assert Music.can_create_artist?(admin)

  editor = generate(user(role: :editor))
  refute Music.can_create_artist?(editor)

  user = generate(user())
  refute Music.can_create_artist?(user)

  refute Music.can_create_artist?(nil)
end
```

Testing policies for read actions looks a bit different. These policies typically result in *filters*, not yes/no answers, meaning that we can't test "can the user run this action?" The answer is usually "yes, but nothing is returned if they do." For these kind of tests, we can use the data option to test that a specific record can be read.

16. https://hexdocs.pm/ash/Ash.Generator.html#action_input/3

Let's say that we get a new requirement that users should be able to look up their own user records and admins should be able to look up *any* user record, by email address. This could be over an API or in the UI; for our purposes, it is not important (and the Ash code looks the same).

The Tunez.Accounts.User resource already has a get_by_email action, but it doesn't have any specific policies associated. We can add a new policy specifically for that action:

```
policy action(:get_by_email) do
  authorize_if expr(id == ^actor(:id))
  authorize_if actor_attribute_equals(:role, :admin)
end
```

To make the action more accessible, we'll add a code interface for the action, in the Tunez.Accounts domain module:

```
resource Tunez.Accounts.User do
  # ...

  define :get_user_by_email, action: :get_by_email, args: [:email]
end
```

Now we can test the interface with the auto-generated can_get_user_by_email? function. Using the data option tells Ash to check the authorization against the provided record or records. It's roughly equivalent to running the query with any authorization filters applied, and checking to see if the given record or records are returned in the results.

```
# Demonstration tests only - this functionality doesn't exist in Tunez!
test "users can only read themselves" do
  [actor, other] = generate_many(user(), 2)

  # this assertion would fail, because the actor *can* run the action
  # but it *wouldn't* return the other user record
  # refute Accounts.can_get_user_by_email?(actor, other.email)

  assert Accounts.can_get_user_by_email?(actor, actor.email, data: actor)
  refute Accounts.can_get_user_by_email?(actor, other.email, data: other)

end

test "admins can read all users" do
  [user1, user2] = generate_many(user(), 2)
  admin = generate(user(role: :admin))

  assert Accounts.can_get_user_by_email?(admin, user1.email, data: user1)
  assert Accounts.can_get_user_by_email?(admin, user2.email, data: user2)
end
```

You should test your policies until you're confident that you've fully covered all of their variations, and then add a few more tests just for good measure!

Testing relationships & aggregates

Ash doesn't provide any special tools to assist in testing relationships or aggregates because none are really needed. You can set up some data in your test, load the relationship or aggregate, and then assert something about the response.

We will, however, use this opportunity to show how you can use authorize?: false to test or bypass your policies for the purpose of testing. A lot of the time, you'll likely want to skip authorization checking when loading data, unless you're specifically testing your policies around that data.

```
# Demonstration test only - this functionality doesn't exist in Tunez
test "users cannot see who created an album" do
  user = generate(user())
  album = generate(album())

  # We *can* load the user record if we skip authorization
  assert Ash.load!(album, :created_by, authorize?: false).created_by

  # If this assertion fails, we know that it must be due to authorization
  assert Ash.load!(album, :created_by, actor: user).created_by
end
```

Testing calculations

Calculations often contain important application logic, so it can be important to test them. You *can* test them the same way you test relationships and aggregates — load them on a record and verify the results — but you can also test them in total isolation using Ash.calculate/3.[17]

To show this, we'll add a temporary calculation to the Tunez.Music.Artist resource that calculates the length of the artist's name using the string_length[18] function:

```
defmodule Tunez.Music.Artist do
  # ...

  calculations do
    calculate :name_length, :integer, expr(string_length(name))
  end
end
```

If we wanted to use this calculation "normally", we would have to construct or load an Artist record, and then load the data:

```
iex(1)> artist = %Tunez.Music.Artist{name: "Amazing!"} |>
                 Ash.load!(:name_length)
```

17. https://hexdocs.pm/ash/Ash.html#calculate/3
18. https://hexdocs.pm/ash/expressions.html#functions

```
#Tunez.Music.Artist<...>
iex(2)> artist.name_length
8
```

Using Ash.calculate/3, we can call the calculation directly, passing in a map of references, or *refs* — data that the calculation needs to be evaluated.

```
iex(30)> Ash.calculate!(Tunez.Music.Artist, :name_length,
                        refs: %{name: "Amazing!"})
8
```

The name_length calculation only relies on a name field, so the rest of the data of any Artist record doesn't matter. This makes it simpler to set up the data required.

This also works for calculations that require the database, such as those written using database fragments.[19] If we rewrite our name_length calculation using PostgreSQL's length function:

```
calculations do
  calculate :name_length, :integer, expr(fragment("length(?)", name))
end
```

We could still call it in iex or in a test, only needing to pass in the name ref:

```
iex(3)> Ash.calculate!(Tunez.Music.Artist, :name_length,
                       refs: %{name: "Amazing!"})
SELECT (length($1))::bigint FROM (VALUES(1)) AS f0 ["Amazing!"]
8
```

You can even define code interfaces *for calculations*. This combines the benefits of Ash.calculate/3, with the benefits of code interfaces.

We'll use define_calculation[20] to define a code interface for our trusty name_length calculation, in the Tunez.Music domain module. A major difference here is how we specify arguments for the code interface compared with defining code interfaces for actions. Because calculations can also accept arguments,[21] they need to be formatted slightly differently. Each of the code interface arguments should be in a tuple tagging it as a ref, or an arg. Our name is a ref, a data dependency of the calculation.

```
resource Tunez.Music.Artist do
  ...
  define_calculation :artist_name_length, calculation: :name_length,
    args: [{:ref, :name}]
```

19. https://hexdocs.pm/ash_postgres/expressions.html
20. https://hexdocs.pm/ash/dsl-ash-domain.html#resources-resource-define_calculation
21. https://hexdocs.pm/ash/calculations.html#arguments-in-calculations

```
end
```

This exposes the name_length calculation defined on the Tunez.Music.Artist resource, as an artist_name_length function on the domain module. If the calculation name and desired function name are the same, the calculation option can be left out.

```
# Demonstration test only - this function doesn't exist in Tunez!
test "name_length shows how many characters are in the name" do
  assert Tunez.Music.artist_name_length!("fred") == 4
  assert Tunez.Music.artist_name_length!("wat") == 3
end
```

Imagine we put a limit on the length of an artist's name, or some other content like a blog post. You could use this calculation to display the number of characters remaining next to the text box while the user is typing, without visiting the database. Then, if you changed the way you count characters in an artist's name, like perhaps ignoring the spaces between words, the logic will be reflected in your view in any API interface that uses that information and even in any *query* that uses the calculation.

Unit testing changesets, queries and other Ash modules

The last tip for testing Ash is that you can unit test directly against an Ash.Changeset, Ash.Query, or by calling functions directly on the Ash.Resource.Change and Ash.Resource.Query modules.

For example, if we want to test our validations for year_released, we don't necessarily need to go through the rigamarole of setting up test data and trying to call actions if we don't want to. We have a few other options.

We could directly build a changeset for our actions and assert that it has a given error. It doesn't matter that it also has other errors. We just care that it has one matching what we're testing.

```
# Demonstration test only - this is covered by action tests in Tunez
test "year_released must be greater than 1950" do
  Album
  |> Ash.Changeset.for_create(:create, %{year_released: 1920})
  |> assert_has_error(fn error ->
    match?(%{message: "must be between 1950 and" <> _}, error)
  end)
end
```

We can apply the same logic above to Ash.Query and Ash.ActionInput to unit test any piece of logic that Ash does eagerly as part of running an action. We can test directly against the modules that we define, as well. Let's write a test that calls into our artist UpdatePreviousNames change.

```
# Demonstration test only - this is covered by action tests in Tunez
# This won't work for logic in hook functions - only code in a change body
test "previous_names store the current name when changing to a new name" do
  changeset =
    %Artist{name: "george", previous_names: ["fred"]}
    |> Ash.Changeset.new()
    # `opts` and `context` aren't used by this change, so we can
    # leave them empty
    |> Tunez.Music.Changes.UpdatePreviousNames.change([], %{})

  assert Ash.Changeset.changing_attribute?(changeset, :previous_names)
  assert {:ok, ["george", "fred"]} = Ash.Changeset.fetch_change(changeset,
    :previous_names)
```

As you can see, there are numerous places where you can drill down for more specific unit testing as needed. This brings us to a *reeeeeally* big question….

Should I actually unit test every single one of these things?

Realistically? No.

Not every single variation of everything needs its own unit test. You can generally have a lot of confidence in your tests by calling your resource actions and making assertions about the results. If you have an action with a single `change` on it that does a little validation or data transformation, test the action directly. You've exercised all of the code, and you know your action works. That's what you really care about, anyway!

You only need to look at unit testing individual parts of your resource if they grow complex enough that you have trouble understanding them in isolation. If you find yourself wanting to write many different combinations of inputs to exercise one part of your action, perhaps that part should be tested in isolation.

Testing Interfaces

All of the tests we've looked at so far have centered around our resources. This is *the* most important type of testing, because it extends to every interface that uses our resources. If the number `5` is an invalid argument value when calling an action, that property will extend to any UI or API that we use to call that action. This doesn't mean, however, that we shouldn't test those higher layers.

What it *does* allow us to do is to be a bit less rigorous in testing these generated interfaces. If we've tested every action, validation, and policy at the Ash level, we only really need to test some basic interactions at the UI/API level to get the most bang for our buck.

Testing GraphQL

Since AshGraphql is built on top of the excellent absinthe library, we can use its great utilities[22] for testing. It offers three different approaches, for testing either resolvers, documents, or HTTP requests.

Ash actions take the place of resolvers, so any tests we write for our actions will cover that facet. Our general goal is to have several end-to-end HTTP request-response sanity tests to verify that the API as a whole is healthy and seperate schema-level tests for different endpoints. These will quickly surface errors if any types happen to accidentally change. We've written some examples of these tests in test/tunez_web/graphql/, so you can see what we mean.

We also highly recommend setting up your CI process (such as GitHub Actions) to guard against accidental changes to your API schema. This can be done by generating a known-good schema definition once with the absinthe.schema.sdl Mix task and committing it to your repository. During your build process, you can trun the task again into a separate file and compare the two files to ensure no breaking changes.

Testing AshJsonApi

Everything we said for testing a GraphQL API above applies to testing an API built with AshJsonApi as well. Since we generate an OpenAPI specification for your API, you can even use the same strategy for guarding against breaking changes.

The main difference when testing APIs built with AshJsonApi is that under the hood, they use Phoenix controllers, so we can use Phoenix helpers for controller tests. There are also some useful helpers in the AshJsonApi.Test module[23] that you can import to make your tests more streamlined. There are some examples of tests for our JSON API endpoints in Tunez, in lib/tunez_web/json_api/.

Testing Phoenix LiveView

Testing user interfaces is *entirely* different than anything else that we've discussed thus far. There are whole books dedicated to solely this topic. LiveView itself has many testing utilities, and often when testing LiveView, we're testing much more than the functionality of our application core.

22. https://hexdocs.pm/absinthe/testing.html
23. https://hexdocs.pm/ash_json_api/AshJsonApi.Test.html

It's unrealistic to cover all (or even most) of the UI testing patterns that exist here, for LiveView or otherwise. We've written a set of tests using our preferred PhoenixTest[24] library, in the test/tunez_web/live folder of Tunez. These include tests for the Artist and Album forms, and the Artist catalog, including the pagination, search and sort functionality.

This should help you get your feet wet, and the documentation for PhoenixTest and Phoenix.LiveViewTest[25] will take you the rest of the way.

And that's a wrap! This was a whirlwind tour through all kinds of testing that we might do in our application. There are a lot more tests available in the Tunez repo (along with some that cover functionality that we haven't built yet), far too many to go over in this chapter.

All of the tools that Ash works with, like Phoenix and Absinthe, have their own testing utilities and patterns that you'll want to spend some time learning as you go along. The primary takeaway is that you'll get the most reward for effort by doing your heavy and exhaustive testing at the resource layer.

Testing is a very important aspect of building any software, and that doesn't change when you're using Ash. Tests are investments that pay off by helping you *understand your code* and *protect against unintentional change* in the future.

In the next chapter, we'll switch back into writing some new features to enhance our domain model. We'll look at adding track listings for albums, adding calculations for track and album durations, and learn how AshPhoenix can help make building nested forms a breeze.

24. https://hexdocs.pm/phoenix_test/
25. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveViewTest.html

CHAPTER 8

Fun With Nested Forms

In the last chapter, we learnt all about how we can test the applications we build with Ash. The framework can do a lot for us, but at the end of the day, *we* own the code we write and the apps we build. With testing tools and know-how in our arsenal, we can be more confident that our apps will continue to behave as we expect.

Now we can get back to the fun stuff: more features! Knowing which artists released which albums is great, but albums don't exist in a vacuum — they have *tracks* on them. (You might even be listening to some tracks from your favorite album right now as you read this.) Let's build a resource to model a Track, and then learn how to manage them.

Setting Up a Track Resource

A track is a music-related resource, so we'll add it to the Tunez.Music domain using the ash.gen.resource Mix task:

```
$ mix ash.gen.resource Tunez.Music.Track --extend postgres
```

This will create a basic empty Track resource in lib/tunez/music/track.ex, as well as listing it as a resource in the Tunez.Music domain. What attributes should a track have? We're probably interested in:

- The order of tracks on the album
- The name of each track
- The duration of each track, which we'll store as a number of seconds
- And finally, we need to know which album the tracks belongs to

We'll also add an id and some timestamps, for informational reasons.

All of the fields will be required, so we can add them to the Tunez.Music.Track resource and mark them all as allow_nil? false:

08/lib/tunez/music/track.ex

```
defmodule Tunez.Music.Track do
  # ...

  attributes do
    uuid_primary_key :id

    attribute :order, :integer do
      allow_nil? false
    end

    attribute :name, :string do
      allow_nil? false
    end

    attribute :duration_seconds, :integer do
      allow_nil? false
      constraints min: 1
    end

    create_timestamp :inserted_at
    update_timestamp :updated_at
  end

  relationships do
    belongs_to :album, Tunez.Music.Album do
      allow_nil? false
    end
  end
end
```

The order field will be an integer, representing its place in the album's track list. The first track will have order 1, the second track order 2, and so on.

The relationship between tracks and albums can go both ways: an album can have many tracks, and that's how we'll work with them most of the time. We'll add that relationship to the Tunez.Music.Album resource:

08/lib/tunez/music/album.ex

```
relationships do
  # ...

➤ has_many :tracks, Tunez.Music.Track do
➤   sort order: :asc
➤ end
end
```

Like artists and their albums, we've specified a sort for the relationship, to always sort tracks on an album by their order attribute using the sort[1] option.

1. https://hexdocs.pm/ash/dsl-ash-resource.html#relationships-has_many-sort

Storing the track duration as a number instead of as a formatted string (eg. "3:32") might seem strange, but it will allow us to do some neat calculations. We can calculate the duration of a whole album by adding up the track durations, or the average track duration for an artist or album. We don't have to *show* the raw number to the user, but having it will be very useful.

Before generating a migration for this new resource, there's one other thing to add. Like we saw in chapter 2 on page 51, albums don't make sense without an associated artist, and neither do tracks without their album. If an album gets deleted, all of its tracks should be deleted too. To do this, we'll customize the reference[2] to the albums table, in the postgres block of the Tunez.Artist.Track resource. We'll add an index to the foreign key as well, with index? true.

08/lib/tunez/music/track.ex

```
postgres do
  # ...

  references do
    reference :album, index?: true, on_delete: :delete
  end
end
```

Now we can generate a migration to create the database table, and run it:

```
$ mix ash.codegen add_album_tracks
$ mix ash.migrate
```

Reading and writing track data

At the moment, the Tunez.Music.Track resource has no actions at all. So what do we need to add? Our end goal is something like the following:

	Name	Duration	
≡	Midnight Mew	4:05	🗑
≡	Black Cat Boogie	4:30	🗑
≡	Zombie Chase	3:50	🗑

On a form like this, we can edit all of the tracks of an album at once via the form for creating or updating an album. We won't be manually calling any actions on the Track resource to do this — Ash will handle it for us, once configured — but the actions still need to *exist*, for Ash to call.

2. https://hexdocs.pm/ash_postgres/dsl-ashpostgres-datalayer.html#postgres-references-reference

The actions we define will therefore be pretty similar to those we would define for any other resource. The fact that our primary interface for tracks will be via an album doesn't mean that we won't *also* be able to manage tracks on their own, but we won't build a UI to do so. So we'll add four actions, for our basic CRUD functionality:

08/lib/tunez/music/track.ex

```
defmodule Tunez.Music.Track do
  # ...

  actions do
    defaults [:read, :destroy]

    create :create do
      primary? true
      accept [:order, :name, :duration_seconds, :album_id]
    end

    update :update do
      primary? true
      accept [:order, :name, :duration_seconds]
    end
  end
end
```

These actions *do* need to be explicitly marked with primary? true. When Ash manages the records for us, it needs to know which actions to use. By default, Ash will look for primary actions of the type it needs, e.g. a primary action of type create to insert new data.

"Wait! Wait!" we hear you cry. "Didn't you say that users wouldn't have to deal with track durations as a number of seconds?" Yes, we did, but we'll add that feature *after* we get the basic form UI up and running.

Managing Relationships for Related Resources

We want to manage tracks via the form for managing an album, so a lot of the code we'll be writing will be in the TunezWeb.Albums.FormLive liveview module. There's a track_inputs/1 function component already defined in the liveview, for rendering a table of tracks for the album using Phoenix's standard inputs_for[3] component. This component will iterate over the data in @form[:tracks], and render a row of input fields for each item in the list.

Add the track_inputs/1 component to the form at the bottom of the main render/1 action, right above the Save button:

3. https://hexdocs.pm/phoenix_live_view/Phoenix.Component.html#inputs_for/1

08/lib/tunez_web/live/albums/form_live.ex

```
<% # ... %>
<.input field={form[:cover_image_url]} label="Cover Image URL" />

<.track_inputs form={form} />

<:actions>
  <% # ... %>
```

In a browser, if you now try to create or edit an album, you'll see an error telling you that you need to do a bit more configuration first:

```
tracks at path [] must be configured in the form to be used with
`inputs_for`. For example:

There is a relationship called `tracks` on the resource `Tunez.Music.Album`.

Perhaps you are missing an argument with `change manage_relationship` in
the action Tunez.Music.Album.update?
```

This is a pretty helpful error message, more so than it might first appear. Ash doesn't know what to do with our attempt to render inputs for an album's tracks. They're not something that the actions for the form, create and update on the Tunez.Music.Album resource, know how to process.

tracks isn't an attribute of the resource, so we can't add it to the accept list in the actions. They're a relationship! To handle tracks in an action, we need to add them as an *argument* to the action, as the error suggests, and then process them with the built-in manage_relationship change function.

Managing relationships with… err… manage_relationship

Using the manage_relationship[4] function is getting its own section because it's *so* flexible and powerful. Some even say that mastering it is the final boss of learning Ash. If you're looking to deal with relationship data in an action, it's likely going to be *some* invocation of manage_relationship, with varying options.

The full set of options is defined in the same-named function on Ash.Changeset[5] (be warned, there are a *lot* of options.) The most common option is the type option: this is a shortcut to different behaviours depending on the data provided. The two most common type values you'll see for forms in the wild are append_and_remove and direct_control.

Using type append_and_remove

append_and_remove is a way of saying "replace the existing links in this relationship with these new links, adding and removing records where necessary".

4. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html#manage_relationship/3
5. https://hexdocs.pm/ash/Ash.Changeset.html#manage_relationship/4

This typically works with IDs of existing records, either singular or as a list. A common example of using this is with tagging. If you provide a list of tag IDs as the argument, Ash can handle the rest.

append_and_remove can also be used for managing belongs_to relationships. In Tunez, we've allowed the foreign key relationships to be written directly, such as the artist_id attribute when creating an Album resource. The create action on Tunez.Music.Album could also be written as follows:

```
create :create do
  accept [:name, :year_released, :cover_image_url]

  argument :artist_id, :uuid, allow_nil?: false
  change manage_relationship(:artist_id, :artist, type: :append_and_remove)
end
```

This code will take the named argument (artist_id) and use it to update the named relationship (artist), using the append_and_remove strategy.

Writing the code using manage_relationship this way does have an extra benefit. Ash will verify that the provided artist_id belongs to a valid artist that the current user is *authorized to read*, before writing the record into the data layer. This could be pretty important! If you're building a form for users to join groups, for example, you wouldn't want a malicious user to edit the form, add the group ID of the secret_admin_group (if they know it), and then join that group!

Using type direct_control

direct_control maps more to what we want to do on our Album form: manage relationship data by editing all of the related records. As the name implies, it gives us direct control over the relationship and the full data of each of the records within it.

While append_and_remove focuses on managing the links between existing records, direct_control is about creating and destroying the related records themselves. If we edit an album and remove a track, that track shouldn't be unlinked from the album, it should be *deleted*.

Following the instructions from the error message we saw previously, we can add a tracks argument and a manage_relationship change to the create and update actions in the Tunez.Music.Album resource. We'll be submitting data for multiple tracks in a list, and each list item will be a map of attributes:

08/lib/tunez/music/album.ex

```
create :create do
  accept [:name, :year_released, :cover_image_url, :artist_id]
➤ argument :tracks, {:array, :map}
```

```
    change manage_relationship(:tracks, type: :direct_control)
  end

  update :update do
    accept [:name, :year_released, :cover_image_url]
    require_atomic? false
    argument :tracks, {:array, :map}
    change manage_relationship(:tracks, type: :direct_control)
  end
```

Because the name of the argument and the name of the relationship to be managed are the same (tracks), we can omit one when calling manage_relationship. Every little bit helps!

Another mention of atomics...

Like our implementation of previous names on page 53 for artists, we also need to mark this update action as require_atomic? false. Because Ash needs to figure out which related records to update, which to add, and which to delete when updating a record, calls to manage_relationship in update actions currently can't be converted into logic to be pushed into the data layer.

In the future, manage_relationship will be improved to support atomic updates for most of the option arrangements that you can provide, but for now it requires us to set require_atomic? false.

Trying to create or edit an album should now render the form without error. You should see an empty tracks table with a button to add a new track (that won't work yet, because haven't implemented it). Our two actions can now actually fully manage relationship data for tracks! To prove this, in iex, you can build some data in the shape that the album create action expects, with an existing artist_id, and then call the action:

```
iex(1)> tracks = [
          %{order: 1, name: "Test Track 1", duration_seconds: 120},
          %{order: 3, name: "Test Track 3", duration_seconds: 150},
          %{order: 2, name: "Test Track 2", duration_seconds: 55}
        ]
[...]
iex(2)> Tunez.Music.create_album!(%{name: "Test Album", artist_id: «uuid»,
        year_released: 2025, tracks: tracks}, authorize?: false)
«SQL queries to create the album and each of the tracks»
#Tunez.Music.Album<
  tracks: [
    #Tunez.Music.Track<order: 1, ...>
    #Tunez.Music.Track<order: 2, ...>,
    #Tunez.Music.Track<order: 3, ...>
  ],
```

```
  name: "Test Album", ...
>
```

Note that we don't have to provide the album_id for any of the maps of track data — we *can't*, because we're creating a new album and it doesn't have an ID yet. Ash takes care of that, creating the album record first, and then adding the new album ID to each of the tracks.

To make these tracks appear in the form when editing the album, we need to *load* them. Not loading the track data is basically the same as saying there are no tracks at all. We can update the mount/3 function in TunezWeb.Albums.FormLive when we load the album and artist, to also load the tracks for the album.

08/lib/tunez_web/live/albums/form_live.ex

```
def mount(%{"id" => album_id}, _session, socket) do
  album =
    Tunez.Music.get_album_by_id!(album_id,
      load: [:artist, :tracks],
      actor: socket.assigns.current_user
    )
  # ...
```

And voilà, the tracks will appear on the form! You can edit the existing tracks and save the album, and the data will be updated. All of the built-in validations from defining constraints and allow_nil? false on the track's attributes will be run. You won't be able to save tracks without a name, or with a duration less than 1 second.

Adding and removing tracks via the form

To make the form really usable, though, we need to be able to add new tracks and delete existing ones. The UI is already in place for it; the form has an "Add Track" button, and each row has a little trash can button to delete it. Currently, the buttons send the events "add-track" and "remove-track" to the FormLive liveview, but the event handlers don't do anything… yet.

Adding new rows for track data

AshPhoenix.Form provides helpers that we can use for adding and removing nested rows in our form, namely add_form[6] and remove_form.[7] In the "add-track" event handler, update the form reference stored in the socket and add a form at the specified *path*, or layer of nesting:

6. https://hexdocs.pm/ash_phoenix/AshPhoenix.Form.html#add_form/3
7. https://hexdocs.pm/ash_phoenix/AshPhoenix.Form.html#remove_form/3

08/lib/tunez_web/live/albums/form_live.ex

```
def handle_event("add-track", _params, socket) do
  socket =
    update(socket, :form, fn form ->
      AshPhoenix.Form.add_form(form, :tracks)
    end)

  {:noreply, socket}
end
```

If you're more familiar with the Phoenix method of adding form inputs using a hidden checkbox,[8] AshPhoenix supports that as well.[9] It's a little less obvious as to what's going on, though, which is why we'd generally opt for the more direct event handler way.

We can also auto-populate data in the added form rows, using the params option to add_form. For example, if we wanted to pre-populate the order when adding new tracks, we could use AshPhoenix.Form.value[10] to introspect the form and set the value:

```
update(socket, :form, fn form ->
  order = length(AshPhoenix.Form.value(form, :tracks) || []) + 1
  AshPhoenix.Form.add_form(form, :tracks, params: %{order: order})
end)
```

Removing existing rows of track data

Ooops, we pressed the "Add Track" button one too many times! Abort, abort!

We can implement the event handler for removing a track form in a similar way to adding a track form. The only real difference is we need to know *which* track to remove. The button for each row therefore has a phx-value-path attribute on it, to pass the name of the current form to the event handler as the path parameter:

08/lib/tunez_web/live/albums/form_live.ex

```
<.button_link phx-click="remove-track" phx-value-path={track_form.name}
  kind="error" size="xs" inverse>
  <span class="hidden">Delete</span>
  <.icon name="hero-trash" class="size-5" />
</.button_link>
```

8. https://hexdocs.pm/phoenix_live_view/Phoenix.Component.html#inputs_for/1-dynamically-adding-and-removing-inputs
9. https://hexdocs.pm/ash_phoenix/nested-forms.html#the-_add_-checkbox
10. https://hexdocs.pm/ash_phoenix/AshPhoenix.Form.html#value/2

This path will be form[tracks][2], if we clicked the delete button for the third track in the list (zero-indexed). That path can be passed directly to AshPhoenix.Form.remove_form to update the parent and delete the form at that path.

08/lib/tunez_web/live/albums/form_live.ex

```
def handle_event("remove-track", %{"path" => path}, socket) do
  socket =
    update(socket, :form, fn form ->
      AshPhoenix.Form.remove_form(form, path)
    end)

  {:noreply, socket}
end
```

AshPhoenix also supports the checkbox method for deleting forms,[11] as well.

And that's it for the basic usability of our track forms! AshPhoenix provides a really nice API for working with forms, making most of what we need to do in our views straightforward.

What about policies?!

If you spotted that we didn't write any policies for our new Track resource, gold star for you! (Gold star even if you didn't. You've earned it.)

Tunez is secure, authorization-wise, as it is right now, but there's no guarantee that it will stay that way. We're not currently running any actions manually for tracks, so they're inheriting policies from the context they're called in. That could change in the future though: we might add a form for managing individual tracks, and without specific policies on the Tunez.Music.Track resource, it would be wide open.

Let's codify a version of our implicit rule of tracks inheriting policies from their parent album, with an accessing_from[12] policy check:

08/lib/tunez/music/track.ex

```
defmodule Tunez.Music.Track do
  use Ash.Resource,
    otp_app: :tunez,
    domain: Tunez.Music,
    data_layer: AshPostgres.DataLayer,
    authorizers: [Ash.Policy.Authorizer]

  policies do
    policy always() do
      authorize_if accessing_from(Tunez.Music.Album, :tracks)
    end
```

11. https://hexdocs.pm/ash_phoenix/nested-forms.html#using-the-_drop_-checkbox
12. https://hexdocs.pm/ash/Ash.Policy.Check.Builtins.html#accessing_from/2

```
➤   end
```

This can be read as "if tracks are being read/created/updated/deleted through a :tracks relationship on the Tunez.Music.Album resource, then the request is authorized". Reading track lists via a load statement to show on the artist profile? A-OK. Ash will run authorization checks for all of the loaded resources — the artist, the albums, and the tracks — and if they all pass, the artist profile will be rendered.

Updating a single album with an included list of track data? Policies will be checked for both the album and the tracks, and the track policy will always pass in this scenario.

Fetching an individual track record in iex, via its ID? Nope, it wouldn't be allowed by this policy. Hmmm… that doesn't sound right. We'll fix that by adding another check in the policy:

08/lib/tunez/music/track.ex

```
policy always() do
  authorize_if accessing_from(Tunez.Music.Album, :tracks)
➤   authorize_if action_type(:read)
end
```

This looks different than the policies we wrote in chapter 6. Those policies used action_type in the policy *condition*, not in individual checks, but both ways will work. This could have been written as two separate policies:

```
policies do
  policy accessing_from(Tunez.Music.Album, :tracks) do
    authorize_if always()
  end

  policy action_type(:read) do
    authorize_if always()
  end
end
```

Our initial version is much more succinct, though, and more readable.

Testing these policies is a little trickier than those in our Artist/Album resources. We don't have code interfaces for the Track actions, and we have to test them *through* the album resource. This is a good candidate for using seeds to generate test data, to clearly separate creating the data from testing what we can do with it.

There are a few tests in the test/tunez/music/track_test.exs file to cover these new policies — you'll also need to uncomment the track() generator function in the Tunez.Generator module.

Reorder All of the Tracks!!!

Now that we can add tracks to an album, we can display nicely-formatted track lists for each album on the artist's profile page. Currently we have a "track data coming soon" placeholder display coming from the track_details function component in TunezWeb.Artists.ShowLive. This is because when the track_details function component is rendered at the bottom of the album_details function component, the provided tracks is a hardcoded empty list.

To put the real track data in there, first we need to load the tracks when we load album data, up in the handle_params/3 function. We already have :albums as a single item in the list of data to load, so to load tracks for each of the albums, we turn it into a keyword list:

08/lib/tunez_web/live/artists/show_live.ex

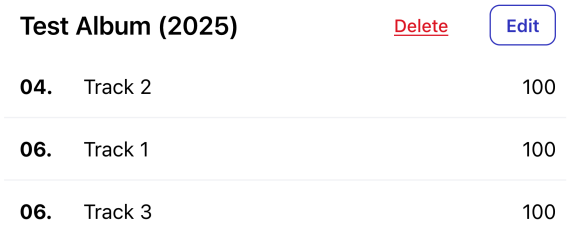
```
defmodule TunezWeb.Artists.ShowLive do
  # ...

  def handle_params(%{"id" => artist_id}, _url, socket) do
    artist =
      Tunez.Music.get_artist_by_id!(artist_id,
        load: [albums: [:tracks]],
        actor: socket.assigns.current_user
      )

    # ...
```

Because we've added a sort for the tracks relationship, we'll always get tracks in the correct order, ordered by order. Then we need to replace the hardcoded empty list in the album_details function component with a reference to the real tracks, loaded on the @album struct.

08/lib/tunez_web/live/artists/show_live.ex

```
    </.header>
    <.track_details tracks={@album.tracks} />
  </div>
</div>
```

Depending on the kinds of data you've been entering while testing, you might now see something like the following when looking at your test album:



This doesn't look great. We don't have any validations to make sure that the track numbers entered are a sequential list, with no duplicates, or anything! But do we *really* want to write validations for that, to put the onus on the user to enter the right numbers? It would be better if we could automatically order them, based on the data in the form. The first track in the list should be track 1, the second track track 2, etc. That way, there would be no chance of mistakes.

Automatic track numbering

This automatic numbering can be done with a tweak to our manage_relationship call, in the create and update actions in the Tunez.Music.Album resource. The order_is_key option[13] will do what we want: take the position of the record in the list, and set it as the value of the attribute we specify.

08/lib/tunez/music/album.ex

```
create :create do
  # ...
  change manage_relationship(:tracks, type: :direct_control,
    order_is_key: :order)
end

update :update do
  # ...
  change manage_relationship(:tracks, type: :direct_control,
    order_is_key: :order)
end
```

With this change, we don't want users to be editing the track order on the form anymore. As the reordering is only done when submitting the form, it would be weird to let them set a number only to change it later. For now, remove the order field from its table cell in the track_inputs function component in TunezWeb.Albums.FormLive, but leave the empty table cell — we'll reuse it in a moment.

08/lib/tunez_web/live/albums/form_live.ex

```
<tr data-id={track_form.index}>
```

13. https://hexdocs.pm/ash/Ash.Changeset.html#manage_relationship/4

```
<td class="px-3 w-20"></td>
<td class="px-3">
```

Now when editing an album, the form will look odd with the missing field, but saving it will set the order attribute on each track to the index of the record in the list. There is one tiny caveat: the list starts from *zero*, as our automatic database indexing starts from zero. No one counts tracks from zero!

We *could* update our track list display to add one to the order field, but this doesn't fix the real problem. Any other views of track data, such as in our APIs, would use the zero-offset value, and be off by one. To solve this, we can keep our zero-indexed order field, but we won't expose it anywhere. Instead, we can separate the concepts of ordering and numbering, and add a calculation for the *number* to display in the UI.

Ordering, numbering, what's the difference?

We're programmers, so we're used to counting things starting at zero, but most people aren't. When we talk about music, or any list of items, we count things starting at one. We even said when we created the order attribute, that the first track would have order 1, etc… and then we didn't actually *do* that. We'll fix that.

In our Tunez.Music.Track resource, add a top-level block for calculations, and define a new calculation:

08/lib/tunez/music/track.ex

```
defmodule Tunez.Music.Track do
  # ...

  calculations do
    calculate :number, :integer, expr(order + 1)
  end
end
```

This uses the same expression[14] syntax we've seen when writing filters, policies, and calculations in the past, to add a new number calculation. It's a pretty simple one, incrementing the order attribute to make it one-indexed.

We'll always want this number calculation loaded, when loading track data. To do that, we can use a custom *preparation*.[15] Like changes add functionality to create and update actions, preparations care used to customize read actions.

14. https://hexdocs.pm/ash/expressions.html
15. https://hexdocs.pm/ash/preparations.html

Add a new preparations block in the Tunez.Music.Track resource, and add a preparation that uses the build/1[16] built-in preparation.

08/lib/tunez/music/track.ex

```
defmodule Tunez.Music.Track do
  use Ash.Resource, # ...

  preparations do
    prepare build(load: [:number])
  end

  # ...
```

Then we can use the number calculation when rendering track details, in the track_details function component:

08/lib/tunez_web/live/artists/show_live.ex

```
<tr :for={track <- @tracks}>
  <th class="whitespace-nowrap w-1 p-3">
➤   {String.pad_leading("#{track.number}", 2, "0")}.
  </th>
```

Perfect! Everything is now in place for the last set of seed data to be imported for Tunez: tracks for all of the seeded albums. To import the track data, run the following on the command line:

```
$ mix run priv/repo/seeds/08-tracks.exs
```

You can also uncomment the last line of the mix seed alias, in the aliases/0 function in mix.exs:

08/mix.exs

```
defp aliases do
  [
    setup: ["deps.get", "ash.setup", "assets.setup", "assets.build", ...],
    "ecto.setup": ["ecto.create", "ecto.migrate"],
➤   seed: [
➤     "run priv/repo/seeds/01-artists.exs",
➤     "run priv/repo/seeds/02-albums.exs",
➤     "run priv/repo/seeds/08-tracks.exs"
➤   ],
    # ...
```

You can run mix seed at any time to fully reset the sample artist, album, and track data in your database. Now, each album will have a full set of tracks. Tunez is looking good!

16. https://hexdocs.pm/ash/Ash.Resource.Preparation.Builtins.html#build/1

Drag n' drop sorting goodness

We have this awesome form: we can add and remove tracks, and everything works well. Managing the order of the tracks is still an issue, though. What if we make a mistake in data entry, and forget track 2? We'd have to remove all the later tracks, and then re-add them after putting track 2 in. It'd be better if we could drag and drop tracks to reorder the list as necessary.

Okay, so our example is a little bit contrived, reordering track lists isn't something that needs to be done often. But reordering lists in general is something that comes up in apps *all* the time — in checklists or todo lists, in your GitHub project board, in your top 5 favorite *Zombie Kittens!!* albums. So let's add it in.

AshPhoenix broadly supports two ways of reordering records in a form — stepping single items up or down the list, or reordering the whole list based on a new order. Both would work for what we want our form to do, but in our experience, the latter is a bit more common and definitely more flexible.

Integrating a SortableJS hook

Interactive functionality like drag and drop generally means integrating a JavaScript library. There are quite a few choices out there, such as Draggable,[17] Interact.js,[18] Pragmatic drag and drop,[19] or you can even build your own using the HTML drag and drop API. We prefer SortableJS.[20]

To that end, we've already set a Phoenix `phx-hook` up on the tracks table, in the `track_inputs` component in `TunezWeb.Albums.FormLive`, that has a basic SortableJS implementation:

08/lib/tunez_web/live/albums/form_live.ex

```
<tbody phx-hook="trackSort" id="trackSort">
  <.inputs_for :let={track_form} field={@form[:tracks]}>
    <tr data-id={track_form.index}>
      <td class="px-3 w-10">
        <span class="hero-bars-3 handle cursor-pointer" />
      </td>
      <td ...>
```

This SortableJS setup is defined in `assets/js/trackSort.js`. It takes the element that the hook is defined on, makes its children `tr` elements draggable, and when

17. https://shopify.github.io/draggable/
18. https://interactjs.io/
19. https://atlassian.design/components/pragmatic-drag-and-drop/about
20. https://sortablejs.github.io/Sortable/

a drag takes place, pushes a "reorder-tracks" event to our liveview with the list of data-ids from the draggable elements.

Note that in our form above, we've also added an icon where the order number input previously sat, to act as a drag *handle*. This is what you click to drag the rows around and reorder them.

With the handle added to the form, you should now be able to drag the rows around by their handles to reorder them. When you drop a row in its new position, your Phoenix server logs will show you that an event was received from the callback defined in the JavaScript hook:

```
[debug] HANDLE EVENT "reorder-tracks" in TunezWeb.Albums.FormLive
  Parameters: %{"order" => ["0", "1", "3", "4", "5", "2", "6", ...]}
[debug] Replied in 433µs
```

This order is the order we've requested that the tracks be ordered in, e.g., in this example, from dragging the third item (index 2) to be placed in the sixth position.

In that "reorder-tracks" event handler, we can use AshPhoenix's sort_forms/3[21] function to reorder the tracks, based on the new order.

08/lib/tunez_web/live/albums/form_live.ex

```
def handle_event("reorder-tracks", %{"order" => order}, socket) do
  socket = update(socket, :form, fn form ->
    AshPhoenix.Form.sort_forms(form, [:tracks], order)
  end)
  {:noreply, socket}
end
```

Give it a try — drag and drop tracks, save the album, and the changed order will be saved. The order (and therefore the number) of each track will be recalculated correctly, and everything is awesome!

Automatic Conversions Between Seconds and Minutes

As we suggested earlier, we don't really want to show a track duration as a number of seconds to users — and that's *any* users, whether they're reading the data on the artist's profile page or editing track data via a form. Users should be able to enter durations of tracks as a string like "3:13", and then Tunez should convert that to a number of seconds before saving it to the database.

21. https://hexdocs.pm/ash_phoenix/AshPhoenix.Form.html#sort_forms/3

Calculating the minutes and seconds of a track

We already have a lot of track data in the database stored in seconds, so the first step is to convert it to a minutes-and-seconds format for display.

We've seen calculations written inline with expressions, such as when we added a number calculation for tracks earlier. Like changes, calculations can also be written using anonymous functions or extracted out to separate calculation modules for reuse. A duration calculation for our Track resource using an anonymous function is written as follows:

```
calculations do
  # ...

  calculate :duration, :string, fn tracks, context ->
    # Code to calculate duration for each track in the list of tracks
  end
end
```

The main difference here is that a calculation function always receives a *list* of records to calculate data for. Even if you're fetching a record by primary key and loading a calculation on the result so there will only ever be one record, the function will still receive a list.

The same behaviour occurs if we define a separate calculation module instead, a module that uses Ash.Resource.Calculation[22] and implements the calculate/3 callback:

08/lib/tunez/music/calculations/seconds_to_minutes.ex

```
defmodule Tunez.Music.Calculations.SecondsToMinutes do
  use Ash.Resource.Calculation

  @impl true
  def calculate(tracks, _opts, _context) do
    # Code to calculate duration for each track in the list of tracks
  end
end
```

This module can then be used as the calculation implementation in the Tunez.Music.Track resource:

08/lib/tunez/music/track.ex

```
calculations do
  calculate :number, :integer, expr(order + 1)
➤ calculate :duration, :string, Tunez.Music.Calculations.SecondsToMinutes
end
```

22. https://hexdocs.pm/ash/Ash.Resource.Calculation.html

The calculate/3 function in the calculation module should iterate over the tracks, and generate nicely-formatted strings representing the number of minutes and seconds of each track. This function should also always *return* a list, where each item of the list is the value of the calculation for the corresponding record in the input list.

08/lib/tunez/music/calculations/seconds_to_minutes.ex

```
def calculate(tracks, _opts, _context) do
  Enum.map(tracks, fn %{duration_seconds: duration} ->
    seconds =
      rem(duration, 60)
      |> Integer.to_string()
      |> String.pad_leading(2, "0")

    "#{div(duration, 60)}:#{seconds}"
  end)
end
```

We would always err on the side of using separate modules to write logic in, instead of anonymous functions. Separate modules allow you to define calculation dependencies using the load/3 callback, document the functionality using describe/1, or even add an alternative implementation of the calculation that can run in the database using expression/2.

An *alternative* implementation — When would that be useful??

Two implementations for every calculation

The way Ash handles calculations is remarkable. Calculations written using Ash's expression syntax can be run *either* in the database or in code. If we had a calculation on the Album resource like

```
calculate :description, :string, expr(name <> " :: " <> year_released)
```

This could be run in the database using SQL, if the calculation is loaded at the same time as the data:

```
iex(1)> Tunez.Music.get_album_by_id!(«uuid», load: [:description])
SELECT a0."id", «the other album fields», (a0."name"::text || ($1 ||
a0."year_released"::bigint))::text::text FROM "albums" AS a0 WHERE (a0."id"
::uuid = $3::uuid) ORDER BY a0."year_released" DESC [" :: ", «uuid»]
#Tunez.Music.Album<description: "Chronicles :: 2022", ...>
```

It can also be run in code using Elixir, if the calculation is loaded on an album already in memory, using Ash.load. By default, Ash will always try to fetch the value from the database to ensure it's up to date, but you can force Ash to

use the data in memory and run the calculation in memory using the reuse_values?: true[23] option:

```
iex(2)> album = Tunez.Music.get_album_by_id!(«uuid»)
SELECT a0."id", a0."name", a0."cover_image_url", a0."created_by_id", ...
#Tunez.Music.Album<description: #Ash.NotLoaded<...>, ...>
iex(3)> Ash.load!(album, :description, reuse_values?: true)
#Tunez.Music.Album<description: "Chronicles :: 2022", ...>
```

Why does this matter? Imagine if, instead of doing a quick string manipulation for our calculation, we were doing something really complicated for every track on an album, and we were loading a *lot* of records at once, such as a band with a huge discography. We'd be running calculations in a big loop that would be slow and inefficient. The database is generally a much more optimized place for running logic with its query planning and indexing; nearly anything that we *can* push into the database, we *should*.

Why are we talking about this now? Because writing calculations in Elixir using calculate/3 is really useful, but it's not the optimal approach. And our calculation for converting a number of seconds to minutes-and-seconds *can* be written using an expression, instead of using Elixir code. It's not an entirely portable expression though, it uses a database fragment to call PostgreSQL's to_char[24] number formatting function.

To use an expression in a calculation module, instead of defining a calculate/3 function, we define a expression/2 function:

08/lib/tunez/music/calculations/seconds_to_minutes.ex

```
defmodule Tunez.Music.Calculations.SecondsToMinutes do
  use Ash.Resource.Calculation

  @impl true
  def expression(_opts, _context) do
    expr(
      fragment("? / 60 || to_char(? * interval '1s', ':SS')",
               duration_seconds, duration_seconds)
    )
  end
end
```

This expression takes the duration_seconds column, converts it to a time, and then formats it. It works pretty well. You can test it in iex by loading a single track and the duration calculation on it:

```
iex(7)> Ash.get!(Tunez.Music.Track, «uuid», load: [:duration])
```

23. https://hexdocs.pm/ash/Ash.html#load/3
24. https://www.postgresql.org/docs/current/functions-formatting.html

```
SELECT t0."id", t0."name", t0."duration_seconds", t0."inserted_at",
t0."updated_at", t0."album_id", t0."order", (t0."duration_seconds"::bigint
/ 60 || to_char(t0."duration_seconds"::bigint * interval '1s', ':SS'))::text
FROM "tracks" AS t0 WHERE (t0."id"::uuid = $1::uuid) LIMIT $2 [«uuid», 2]
#Tunez.Music.Track<duration: "5:04", duration_seconds: 304, ...>
```

Calculations like this are a good candidate for testing!

There's a test in Tunez for this calculation, covering various durations and verifying the result, in test/tunez/music/calculations/seconds_to_minutes_test.exs. This test proved invaluable, because our own initial implementation of the expression didn't properly account for tracks over one hour long!

This expression is pretty short, and *could* be dropped back into our Tunez.Music.Track resource, but keeping it in the module has one distinct benefit — we can re-use it!

Updating the track list with formatted durations

We can also use our SecondsToMinutes calculation module to generate durations for entire albums, with the help of an aggregate. Way way back in Relationship Calculations as Aggregates, on page 79, we wrote aggregates like first and count for an artist's related albums. Ash also provides a sum aggregate type[25] for, you guessed it, summing up data from related records.

So to generate the duration of an album, we can add an aggregate in our Album resource to add up the duration_seconds of all of its tracks, and then re-use the SecondsToMinutes calculation we just wrote to format it!

08/lib/tunez/music/album.ex

```
defmodule Tunez.Music.Album do
  # ...

  aggregates do
    sum :duration_seconds, :tracks, :duration_seconds
  end

  calculations do
    calculate :duration, :string, Tunez.Music.Calculations.SecondsToMinutes
  end
end
```

Now that we have nicely-formatted durations for an album and its tracks, let's update the track list on the artist profile to show them. We can load the track duration calculation as part of the default preparation for Tracks, alongside the number calculation:

25. https://hexdocs.pm/ash/aggregates.html#aggregate-types

08/lib/tunez/music/track.ex

```
preparations do
  prepare build(load: [:number, :duration])
end
```

Album durations are less critical — for now we probably only need them on this artist profile page. In Tunez.Artists.ShowLive, load the duration for each album:

08/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(%{"id" => artist_id}, _url, socket) do
  artist =
    Tunez.Music.get_artist_by_id!(artist_id,
      load: [albums: [:duration, :tracks]],
      actor: socket.assigns.current_user
    )
```

The album_details function component can then be updated to include the duration of the album:

08/lib/tunez_web/live/artists/show_live.ex

```
<.h2>
  {@album.name} ({@album.year_released})
  <span :if={@album.duration} class="text-base">({@album.duration})</span>
</.h2>
```

And the track_details function component can be updated to use the duration field instead of duration_seconds.

08/lib/tunez_web/live/artists/show_live.ex

```
<tr :for={track <- @tracks}>
  <% # ... %>
  <td class="whitespace-nowrap w-1 text-right p-2">{track.duration}</td>
</tr>
```

And it looks *awesome*!

Verda Horizonto (2023) (37:37) Delete Edit

01.	Vojaĝo al la Lumo	4:20
02.	Arbaraj Sentoj	3:55
03.	Flustro de la Vento	5:10

There's only one last thing we need to make better: the Album form, so users can enter human-readable durations, instead of seconds.

Calculating the seconds of a track

At the moment, the actions in the Tunez.Music.Track resource will accept data for the duration_seconds attribute, in both the create and update actions, and save it to the data layer. Instead of accepting the attribute directly, we can pass in the formatted version of the duration as an argument to the action, and then use a change to process that argument. To prevent the change running when no duration argument is provided, use the only_when_valid? option when configuring the change.

Again, the update action should be marked with require_atomic?: false. This change *could* be written in an atomic way (more on that in We Need to Talk About Atomics, on page 249), but because these actions are already running non-atomically via the album, we'll leave it as-is.

08/lib/tunez/music/track.ex

```
actions do
  # ...

  create :create do
    primary? true
➤   accept [:order, :name, :album_id]
➤   argument :duration, :string, allow_nil?: false
➤   change Tunez.Music.Changes.MinutesToSeconds, only_when_valid?: true
  end

  update :update do
    primary? true
➤   accept [:order, :name]
➤   require_atomic? false
➤   argument :duration, :string, allow_nil?: false
➤   change Tunez.Music.Changes.MinutesToSeconds, only_when_valid?: true
  end
end
```

This means that we can call the actions with a map of data, including a duration key, and the outside world doesn't need to know anything about the internal representation or storage of the data.

Now we need to implement the MinutesToSeconds change module, which should be in a new file at lib/tunez/music/changes/minutes_to_seconds.ex. Like the UpdatePreviousNames module we created for artists in Defining a change module, on page 55, this will be a separate module that uses Ash.Resource.Change,[26] and defines a change/3 action:

08/lib/tunez/music/changes/minutes_to_seconds.ex

```
defmodule Tunez.Music.Changes.MinutesToSeconds do
```

26. https://hexdocs.pm/ash/Ash.Resource.Change.html

```
  use Ash.Resource.Change

  @impl true
  def change(changeset, _opts, _context) do
  end
end
```

This change function can have any Elixir code in it, so we can extract the duration argument from the provided changeset, validate the format and value, and convert it to a number:

08/lib/tunez/music/changes/minutes_to_seconds.ex

```
def change(changeset, _opts, _context) do
  {:ok, duration} = Ash.Changeset.fetch_argument(changeset, :duration)

  with :ok <- ensure_valid_format(duration),
       :ok <- ensure_valid_value(duration) do
    changeset
    |> Ash.Changeset.change_attribute(:duration_seconds, to_seconds(duration))
  else
    {:error, :format} ->
      Ash.Changeset.add_error(changeset, field: :duration,
        message: "use MM:SS format"
      )

    {:error, :value} ->
      Ash.Changeset.add_error(changeset, field: :duration,
        message: "must be at least 1 second long"
      )
  end
end

defp ensure_valid_format(duration) do
  if String.match?(duration, ~r/^\d+:\d{2}$/) do
    :ok
  else
    {:error, :format}
  end
end

defp ensure_valid_value(v) when v in ["0:00", "00:00"], do: {:error, :value}
defp ensure_valid_value(_value), do: :ok

defp to_seconds(duration) do
  [minutes, seconds] = String.split(duration, ":", parts: 2)
  String.to_integer(minutes) * 60 + String.to_integer(seconds)
end
```

It's a little bit long, but it neatly encapsulates our requirements.

These checks in the change module might feel a bit like validations, that belong in the Track resource. We'd argue that they specifically relate to the duration *argument* being processed, and not any attributes on the resource

itself. If we wanted to add support for other duration formats later, such as "2m12s" or "five minutes", we'd only have to update the code in one place — here in this change module, to validate and parse the value.

You can test the change out in iex by building a changeset for a track. You don't need to submit it or even validate it, but you'll see the conversion:

```
iex(4)> Tunez.Music.Track
Tunez.Music.Track
iex(5)> |> Ash.Changeset.for_create(:create, %{duration: "02:12"})
#Ash.Changeset<
  attributes: %{duration_seconds: 132},
  arguments: %{duration: "02:12"},
  ...
```

Invalid values will report the "use MM:SS format" error, and missing values will report that the field is required.

The very very last thing left to do is to update our Album form to use the duration attribute of tracks, instead of duration_seconds. For existing tracks, this will display the formatted value (which is auto-loaded via the load preparation) and then convert it back to seconds on save. The UI is none the wiser!

08/lib/tunez_web/live/albums/form_live.ex

```
<tr data-id={track_form.index}>
  <% # ... %>
  <td class="px-0 w-36">
➤   <label for={track_form[:duration].id} class="hidden">Duration</label>
➤   <.input field={track_form[:duration]} />
  </td>
```

Adding Track Data to API Responses

We can't forget about our API users; they'd like to be able to see track information for albums, too! To support the Track resource in the APIs, use the ash.extend Mix task to add the extensions and the basic configuration:

```
$ mix ash.extend Tunez.Music.Track json_api
$ mix ash.extend Tunez.Music.Track graphql
```

Because we'll always be reading or updating tracks in the context of an album, we don't need to add any JSON API endpoints or GraphQL queries or mutations for them: the existing album endpoints will be good enough. We do, however, need to mark relationships and attributes as public?: true if we want them to be readable. This includes the tracks relationship in the Tunez.Music.Album resource:

08/lib/tunez/music/album.ex

```
relationships do
```

```
  # ...

  has_many :tracks, Tunez.Music.Track do
    sort order: :asc
➤   public? true
  end
```

And the attributes to show for each track, in the Tunez.Music.Track resource. This doesn't have to include our internal order or duration_seconds attributes!

08/lib/tunez/music/track.ex

```
attributes do
  # ...

  attribute :name, :string do
    allow_nil? false
➤   public? true
  end

  # ...
end

calculations do
  calculate :number, :integer, expr(order + 1) do
➤   public? true
  end

  calculate :duration, :string, Tunez.Music.Calculations.SecondsToMinutes do
➤   public? true
  end
end
```

This is all we need to do for GraphQL. As you only fetch the fields you specify, consumers of the API can automatically fetch tracks of an album, and can read all, some, or none of the track attributes if they want to. You may want to disable automatic filterability and sortability with derive_filter? false and derive_sort? false in the Track resource, but that's about it.

Special treatment for the JSON API

Our JSON API needs a little more work, though. To allow tracks to be included when reading an album, we need to manually configure that with the includes option in the Tunez.Music.Album resource:

08/lib/tunez/music/album.ex

```
json_api do
  type "album"
➤ includes [:tracks]
end
```

This will allow users to add the include=tracks query parameter to their requests to Album-related endpoints, and the track data will be included. If you want

to allow tracks to be includable when reading *artists*, e.g., when searching or fetching an artist by ID, that includes option must be set separately as part of the Tunez.Music.Artist json_api configuration.

08/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  json_api do
    type "artist"
➤   includes albums: [:tracks]
    derive_filter? false
  end
```

With this config, users can request either albums to be included for an artist with include=albums in the query string, or albums and their tracks with include=albums.tracks. Neat!

As we learned in What data gets included in API responses?, on page 89, by default only *public attributes* will be fetched and returned via the JSON API. This isn't great for tracks, because only the name is a public attribute — number and duration are both calculations! For tracks, it would make more sense to configure the default_fields[27] that are always returned for every response, this way we can include the attributes and calculations we want.

08/lib/tunez/music/track.ex

```
json_api do
  type "track"
➤ default_fields [:number, :name, :duration]
end
```

Now our API users also have a good experience! They can access and manage track data for albums, just like web UI users can.

We covered a lot in this chapter, and there are so many little fiddly things about forms to make them *just right*. It'll take practice getting used to, especially if you want to build forms with different UIs such as adding/removing tags, but the principles will stay the same.

In our next chapter, we'll start adding some personalization to Tunez, using everything we've learned so far to let users follow their favorite artists. And we'll make it *smart* — building code interfaces that really speak our domain language, and uncovering insights like who the most popular artists are. It'll be fun!

27. https://hexdocs.pm/ash_json_api/dsl-ashjsonapi-resource.html#json_api-default_fields

CHAPTER 9

Now Following: Your Favorite Artists

Tunez is really starting to come together — we're collecting a lot of really useful information about artists, and users can quickly find the information they're looking for. But the app is completely *static*. There's no reason for users to engage, to regularly check back and see what's new, because they can't easily get updates on things they're interested about. We need that cool factor. We want users to be able to make Tunez work for *them*!

As part of that cool factor, we'll add a notification system to the app, so we can immediately find out when our favorite artists release new albums. But before we can get notified about updates for the artists we follow, we first need Tunez to know who our followed artists *are*.

Modelling with a Many-to-Many Relationship

We can model the link between users and their followed artists with a *many-to-many*[1] relationship — each user can have many followed artists, and each artist can have many ardent followers.

In Ash (and in a lot of other frameworks), this is implemented using a *join resource*. This join resource will sit in between our two existing resources of `Tunez.Music.Artist` and `Tunez.Accounts.User`, joining them together, and have a `belongs_to` relationship to each of them. Thus, each link between a user and an artist will be a record in the join resource — if ten users each follow ten different artists, then the join table will have 100 records.

Creating the ArtistFollower resource

The hardest problem in computer science is always naming things, and resources can be no exception. What should we *call* this join resource? Some

1. https://hexdocs.pm/ash/relationships.html#many-to-many

join relationships naturally lend themselves to nice names such as "GroupMembership" or "MailingListSubscription", but a lot don't. Ultimately, as long as the name makes sense, it doesn't really matter. If all else fails, smush the two resource names together, e.g. "ArtistUser". We've chosen ArtistFollower, but it could just as easily have been something like "LikedArtist" or "FavoriteArtist".

And which domain should it go in? This is our first cross-domain relationship, so should it go in the Tunez.Music or the Tunez.Accounts domain? Again, it doesn't make a huge difference. We've chosen Tunez.Music, as the relationship will be made in the direction of users -> artists, so it "feels" closer to the music side.

With all the big decisions out of the way, we'll generate our basic resource:

```
$ mix ash.gen.resource Tunez.Music.ArtistFollower --extend postgres
```

Inside the new resource in lib/tunez/music/artist_follower.ex, we won't be storing any data so we don't need any attributes. We *do* need to add relationships, though, for the user doing the following and the artist they want to follow:

09/lib/tunez/music/artist_follower.ex

```
defmodule Tunez.Music.ArtistFollower do
  # ...

  relationships do
    belongs_to :artist, Tunez.Music.Artist do
      primary_key? true
      allow_nil? false
    end

    belongs_to :follower, Tunez.Accounts.User do
      primary_key? true
      allow_nil? false
    end
  end
end
```

There's something interesting in this snippet: we didn't add an id attribute to use as a primary key, but we *do* need some way of uniquely identifying each record of the join resource. The combination of the two belongs_to foreign keys works well for this purpose, as a *composite primary key* — a user can't follow the same artist more than once, so the combination of follower_id and artist_id will always be unique. Adding primary_key? true to both relationships will create one primary key with both columns.

If an artist gets deleted, or a user deletes their account, we want to set the on_delete property of the foreign keys to delete all of the follower links, just like we did with an album's tracks or an artist's albums:

09/lib/tunez/music/artist_follower.ex

```
postgres do
  # ...

  references do
    reference :artist, on_delete: :delete, index?: true
    reference :follower, on_delete: :delete
  end
end
```

Now that the resource is set up, generate a migration for it, and then run the migration:

```
$ mix ash.codegen create_artist_followers
$ mix ash.migrate
```

Using ArtistFollower to link Artists and Users

With the join resource in place, we can define the many-to-many relationship we're after. It will go both ways — from a user record, we'll be able to load all of their followed artists; and from an artist record, we'll be able to load all of their followers.

In the Tunez.Music.Artist resource, we first define a has_many relationship for the join resource, and then a many_to_many[2] relationship using that has_many relationship:

09/lib/tunez/music/artist.ex

```
relationships do
  # ...

  has_many :follower_relationships, Tunez.Music.ArtistFollower

  many_to_many :followers, Tunez.Accounts.User do
    join_relationship :follower_relationships
    destination_attribute_on_join_resource :follower_id
  end
end
```

By default, Ash will look for a foreign key matching the name of the resource we're linking to, in this case a user_id because the many-to-many relationship is for a User resource. Because we've used follower_id in the join resource, to make it super clear which way the relationship goes, we have to specify that that's the key to use to link through, using destination_attribute_on_join_resource.[3]

2. https://hexdocs.pm/ash/relationships.html#many-to-many
3. https://hexdocs.pm/ash/dsl-ash-resource.html#relationships-many_to_many-destination_attribute_on_join_resource

It's not strictly necessary to define the join relationship — we could have written the many-to-many relationship to go *through* the join resource directly:

```
many_to_many :followers, Tunez.Accounts.User do
  through Tunez.Music.ArtistFollower
  destination_attribute_on_join_resource :follower_id
end
```

Ash would still set up a relationship behind the scenes, named artist_followers_join_assoc, but we wouldn't have any access to it. This might be okay for your use case, but wouldn't allow any customization of the relationship, such as sorting and filtering.

In our use case, most of the questions we'll be asking can also be answered by the join relationship directly. How many followers does a given artist have? Is the authenticated user one of them? Using the join relationship will save us, well, an extra database join for every query!

Add similar relationships in the Tunez.Accounts.User resource, to create the many-to-many with the Tunez.Music.Artist resource:

09/lib/tunez/accounts/user.ex

```
relationships do
  has_many :follower_relationships, Tunez.Music.ArtistFollower do
    destination_attribute :follower_id
  end

  many_to_many :followed_artists, Tunez.Music.Artist do
    join_relationship :follower_relationships
    source_attribute_on_join_resource :follower_id
  end
end
```

To be able to use the Tunez.Music.ArtistFollower join resource, it also needs at least a basic read action. We can add a default one, with a policy to allow anyone to read them:

09/lib/tunez/music/artist_follower.ex

```
defmodule Tunez.Music.ArtistFollower do
  use Ash.Resource,
    # ...
    authorizers: [Ash.Policy.Authorizer]

  # ...

  actions do
    defaults [:read]
  end

  policies do
```

```
    policy action_type(:read) do
      authorize_if always()
    end
  end
end
```

Now we can run a query in iex, and see the follower relationships of an artist:

```
iex(8)> Tunez.Music.get_artist_by_id!(«uuid», load: [:follower_relationships])
«two SQL queries to load the data»
#Tunez.Music.Artist<follower_relationships: [], ...>
```

It appears to work! But it's not super exciting, as no artists have any followers yet. Let's build the user interface to let users see and update which artists they follow.

Who Do You Follow?

With our shiny new relationships in place, we can use them to determine if a given user follows a given artist, and show that information in the app.

We'l write this as a custom calculation, as an expression that uses the exists[4] sub-expression. exists lets us check if any records in a relationship match a given condition, so we can use it to check if any of an artist's followers are the current user, the actor running this query.

This will be loaded as part of an Artist record and shown on the artist profile page, so the calculation can be put on the Tunez.Music.Artist resource:

09/lib/tunez/music/artist.ex

```
calculations do
  calculate :followed_by_me,
            :boolean,
            expr(exists(follower_relationships, follower_id == ^actor(:id)))
end
```

This uses the same actor template[5] we used when writing policies based on the current user's role, back in Filling out policies, on page 153. We *could* write it in a more generic way to check for a user passed in as an argument (because calculations can take arguments!),[6] or a list of users (maybe we add user friends down the track, and we want to see if any of our friends follow an artist), but for now we only care about who the logged-in user is following.

4. https://hexdocs.pm/ash/expressions.html#sub-expressions
5. https://hexdocs.pm/ash/expressions.html#templates
6. https://hexdocs.pm/ash/calculations.html#arguments-in-calculations

Showing the current following status

The option to follow or unfollow a user will be shown on an artist's profile page, up in the header:

The star will be filled in if they are following the user, and hollow like this if they're not. Clicking the star will toggle the follow status, from following to unfollowing and back again.

To show the current follow status (is the user following this artist or not?) we'll load the new followed_by_me calculation, when we load the artist data in Tunez.Artists.ShowLive. The calculation requires an actor, and we're already supplying the actor when we call Tunez.Music.get_artist_by_id!, so everything will work.

09/lib/tunez_web/live/artists/show_live.ex

```
def handle_params(%{"id" => artist_id}, _url, socket) do
  artist =
    Tunez.Music.get_artist_by_id!(artist_id,
      load: [:followed_by_me, albums: [:duration, :tracks]],
      actor: socket.assigns.current_user
    )
  # ...
```

We've pre-prepared a function component named follow_toggle that will use the value and show the follow status, so add it after the artist name:

09/lib/tunez_web/live/artists/show_live.ex

```
<.h1>
  {@artist.name}
  <.follow_toggle on={@artist.followed_by_me} />
</.h1>
```

Clicking the star icon to follow the artist will do a little animation and then trigger the "follow" event handler, defined further down in the liveview. It currently doesn't do anything, but we'll flesh out the functionality now.

Following a new artist

Our liveview doesn't know anything about how our data is structured, or the relationships between our resources. And it doesn't need to care! If we provide a nice code interface function like Tunez.Music.follow_artist(@artist, actor: current_user), the code in our liveview can be super simple and we can tuck all the logic away inside our domain and resources.

Our envisaged follow_artist function will take the artist record as an argument, and create a new ArtistFollower record. In the Tunez.Music domain module, add

the code interface that describes this, pointing to a (not yet defined) create action in the ArtistFollower resource:

09/lib/tunez/music.ex

```
resources do
  # ...

  resource Tunez.Music.ArtistFollower do
    define :follow_artist, action: :create, args: [:artist]
  end
end
```

Next we need the create action, in the Tunez.Music.ArtistFollower resource. What arguments should it take?

Structs for action arguments, and custom inputs

We *could* add an argument for the artist record to the create action, and validate the type with a constraint[7] to make sure it's a real Artist:

```
create :create do
  argument :artist, :struct do
    allow_nil? false
    constraints instance_of: Tunez.Music.Artist
  end
  # ...
```

This would create the function definition we want to use in our web app… but our web app *isn't* the only interface that might be using this action. What if we also wanted to allow users to follow artists via the GraphQL API? Let's add that and see what it looks like.

To enable GraphQL support for the ArtistFollower resource, extend the resource with graphql:

```
$ mix ash.extend Tunez.Music.ArtistFollower graphql
```

And then add a new mutation for the create action, in the Tunez.Music domain module:

09/lib/tunez/music.ex

```
graphql do
  # ...

  mutations do
    # ...
    create Tunez.Music.ArtistFollower, :follow_artist, :create
  end
end
```

7. https://hexdocs.pm/ash/dsl-ash-resource.html#actions-create-argument-constraints

In the GraphQL playground at http://localhost:4000/gql/playground, the new followArtist mutation will be listed. It has an input argument of a generated FollowArtistInput! type, which is…

```
type FollowArtistInput { artist: JsonString! }
```

Oh, gross. The mutation expects a JSON-serialized version of the artist record! Ideally, our APIs would accept the *ID* of the artist to follow. To get that, we'd have to use the artist ID as the argument to the create action, instead of the full artist record.

```
create :create do
  argument :artist_id, :uuid do
    allow_nil? false
  end
```

Or because artist_id is an attribute of the resource, via the :artist relationship, we could accept the attribute directly:

09/lib/tunez/music/artist_follower.ex

```
create :create do
  accept [:artist_id]
```

But now the code interface is wrong, it requires you to pass in an artist ID instead of an artist struct! So annoying.

Way back in code on page 82, we saw an example of how we could configure the code interface to add extra functionality to the action, like loading related records. Our APIs for searching wouldn't load the data, but calling the interface *would*. We can use a similar pattern for pre-processing the arguments to the create action at the code interface layer, using *custom inputs*.[8]

Our code interface function can still accept the artist argument, which will be the full Artist record. But we'll define that argument as a custom input for the code interface specifically — this will let us write a transform function, to convert it to the artist_id argument that the action expects.

09/lib/tunez/music.ex

```
resource Tunez.Music.ArtistFollower do
  define :follow_artist do
    action :create
    args [:artist]

    custom_input :artist, :struct do
      constraints instance_of: Tunez.Music.Artist
      transform to: :artist_id, using: & &1.id
    end
```

8. https://ash-project.github.io/ash/code-interfaces.html#customizing-the-generated-function

```
  end
end
```

It's a little bit verbose, and it requires using the block syntax for all of the options for the code interface, but this is a *really* powerful (and customizable) technique. The constraint on the artist argument has moved from the action to the code interface, so the code interface function still accepts (and type-checks) an artist:

```
iex(1)> h Tunez.Music.follow_artist

             def follow_artist(artist, params \\ nil, opts \\ nil)

Calls the create action on Tunez.Music.ArtistFollower.
```

But other interfaces that derive directly from our actions, such as the GraphQL API, will use the ID instead:

```
type FollowArtistInput { artistId: ID! }
```

That's neat!

We've suitably addressed that issue, and the artist relationship will be correctly set on the ArtistFollower record. For the follower relationship, the only other data in this resource, we can use the relate_actor[9] built-in change.

09/lib/tunez/music/artist_follower.ex

```
create :create do
  accept [:artist_id]

  change relate_actor(:follower, allow_nil?: false)
end
```

And who should be authorized to run this create action in our resource? Well, anyone, really, as long as they're logged in so we know who is following who! We can add a policy for that:

09/lib/tunez/music/artist_follower.ex

```
policies do
  # ...

  policy action_type(:create) do
    authorize_if actor_present()
  end
end
```

After adding the policy, you can test out the action in iex. We've also added some tests in the Tunez app for it, in test/tunez/accounts/artist_follower_test.exs.

```
iex(5)> artist = Tunez.Music.get_artist_by_id!(«artist_uuid»)
```

9. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html#relate_actor/2

```
#Tunez.Music.Artist<...>
iex(6)> user = Ash.get!(Tunez.Accounts.User, «user_uuid», authorize?: false)
#Tunez.Accounts.User<...>
iex(7)> Tunez.Music.follow_artist(artist, actor: user)
INSERT INTO "artist_followers" ("artist_id","follower_id") VALUES ($1,$2)
RETURNING "follower_id","artist_id" [«artist_uuid», «user_uuid»]
{:ok, #Tunez.Music.ArtistFollower<...>}
```

In the "follow" event handler in our TunezWeb.Artists.ShowLive, we'll use the new function to make the current user follow the artist being shown, and handle the response accordingly. In the successful case, we don't need to show a flash message, but we *can* update the loaded artist record to say that yes, we now follow them!

09/lib/tunez_web/live/artists/show_live.ex

```
def handle_event("follow", _params, socket) do
  socket =
    case Tunez.Music.follow_artist(socket.assigns.artist,
           actor: socket.assigns.current_user
         ) do
      {:ok, _} ->
        update(socket, :artist, & %{&1 | followed_by_me: true})

      {:error, _} ->
        put_flash(socket, :error, "Could not follow artist")
    end

  {:noreply, socket}
end
```

Clicking the follow star will now do a little spin and then fill in, showing that the artist is now followed. Awesome! Follow all of the artists!!!

Unfollowing an old artist

But maybe we're just not digging some of this music anymore, and want to unfollow some of these artists. Clicking the star icon again should *unfollow* them, reverting back to our previous state.

Unfollowing, or deleting the relevant ArtistFollower record, is a little trickier to implement than following. To make a similar API to following artists, we'd define a code interface like Tunez.Music.unfollow_artist(@artist, actor: current_user) that pointed to a destroy action in the Tunez.Music.ArtistFollower resource. But the first argument to a typical destroy action is the record to be destroyed — which we don't have here.

Ash has our back, as usual. If we add the require_reference?[10] option to our code interface, we can skip providing a record to be deleted, and write some logic in the action to find the correct record instead.

Using the same idea with a custom input for the artist, our code interface would look like this:

09/lib/tunez/music.ex

```
resource Tunez.Music.ArtistFollower do
  # ...
  define :unfollow_artist do
    action :destroy
    args [:artist]
    require_reference? false

    custom_input :artist, :struct do
      constraints instance_of: Tunez.Music.Artist
      transform to: :artist_id, using: & &1.id
    end
  end
end
```

What would the action look like, though? If we had an empty action, that just accepted the artist_id argument (after the custom input transformation) but didn't do anything with it:

```
destroy :destroy do
  argument :artist_id, :uuid do
    allow_nil? false
  end
end
```

The result (if it wasn't currently prevented by missing policies!) would be surprising — it would delete *all* ArtistFollower records! When we provide a single record to a destroy action to be destroyed, it's used as a filter by Ash internally, to delete the record in the data layer with the same primary key. Without that filter, Ash will try and delete eeeeeeverything. This is very clearly *not* what we want.

To fix this, we'll add our own filter to the action, in a very similar way as we do when filtering read actions. The only difference is that we have to apply the filter as a *change*, instead of calling it directly:

09/lib/tunez/music/artist_follower.ex

```
destroy :destroy do
  argument :artist_id, :uuid do
    allow_nil? false
```

10. https://hexdocs.pm/ash/dsl-ash-resource.html#code_interface-define-require_reference?

```
    end
➤   change filter(expr(artist_id == ^arg(:artist_id) &&
➤     follower_id == ^actor(:id)))
  end
```

For an authorization policy, we'll use the same actor_present built-in check — our filter already accounts for the actor, to ensure that they can only delete their *own* followed artists:

09/lib/tunez/music/artist_follower.ex

```
policies do
  # ...

  policy action_type(:destroy) do
    authorize_if actor_present()
  end
end
```

This behaves as expected, and will delete only the record we want, but the return type is slightly odd:

```
iex(8)> Tunez.Music.unfollow_artist!(artist, actor: user)
«SQL query to delete ArtistFollowers»
%Ash.BulkResult{
  status: :success, errors: nil, records: nil,
  notifications: [], error_count: 0
}
```

A short detour into bulk actions

We'll revisit bulk actions more in the next chapter on page 227, but the short version is that most actions can be run for *one* record (e.g. destroy this one record in particular) or in *bulk* (destroy this entire list of records). Switching between the two behaviours depends on what you call the action with — if you call a create action with a list of records to be created, as opposed to a single map of data, you'll get a bulk create.

Our destroy action uses a filter to narrow down which records to delete, but a filter will *always* return a list, even if that list only has one record in it. Because a list is being passed to the underlying core destroy functionality, we get the bulk behaviour of the action, and get a special bulk result type back.

We *could* still use the action as-is, and match on the BulkResult in our liveview, but that's leaking implementation details out into our view. It shouldn't care what we're doing behind the scenes!

Earlier, we saw how to use the get_by option[11] on code interfaces, to read a single record by some unique field such as id. Using get_by will automatically enable a few other options behind the scenes, including get? true — the Ash flag for saying that this function will return at most one record. If our unfollow_artist code interface uses get? true instead of require_reference? false, then the bulk result will be introspected, and if exactly zero or one record was deleted, it will return :ok like a standard destroy action.

Note that if the action deletes *more* than one result, an error will be returned. This error is generated *after* the actual deletion is complete, so the records will still be deleted despite the error response. Test your actions thoroughly! We've added tests for our destroy in test/tunez/music/artist_follower_test.exs, to ensure that our filter is working and only the correct record is destroyed.

Using get? true will also automatically set require_reference? false, which is super convenient for us. Updating the options means that our action works as we want — we'll get either :ok, or an error tuple.

09/lib/tunez/music.ex

```
resource Tunez.Music.ArtistFollower do
  # ...
  define :unfollow_artist do
    action :destroy
    args [:artist]
➤   get? true

    # ...
  end
end

iex(6)> Tunez.Music.unfollow_artist(artist, actor: user)
:ok
```

Integrating the code interface into the liveview

Back in the TunezWeb.Artist.ShowLive liveview, we now have the pieces to connect up in the "unfollow" event handler, much like we did for "follow". We expect this to always be :ok, but in the off chance that something goes wrong, we can let the user know.

09/lib/tunez_web/live/artists/show_live.ex

```
def handle_event("unfollow", _params, socket) do
  socket =
    case Tunez.Music.unfollow_artist(socket.assigns.artist,
           actor: socket.assigns.current_user
```

11. https://hexdocs.pm/ash/dsl-ash-domain.html#resources-resource-define-get_by

```
          ) do
        :ok ->
          update(socket, :artist, & %{&1 | followed_by_me: false})

        {:error, _} ->
          put_flash(socket, :error, "Could not unfollow artist")
      end

    {:noreply, socket}
  end
```

Authenticated users will now be able to follow and unfollow any artist, by clicking on the little star icon on an artist's profile. But wait, so can *unauthenticated* users! We added authorization policies for the action, but not in the view — so all users can see the star icon.

To fix this, we can add a policy check when calling the follow_toggle function component to render the star in TunezWeb.Artists.ShowLive. Ash has auto-generated can_follow_artist? and can_unfollow_artist? functions for our code interfaces, so you can pick one to conditionally render the icon.

09/lib/tunez_web/live/artists/show_live.ex

```
<.h1>
  {@artist.name}
  <.follow_toggle
    :if={Tunez.Music.can_follow_artist?(@current_user, @artist)}
    on={@artist.followed_by_me}
  />
</.h1>
```

Spicing Up the Artist Catalog

With this one new relationship, we can do some pretty neat things using ideas and concepts we've already learned. Let's make the artist catalog a bit more interesting!

Showing the follow status for each artist

It's a pain to have to click through to the artist profile to see if we follow them or not, so let's add a little "following" icon to artists in the catalog if the logged-in user follows them.

In TunezWeb.Artists.IndexLive, we load the artists to display with the Tunez.Music.search_artists function. This has all of its load statements tucked away on the code interface function, in case we want to reuse the whole search. To add loading the followed_by_me calculation for each artist, edit the options in the code interface in Tunez.Music:

09/lib/tunez/music.ex

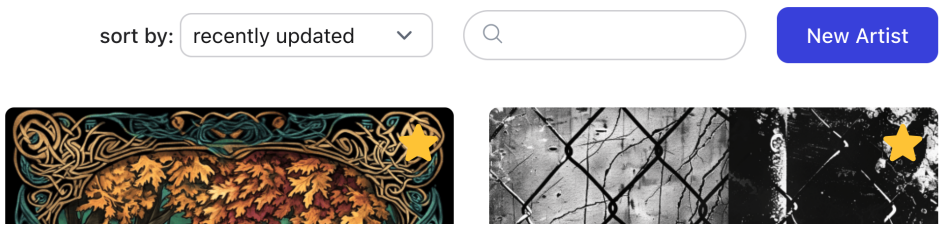
```
resource Tunez.Music.Artist do
  # ...

  define :search_artists,
    action: :search,
    args: [:query],
    default_options: [
      load: [
➤       :followed_by_me, :album_count, :latest_album_year_released, :cover_image_url
      ]
    ]
end
```

Then we can render an icon in each artist_card of the liveview, if the user follows the artist. We've included a small follow_icon component for this purpose:

09/lib/tunez_web/live/artists/index_live.ex

```
<.link navigate={~p"/artists/#{@artist.id}"}>
➤   <.follow_icon :if={@artist.followed_by_me} />
  <.cover_image image={@artist.cover_image_url} />
</.link>
```

Each of the artist album covers will now show a small star icon if you've followed them. Pretty nifty!



While we're in here, why don't we show how many followers each artist has?

Showing follower counts for each artist

Like we wrote an aggregate to count albums for an artist, we can write an aggregate to count their followers as well. This will go in the aggregates block, in the Tunez.Music.Artist resource:

09/lib/tunez/music/artist.ex

```
aggregates do
  # ...

➤   count :follower_count, :follower_relationships
end
```

We don't need to know who the followers actually *are*, just how many there are, so we can use the join relationship in the aggregate. To show this new aggregate in the artist catalog, again edit the options in the code interface to load it when searching artists:

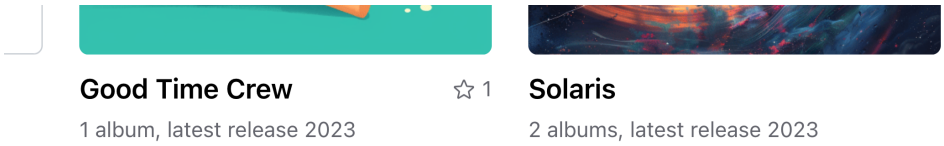
09/lib/tunez/music.ex

```
resource Tunez.Music.Artist do
  # ...

  define :search_artists,
    action: :search,
    args: [:query],
    default_options: [
      load: [
➤       :follower_count, :followed_by_me, :album_count,
        :latest_album_year_released, :cover_image_url
      ]
    ]
end
```

And then use the aggregate value in the artist_card function component, in the Tunez.Artist.IndexLive liveview. We've provided a follower_count_display component, which will show friendly numbers like "12" or "3.6K" or "22.1M".

09/lib/tunez_web/live/artists/index_live.ex

```
<p class="flex justify-between">
  <.link ...>{@artist.name}</.link>
➤ <.follower_count_display count={@artist.follower_count} />
</p>
```



In your development Tunez app it's probably not really exciting to view, because you might only have one or two accounts that follow a handful of artists. In a real app though, as people sign up and follow artists, you might start seeing some popularity trends! Let's surface some of those trends.

Sorting artists by follow status and follower count

In Sorting based on aggregate data, on page 83 we learned how to use aggregates like album_count to sort search results. You can also sort by calculations — if we sort by the followed_by_me calculation, all of the user's followed artists would show up first in the search results. We can also add an option for sorting by artist popularity!

The list of sort options is in the sort_options/0 function, in TunezWeb.Artists.IndexLive. We can add -followed_by_me and -follower_count to the end of the list, the - signifying to sort in descending order to get true/higher values first:

09/lib/tunez_web/live/artists/index_live.ex

```
defp sort_options do
  [
    # ...
    {"latest album release", "--latest_album_year_released"},
➤   {"popularity", "-follower_count"},
➤   {"followed artists first", "-followed_by_me"}
  ]
end
```

This sort value is used with the sort_input option for building queries, meant for untrusted user input. To signify that yes, we'll allow these fields to be sorted on, they have to be marked public? true in the Tunez.Music.Artist resource:

09/lib/tunez/music/artist.ex

```
calculations do
  calculate :followed_by_me,
            :boolean,
            expr(exists(follower_relationships, follower_id == ^actor(:id))) do
➤   public? true
  end
end

aggregates do
  # ...

  count :follower_count, :follower_relationships do
➤   public? true
  end
end
```

And this works great! Let's take a minute to think about what we've built here.

Our catalog displays a lot of different information — each of our calculations/aggregates would have to be written as a separate Ecto subquery, that could be both selected and possibly sorted on. We'd also likely need a separate library like Flop,[12] to do a lot of the heavy lifting.

But with Ash, we've been working at a higher level of abstraction. We've defined relationships between our resources, and we got all this extra functionality basically for free, using standard Ash features like calculations and aggregates. It's pretty amazing!

12. https://hexdocs.pm/flop

Now that we know which artists a user follows, we can move on to what we *really* want to build - a real-time notification system!

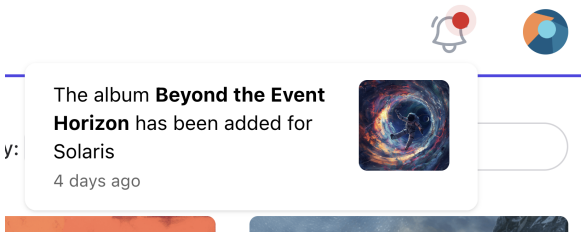
CHAPTER 10

PubSub and Real-Time Notifications

In the last chapter, we did a lot of the setup work for building our notification system — we now know who each user's favorite artists are. We also used that information in some cool ways, such as sorting artists by popularity. There was a lot of bang for our follower buck! And now we can build out the notification functionality.

Notifying Users about New Albums

The web app currently has a notification bell in the top menu for authenticated users, but there's never been any notifications to display.... until now. Our end goal here is that users will receive notifications when new albums are added for the artists they follow. These notifications should be persisted, and stay until the user clicks on them.



To do this, we'll need a new resource representing a notification message. In Tunez our notifications will only ever be for showing new albums to users, so the resource can be pretty simple — it will only need to store who to show the notification *to*, and the album to show the notification *for*.

Creating the Notification resource

Like our `ArtistFollower` resource, this new `Notification` resource crosses domain boundaries in linking users in the `Tunez.Accounts` domain and albums in the

Tunez.Music domain. Notifications are pretty personalized though, and "feel" closer to users, so we'll put the new resource in the Tunez.Accounts domain.

First, generate a new empty resource:

```
$ mix ash.gen.resource Tunez.Accounts.Notification --extend postgres
```

And then add the attributes and relationships we want to store:

10/lib/tunez/accounts/notification.ex

```
postgres do
  # ...

  references do
    reference :user, index?: true, on_delete: :delete
    reference :album, on_delete: :delete
  end
end

attributes do
  uuid_primary_key :id

  create_timestamp :inserted_at
end

relationships do
  belongs_to :user, Tunez.Accounts.User do
    allow_nil? false
  end

  belongs_to :album, Tunez.Music.Album do
    allow_nil? false
  end
end
```

We've included an id attribute because we'll be wanting to dismiss/delete individual notifications when they're clicked on, as well as an inserted_at timestamp so we can show how long ago the notifications were generated. Because the notifications should be deleted if either the user or the album is deleted, we've also configured the database references for the relationships to be on_delete: :delete.

Generate a migration to create the resource in the database, and run it:

```
$ mix ash.codegen create_notifications
$ mix ash.migrate
```

Creating notifications on demand

The Notification resource is all set up, so now we can turn to creating notifications when an album is created, to let the followers know about it. We can do this with a change, in the Tunez.Music.Album resource. It's really a side effect

of creating an album, and we want the change to run whenever *any* create-type action is called, so we'll add the change as a global change in the changes block.

We'll tuck all of the logic away in a separate change module, so it's a one-liner to add the new change to the Tunez.Music.Album resource:

10/lib/tunez/music/album.ex

```
changes do
  change Tunez.Accounts.Changes.SendNewAlbumNotifications, on: [:create]
  # ...
```

The Tunez.Accounts.Changes.SendNewAlbumNotifications module doesn't exist yet, but we know that it should be a module that uses Ash.Resource.Change,[1] and defines a change/3 callback with the code to run.

10/lib/tunez/accounts/changes/send_new_album_notifications.ex

```
defmodule Tunez.Accounts.Changes.SendNewAlbumNotifications do
  use Ash.Resource.Change

  @impl true
  def change(changeset, _opts, _context) do
    # Create notifications here!
    changeset
  end
end
```

Because it's included in actions in the Album resource, the changeset will have the details of the album being created, including the artist_id. We can use that ID to fetch the artist and all of its followers, and then use a *bulk action* to create a notification for each follower.

Running Actions in Bulk

We briefly talked about bulk actions when we saw a surprising BulkResult while unfollowing artists, but now we're very intentionally going to write one.

Imagine that Tunez is super popular, and one artist now has thousands or even tens of thousands of followers. If they release a new album, our SendNewAlbumNotifications change module would be responsible for creating *tens of thousands* of Notification records in the database. We *could* do that one at a time, iterating over the followers and calling a create action for each, but that would be *really* inefficient.

Instead, we can call the create action *once*, with a list of records to be created. Ash will run all of the pre-database logic such as validations and changes for

1. https://hexdocs.pm/ash/Ash.Resource.Change.html

each item in the list, but then intelligently batch the insert of multiple records into as few database queries as possible.

Any action can be made into a bulk action by changing what data is passed to the action, so we can test bulk behaviour with our existing actions.

Testing Artist bulk create

We know how to create *one* record, by calling either a code interface function, or Ash.create.

```
iex(1)> # user is a loaded record with role = :admin
iex(2)> Tunez.Music.create_artist(%{name: "New Artist"}, actor: user)
INSERT INTO "artists" («fields») VALUES ($1,$2,$3,$4,$5,$6,$7) RETURNING
«fields» [«data»]
{:ok, #Tunez.Music.Artist<...>}
```

We can use the same code to run bulk actions, by changing what we pass in. Instead of a single map, we can call the code interface with a *list* of maps.

```
iex(3)> data = [%{name: "New Artist 1"}, %{name: "New Artist 2"}]
[...]
iex(4)> Tunez.Music.create_artist(data, actor: user)
INSERT INTO "artists" («fields») VALUES ($1,$2,$3,$4,$5,$6,$7),
($8,$9,$10,$11,$12,$13,$14) RETURNING «fields» [«data for both records»]
%Ash.BulkResult{
  status: :success, errors: [], records: nil,
  notifications: nil, error_count: 0
}
```

Boom, two records inserted with a single database query.

If you want to be really explicit about running actions as bulk actions, Ash has functions like Ash.bulk_create that can only be run with lists of data. These are what we've used in the seed files for Tunez, in priv/repo/seeds/.

```
iex(5)> Ash.bulk_create(data, Tunez.Music.Artist, :create, actor: user)
%Ash.BulkResult{
  status: :success, errors: [], records: nil,
  notifications: nil, error_count: 0
}
```

By default, you don't get a lot of information back in a bulk result, not even the records being created or updated. This is for performance reasons — if you're inserting a lot of data, it's a lot of work to get the results back from the database, build the structs, and return them to you! You'll get the errors, if any occurred, but if the bulk result has the status :success then you can safely assume that all of the records were successfully created.

The default behaviour *can* be customized, via any of the options listed for Ash.bulk_create[2] (or bulk_update, or bulk_destroy.) These same options, such as return_records? to actually get the created/updated records back, can also be used for code interface functions by including them under the bulk_options option key.

```
iex(11)> Ash.bulk_create([%{name: "Test"}], Tunez.Music.Artist, :create,
         actor: user, return_records?: true)
%Ash.BulkResult{status: :success, records: [#Tunez.Music.Artist<...>], ...}

iex(12)> Tunez.Music.create_artist([%{name: "Test"}], actor: user,
         bulk_options: [return_records?: true])
%Ash.BulkResult{status: :success, records: [#Tunez.Music.Artist<...>], ...}
```

Bulk actions are really powerful and let you get things done very efficiently. They'll speed up what we want to do — inserting possibly-many notifications for users about new albums.

Back to album notifications

Now that we have a grip on bulk actions, we can write one in our SendNewAlbumNotifications change module.

We'll use an after_action hook[3] as part of the change function, to ensure that we only create notifications once, after the album is successfully created. The callback in the hook has access to the newly-created album, so we can use it to load up all of the artist's followers, and then build maps of data to bulk create.

10/lib/tunez/accounts/changes/send_new_album_notifications.ex

```
def change(changeset, _opts, _context) do
  Ash.Changeset.after_action(changeset, fn _changeset, album ->
    album = Ash.load!(album, artist: [:follower_relationships])

    album.artist.follower_relationships
    |> Enum.map(fn %{follower_id: follower_id} ->
      %{album_id: album.id, user_id: follower_id}
    end)
    |> Ash.bulk_create!(Tunez.Accounts.Notification, :create)

    {:ok, album}
  end)
end
```

The after action callback can return either an {:ok, album} tuple or an {:error, changeset} tuple. If it returns an error tuple, the record (in this case, the album)

2. https://hexdocs.pm/ash/Ash.html#bulk_create/4
3. https://hexdocs.pm/ash/Ash.Changeset.html#after_action/3

won't be created after all — the database transaction will be rolled back, and the whole action will return the changeset with the error.

Before you can test out the new code, we need to define the :create action on the Tunez.Accounts.Notification resource! The bulk action will try to call it, but then error because the action doesn't exist. The action will be pretty simple, the map of data contains the two foreign keys and they can be accepted directly:

10/lib/tunez/accounts/notification.ex

```
actions do
  create :create do
    accept [:user_id, :album_id]
  end
end
```

Now that we have *actions* on the resource, we should also add *policies* for it. For something like this, which will only ever be done as a system action and never called from outside the domain model, we can forbid it from all access.

10/lib/tunez/accounts/notification.ex

```
defmodule Tunez.Accounts.Notification do
  use Ash.Resource,
    # ...
    authorizers: [Ash.Policy.Authorizer]

  policies do
    policy action(:create) do
      forbid_if always()
    end
  end

  # ...
```

Our internal SendNewAlbumNotifications module can still call it though, so we'll bypass that authorization check there.

10/lib/tunez/accounts/changes/send_new_album_notifications.ex

```
album.artist.follower_relationships
|> Enum.map(fn %{follower_id: follower_id} ->
  %{album_id: album.id, user_id: follower_id}
end)
|> Ash.bulk_create!(Tunez.Accounts.Notification, :create,
  authorize?: false
)
```

Now you can test out the new code! If you follow an artist in your Tunez app, and then create a new album for that artist, you should see a new notification being created in the server logs when you save the album:

```
[debug] HANDLE EVENT "save" in TunezWeb.Albums.FormLive
  Parameters: %{"form" => %{"cover_image_url" => "", "name" => "Test Album
```

```
  Name", "year_released" => "2025"}}
INSERT INTO "albums" («fields») VALUES («values») RETURNING «fields»
[«album_uuid», "Test Album Name", «now», «now», «artist_uuid», nil,
«creator_uuid», «creator_uuid», 2025]
«queries to load the album's artist's followers»
INSERT INTO "notifications" ("id","album_id","inserted_at","user_id") VALUES
($1,$2,$3,$4) [«uuid», «album_uuid», «now», «user_uuid»]
```

If you have multiple users in your database that all follow that artist, you may even see multiple notifications being created at once!

Optimizing big queries with streams

We can go even further with improving the after_action callback in the SendNewAlbumNotifications change module. We're efficiently *inserting* all the notifications we create using a bulk action, but we still have to *load* all of the follower relationships from the data layer first. For a popular artist with a lot of followers, this could be pretty slow, and take up a lot of memory.

We can turn to *streaming* the data from the data layer — fetching the follower data in batches, processing each batch, and then using the bulk create to insert all the notifications. For larger datasets, it's *significantly* more memory efficient than loading all the records at once, because Elixir and Ash don't have to keep track of all the data.

All read actions can return their results via streaming, so instead of using Ash.load to load the relationship data we need, we'll create a new read action to run directly. We're loading follower relationships, which are Tunez.Music.ArtistFollower records, so the new action will go on the ArtistFollower resource to read all records for a given artist ID.

10/lib/tunez/music/artist_follower.ex

```
read :for_artist do
  argument :artist_id, :uuid do
    allow_nil? false
  end

  filter expr(artist_id == ^arg(:artist_id))
  pagination keyset?: true, required?: false
end
```

The action accepts an artist_id to fetch follower relationships for, and uses it in a filter. The action has to support pagination, for streaming — but we can mark it as required? false so we don't have to use it.

Set up a code interface function for the action in the Tunez.Music domain:

10/lib/tunez/music.ex

```
resource Tunez.Music.ArtistFollower do
```

```
    # ...
➤   define :followers_for_artist, action: :for_artist, args: [:artist_id]
  end
```

Then we can rewrite the after_action callback to call our new action, with the stream?: true option for streaming.

10/lib/tunez/accounts/changes/send_new_album_notifications.ex

```
def change(changeset, _opts, _context) do
  changeset
  |> Ash.Changeset.after_action(fn _changeset, album ->
➤   Tunez.Music.followers_for_artist!(album.artist_id, stream?: true)
➤   |> Stream.map(fn %{follower_id: follower_id} ->
➤     %{album_id: album.id, user_id: follower_id}
➤   end)
    |> Ash.bulk_create!(Tunez.Accounts.Notification, :create,
      authorize?: false
    )

    {:ok, album}
  end)
end
```

The code doesn't look a whole lot different! Instead of loading the data with Ash.load and then iterating over it with Enum.map/2, we call our new Tunez.Music.followers_for_artist function and then iterate over it with Stream.map/2. We don't have to change the bulk create — it can already work with streams. This new version should run in roughly the same amount of time, but be a lot kinder on your server's memory usage.

Because Ash uses Ecto under the hood, your database queries are subject to Ecto's limits, such as the query timeout[4] configuration. By default, an Ash bulk create can take at most 15 seconds. That's enough time to process a *lot* of records, but if you need more time, you can either extend the timeout or implement the functionality differently.

For example, you could create a generic action[5] that runs an SQL query to insert the notifications records directly.

Now that notifications are being created, we should update the UI of the web app to show them to users. We'll look at this in two parts — loading and showing the notifications on page load, and then updating them in real time as new notifications are sent.

4. https://hexdocs.pm/ecto/Ecto.Repo.html#module-shared-options
5. https://hexdocs.pm/ash/generic-actions.html

Showing Notifications to Users

The notification bell in the main navigation bar is implemented in its own LiveView module, rendered from the user_info function component in the TunezWeb.Layouts module:

10/lib/tunez_web/components/layouts.ex

```
<%= if @current_user do %>
  {live_render(@socket, TunezWeb.NotificationsLive, sticky: true,
    id: :notifications_container)}
  <% # ... %>
```

This NotificationsLive liveview is marked as *sticky*, meaning it won't need to reload as we navigate around and use the app. It'll stay open on the server, alongside the page liveview we're currently using such as TunezWeb.Artists.IndexLive, and each new page liveview will connect to it to render it.

Inside TunezWeb.NotificationsLive, in lib/tunez_web/live/notifications_live.ex, there's a whole template set up to render notifications. However, the notifications are currently hardcoded as an empty list, in the mount/3 function.

To render the real notifications for the logged-in user, we need a read action on the Tunez.Accounts.Notification resource. From the outside, we might name the code interface function something like notifications_for_user, and call it like this:

10/lib/tunez_web/live/notifications_live.ex

```
def mount(_params, _session, socket) do
  notifications = Tunez.Accounts.notifications_for_user!(
    actor: socket.assigns.current_user
  )
  {:ok, assign(socket, notifications: notifications)}
end
```

This code interface function then needs to be defined, in the Tunez.Accounts domain module in lib/tunez/accounts.ex:

10/lib/tunez/accounts.ex

```
resources do
  # ...

  resource Tunez.Accounts.Notification do
    define :notifications_for_user, action: :for_user
  end
end
```

And then finally the action can be added to the Tunez.Accounts.Notification resource!

10/lib/tunez/accounts/notification.ex

```
actions do
  # ...
```

```
    read :for_user do
      prepare build(load: [album: [:artist]], sort: [inserted_at: :desc])
      filter expr(user_id == ^actor(:id))
    end
  end
```

The read action includes a filter to only select notifications for the actor calling the action. We'll also load all of the related data we need to render the notifications, and sort them so that the latest notifications appear first.

We need to add a policy that covers the action, so who should be able to run it? Well, anyone — with the filtering in the action, any authenticated user should be able to run it and they'll only ever get back their *own* notifications. So we can use the built-in actor_present policy check[6] again.

10/lib/tunez/accounts/notification.ex

```
policies do
  policy action(:for_user) do
    authorize_if actor_present()
  end

  # ...
end
```

Everything looks all good, right? But refreshing the page will give a bit of a surprise — the NotificationsLive liveview doesn't have the current_user stored in the socket! Why not??

A brief detour into LiveView process shenanigans

This gotcha is caused by a quirk in how LiveView works, in particular, sticky child liveviews. When a liveview is initially created, it only has access to data stored in the *session*, and this is the same for both liveviews mounted in your router, and any nested liveviews.

Most of the time this doesn't matter, because we're only rendering one liveview and being done with it. In this case, however, it does. The page liveviews, such as TunezWeb.Artists.ShowLive, get the current user via an on_mount callback set up in your app's router with ash_authentication_live_session. This callback will read the authentication token stored in the session, load the correct user record, and store it in socket.assigns.

So TunezWeb.NotificationsLive will need to load its *own* copy of the current user. We can use one of AshAuthenticationPhoenix's helpers for this. When we

6. https://hexdocs.pm/ash/Ash.Policy.Check.Builtins.html#actor_present/0

installed it, it created the TunezWeb.LiveUserAuth module in our app, with some on_mount callbacks for us to use.

The on_mount(:current_user) callback is the one we're after. It uses the same AshAuthenticationPhoenix functionality as ash_authentication_live_session, to read the authentication token (which our liveview *does* have access to) and use it to load and assign the current user.

After all that explanation, the fix turns out to be one line of code — calling that on_mount callback at the top of the NotificationsLive liveview:

10/lib/tunez_web/live/notifications_live.ex

```
defmodule TunezWeb.NotificationsLive do
  use TunezWeb, :live_view

➤ on_mount {TunezWeb.LiveUserAuth, :current_user}

  def mount(_params, _session, socket) do
    # ...
```

It *is* possible for parent and child liveviews to "share" assigns,[7] but this is a performance optimization and shouldn't be relied on. And it doesn't work at all for sticky liveviews — these are totally de-coupled from their calling liveview.

If you refresh your app, to recompile the changes to NotificationsLive and re-initialize it, you should now see the notification (or notifications) you created earlier when testing SendNewAlbumNotifications! No-one will be able to ignore that red pinging notification bell. Excellent.

OK, tell me about that new album… and then go away

It's great to know what new albums there are, and clicking on the notification will redirect to the details of the album on the artist profile. But the notification doesn't disappear after clicking on it! That's *really* annoying. If a user clicks on a notification, it should be dismissed (deleted).

Currently, if you click on a notification, it sends the "dismiss-notification" event to the NotificationsLive liveview, via a JS.push. There's an event handler set up to process that event, but it's empty.

The notifications are all stored in the socket, so to process the notification that the user clicked on, we can find it based on its ID. Then we need a new action on the Notification resource, to actually dismiss it. The code is a little

7. https://hexdocs.pm/phoenix_live_view/Phoenix.Component.html#assign_new/3-when-connected

bit verbose here, but we can tidy it up when we make this liveview more real-time.

10/lib/tunez_web/live/notifications_live.ex

```
def handle_event("dismiss-notification", %{"id" => id}, socket) do
  notification = Enum.find(socket.assigns.notifications, &(&1.id == id))

  Tunez.Accounts.dismiss_notification(
    notification,
    actor: socket.assigns.current_user
  )

  notifications = Enum.reject(socket.assigns.notifications, &(&1.id == id))
  {:noreply, assign(socket, notifications: notifications)}
end
```

The new action doesn't have to be anything fancy — we only want to delete the notification. We could soft-delete the notification by setting a dismissed_at timestamp and then showing "read" notifications differently to "unread" ones, but for Tunez, a standard default destroy is fine.

10/lib/tunez/accounts.ex

```
resources do
  # ...

  resource Tunez.Accounts.Notification do
    define :notifications_for_user, action: :for_user
    define :dismiss_notification, action: :destroy
  end
end
```

10/lib/tunez/accounts/notification.ex

```
actions do
  defaults [:destroy]

  # ...
```

The destroy action needs authorization, so we can use the relates_to_user_via/2 built-in check to ensure that users can only dismiss their own notifications:

10/lib/tunez/accounts/notification.ex

```
policies do
  # ...

  policy action(:destroy) do
    authorize_if relates_to_actor_via(:user)
  end
end
```

This works pretty well — when you click on a notification, you get to see the album details *and* the notification will disappear, along with the annoying red ping (if it was the only notification in the list).

There's just one thing left to do. At the moment, users will only get new notifications when they reload the page, due to our sticky liveview only fetching notifications in the mount/3 callback. They need to find out about new albums *immediately*! It's a matter of internet street cred… I mean, life and death!!

Updating Notifications in Real-Time

For real-time goodness, our NotificationsLive liveview needs some way of finding out when new Notification records are created. For this, we can turn to a publish/subscribe mechanism, also known as *pub/sub* (or pubsub). The Notification resource will *publish* updates for every action that we set it up for, with a given *topic* name, and then the liveview can *subscribe* to that topic to receive the updates and update the page with the new notification details.

Phoenix has a pubsub adapter[8] built into it, for use with features like channels[9] and presence.[10] Ash also comes with a pubsub notifier,[11] that works with Phoenix's pubsub (or any other pubsub) to let us set up systems that can respond to events in real time.

Setting up the publish mechanism

To enable pubsub broadcasting for notifications, we first need to configure it as a *notifier* in the Notification resource. Notifiers[12] are way to set up side effects for your actions, but only those really lightweight kind of side effects that if an error occurs and it doesn't go through, its not a big deal. We call these kinds of side effects, "at most once" side effects because that's how often they will occur.

Pubsub is a perfect use case for this — if something goes wrong and a publish message is missed, that's okay because it's only an enhancement to get the data on the page a little bit quicker. The Notification record is still created in the database, and the user will see it when they reload the page.

To configure notifiers for a resource, add the notifiers option to use Ash.Resource:

10/lib/tunez/accounts/notification.ex

```
defmodule Tunez.Accounts.Notification do
  use Ash.Resource,
    otp_app: :tunez,
    domain: Tunez.Accounts,
```

8. https://hexdocs.pm/phoenix_pubsub/
9. https://hexdocs.pm/phoenix/channels.html
10. https://hexdocs.pm/phoenix/presence.html
11. https://hexdocs.pm/ash/Ash.Notifier.PubSub.html
12. https://hexdocs.pm/ash/notifiers.html

```
    data_layer: AshPostgres.DataLayer,
    authorizers: [Ash.Policy.Authorizer],
➤   notifiers: [Ash.Notifier.PubSub]
```

Once that's done, we can use the pub_sub DSL[13] in the resource to enable publishing broadcasts whenever specific actions are run. Because we're in a Phoenix app, our Phoenix Endpoint module (TunezWeb.Endpoint) will handle all pubsub functionality.

In our specific case, we want to broadcast a message whenever the create action is run. As we care about notifications on a per-user basis (like we implemented a notifications_for_user function!), we'll use a topic for messages that includes the :user_id topic template.[14] Ash will replace this with the actual user ID that the notification is for.

10/lib/tunez/accounts/notification.ex

```
defmodule Tunez.Accounts.Notification do
  # ...

  pub_sub do
    prefix "notifications"
    module TunezWeb.Endpoint
    publish :create, [:user_id]
  end
```

This will broadcast messages with a topic like notifications:<user_id>, whenever a Notification is created. Awesome! How do we know if its working, though? Where do the messages *go*? Before we set up the subscriber, it would be great to be able to see what's going on and if our messages are actually getting sent.

Debugging pubsub publishing

Pubsub can be tricky to get working properly, because it feels like magic going on behind the scenes. To make it a bit easier, while building your pubsub setup we'd strongly recommend enabling Ash's pubsub debugging, which logs when messages are sent and their content.

You can do this with the following config in your config/dev.exs file:

10/config/dev.exs

```
config :ash, :pub_sub, debug?: true
```

Then, if you start an iex session and manually create a new Notification, you'll be able to see the pubsub message being broadcast:

13. https://hexdocs.pm/ash/dsl-ash-notifier-pubsub.html#pub_sub
14. https://hexdocs.pm/ash/Ash.Notifier.PubSub.html#module-topic-templates

```
iex(1)> Ash.Changeset.for_action(Tunez.Accounts.Notification, :create,
        %{user_id: «user_uuid», album_id: «album_uuid»})
        |> Ash.create!(authorize?: false)
INSERT INTO "notifications" ...

[debug] Broadcasting to topics ["notifications:«user_uuid»"] via
TunezWeb.Endpoint.broadcast

Notification:

%Ash.Notifier.Notification{resource: Tunez.Accounts.Notification, domain:
Tunez.Accounts, action: %Ash.Resource.Actions.Create{name: :create,
primary?: true, description: nil, error_handler: nil, accept: ...
```

Ash has built a Ash.Notifier.Notification struct (not to be confused with a Tunez.Accounts.Notification!) and that's what will be sent out in the broadcast.

If we try to generate pubsub messages in iex by creating a new album for an artist that has at least one follower, though, we *won't* see that pubsub debug message printed:

```
iex(2)> Tunez.Music.create_album!(%{artist_id: «artist_uuid»,
        name: "New Album", year_released: 2022}, actor: user)
INSERT INTO "albums" ("id","name","inserted_at","updated_at", ...
«SELECT query to load the artist followers»
INSERT INTO "notifications" ("id","album_id","inserted_at", ...
#Tunez.Music.Album<...>
```

So we've done something in the create action of Tunez.Music.Album, that is preventing pubsub messages from being created or sent.

Putting our detective caps on

A good place to start debugging would be where the Notifications are being created — in the SendNewAlbumNotifications module. It uses a bulk action, to generate notifications for all of an artist's followers at once. If we create notifications using a bulk action in iex, do we get pubsub messages sent?

```
iex(3)> Ash.bulk_create([%{user_id: «user_uuid», album_id: «album_id»}],
        Tunez.Accounts.Notification, :create, authorize?: false)
INSERT INTO "notifications" ("id","album_id","inserted_at","user_id") ...
%Ash.BulkResult{notifications: nil, ...}
```

We don't! Notifications aren't generated by default for bulk actions, just like records aren't returned, also for performance reasons. To configure a bulk action to generate and auto-send any notifications, you can use the notify? true option of Ash.bulk_create.[15]

```
iex(4)> Ash.bulk_create([%{user_id: «user_uuid», album_id: «album_id»}],
```

15. https://hexdocs.pm/ash/Ash.html#bulk_create/4

```
➤         Tunez.Accounts.Notification, :create, notify?: true)
INSERT INTO "notifications" ("id","user_id","album_id","inserted_at") ...
[debug] Broadcasting to topics ["notifications:«user_uuid»"] via
TunezWeb.Endpoint.broadcast

Notification:

%Ash.Notifier.Notification{resource: Tunez.Accounts.Notification, ...}

%Ash.BulkResult{...}
```

Perfect! If we add this same option to our SendNewAlbumNotifications change function, Ash generate and send notifications for us:

10/lib/tunez/accounts/changes/send_new_album_notifications.ex

```
def change(changeset, _opts, _context) do
  changeset
  |> Ash.Changeset.after_action(fn _changeset, album ->
    Tunez.Music.followers_for_artist!(album.artist_id, stream?: true)
    |> Stream.map(fn %{follower_id: follower_id} ->
      %{album_id: album.id, user_id: follower_id}
    end)
    |> Ash.bulk_create!(Tunez.Accounts.Notification, :create,
➤     authorize?: false, notify?: true
    )

    {:ok, album}
  end)
end
```

After making that change, if you recompile (or restart iex) you'll see the notification being sent when creating an album.

```
iex(5)> Tunez.Music.create_album!(%{artist_id: «artist_uuid»,
        name: "Son Of New Album", year_released: 2025}, actor: user)
INSERT INTO "albums" ("id","name","inserted_at","updated_at", ...
«SELECT query to load the artist followers»
INSERT INTO "notifications" ("id","album_id","inserted_at","user_id") ...
[debug] Broadcasting to topics ["notifications:«user_uuid»"] via
TunezWeb.Endpoint.broadcast

Notification:

%Ash.Notifier.Notification{resource: Tunez.Accounts.Notification, domain: ...}

#Tunez.Music.Album<...>
```

If we restart our web app (to get the updated debug config), and then create an album in the UI for an artist that has a follower, we'll also see the notification being sent in the web server logs. Perfect.

Limiting data sent within notifications

These Ash.Notifier.Notification structs are pretty big — there's a lot of information in there about the action that was called, the changeset that was built, the record that was created, the actor, and so on. All of that information will be broadcast as part of the pubsub message, which can be a bit unwieldly.

It can also be a security issue. Because any liveview, with any authenticated user, can subscribe to a pubsub topic and receive broadcasts, we don't have any way of restricting the data in the notification to stop the recipient seeing data they aren't authorized to see.

To prevent issues, Ash lets you define a transform[16] function for your pubsub notifications. Each publish or publish_all line can have its own transform function, or you can define one for the entire pub_sub block. This function receives the full Ash.Notifier.Notification struct, and lets you either strip data from it, or rebuild it in a way that makes sense for your app.

The behavior of pubsub transform functions may change in Ash 4.0 — see this GitHub issue[17] for details.

Our planned implementation for our NotificationsLive liveview will be pretty simple. If it gets a message that there's a new notification, it will reload the user's notification list. So the broadcast we send doesn't need many details in it; a subset of data from the created Tunez.Accounts.Notification will be sufficient.

10/lib/tunez/accounts/notification.ex

```
pub_sub do
  prefix "notifications"
  module TunezWeb.Endpoint

  transform fn notification ->
    Map.take(notification.data, [:id, :user_id, :album_id])
  end

  publish :create, [:user_id]
end
```

Configuring a transform won't change the debug information printed in the server logs, but it will change the data in the actual broadcast message.

16. https://hexdocs.pm/ash/dsl-ash-notifier-pubsub.html#pub_sub-transform
17. https://github.com/ash-project/ash/issues/1792

Setting up the subscribe mechanism

Compared to the publish side of the mechanism, subscription is a lot more straightforward! Ash doesn't provide any helpers to handle subscribing to pubsub topics, or processing the messages — it doesn't *need* to, they're none of its concern. Ash's responsibilities end when the messages are sent, and it's our liveview's responsibility to listen and react.

To start listening for the pubsub messages in our NotificationsLive liveview, update the mount/3 function and *subscribe* to the topic we defined for our messages:

10/lib/tunez_web/live/notifications_live.ex

```
def mount(_params, _session, socket) do
  # ...

  if connected?(socket) do
    "notifications:#{socket.assigns.current_user.id}"
    |> TunezWeb.Endpoint.subscribe()
  end

  {:ok, assign(socket, notifications: notifications)}
end
```

The endpoint module will then send a message to the liveview when a pubsub broadcast is received, which has to be received with a handle_info callback. We don't have any handle_info callbacks set up, but we can add a simple one which will reload the list of notifications when any messages are received:

10/lib/tunez_web/live/notifications_live.ex

```
def handle_info(%{topic: "notifications:" <> _}, socket) do
  notifications = Tunez.Accounts.notifications_for_user!(
    actor: socket.assigns.current_user
  )

  {:noreply, assign(socket, notifications: notifications)}
end
```

We'll do some pattern matching to make sure we're getting the right type of messages, but that's it. If we were going to receive more than one type of message, or we needed to do something more involved with the specific message we received, the logic here would have to be a bit more complex. But for Tunez, where we're only receiving one type of message and don't expect users to have a million notifications at the same time, it's fine!

And if you inspect and print out the received pubsub message, you'll see it's very trim, taut, and terrific:

```
[(tunez 0.1.0) lib/tunez_web/live/notifications_live.ex:67:
TunezWeb.NotificationsLive.handle_info/2]
message #=> %Phoenix.Socket.Broadcast{
```

```
  topic: "notifications:«user_uuid»",
  event: "create",
  payload: %{id: «uuid», album_id: «album_uuid», user_id: «user_uuid»}
}
```

Deleting notifications

There's one last spanner to throw in the real-time works — what happens when a notification is *deleted*? This could happen if you have Tunez open on both your computer and your phone, and you click a notification on one device — the other would still show that you have a notification to view.

Or a new album could be added, but then deleted. The Tunez.Accounts.Notification resource is set up with a database reference to delete notifications if the album is deleted, but users will still see that notification until their notification list is refreshed.

Let's look at how we can address these issues, for a smooth experience.

Broadcasting delete messages

Like we set up pubsub for the Tunez.Accounts.Notification create action, we can also use pubsub to broadcast calls to the destroy action. This will resolve one of our issues, when a user has the app open in two places at once. Deleting a notification on one device will send a pubsub message to the other.

10/lib/tunez/accounts/notification.ex

```
pub_sub do
  # ...

  publish :create, [:user_id]
➤ publish :destroy, [:user_id]
end
```

Because we've used the exact same pubsub topic, notifications:<uuid>, we don't even need to change our NotificationsLive implementation. Receiving a destroy message should behave exactly the same as a create message, and reload the list of notifications.

We *can* clean up a little bit of our "dismiss-notification" event handler logic in NotificationsLive though. We don't need to manually remove the dismissed notification from the list when a user clicks on one — the pubsub process will handle that for us!

10/lib/tunez_web/live/notifications_live.ex

```
def handle_event("dismiss-notification", %{"id" => id}, socket) do
  notification = Enum.find(socket.assigns.notifications, &(&1.id == id))

  Tunez.Accounts.dismiss_notification(
    notification,
```

```
      actor: socket.assigns.current_user
    )

➤   {:noreply, socket}
  end
```

This won't resolve our second issue, though. Because the database reference handles the deletion entirely within the database, our app doesn't know that it's even taken place, and can't notify anyone!

Cascading deletes in code

When we covered deleting related resources on page 51, we covered two approaches — specifying the ON DELETE behaviour on the database reference, to do the delete within the database, or doing it in code by using a cascade_destroy for the related records.

So far, we've always opted for the database method because it's much more efficient. However, this is a case where we have business logic to run (sending pubsub messages) when we delete the related records, so we'll have to switch to the less-performant approach.

To remove the automatic deletion of notifications when their related album is deleted, update the database reference to :album in the Tunez.Accounts.Notification resource, to remove the on_delete: :delete.

10/lib/tunez/accounts/notification.ex

```
postgres do
  # ...

  references do
    reference :user, index?: true, on_delete: :delete
➤   reference :album
  end
end
```

Generate a migration for the database change, and then run it:

```
$ mix ash.codegen remove_notification_album_cascade_delete
$ mix ash.migrate
```

This breaks the ability to delete albums that have any related notifications waiting to be seen, but we'll fix that now!

We need to manually destroy related notifications, when the destroy action of the Tunez.Music.Album resource is called. At the moment, that action is defined as a default action. And we don't even *have* a relationship defined between albums and notifications! We'll have to add that first, it should be a has_many

relationship as there can be many notifications for different users, all for the same album:

10/lib/tunez/music/album.ex
```
relationships do
  # ...

  has_many :notifications, Tunez.Accounts.Notification
end
```

Then we can write a new destroy action, removing the default implementation from the defaults list:

10/lib/tunez/music/album.ex
```
actions do
  defaults [:read]

  destroy :destroy do
    primary? true
    change cascade_destroy(:notifications, return_notifications?: true,
      after_action?: false)
  end

  # ...
```

This new action uses the cascade_destroy[18] built-in change to read the related Tunez notifications for an album, and call the default destroy action for them all as a bulk action (in a before_action hook, by declaring after_action?: false). It looks kind of weird because we also need to configure cascade_destroy to get the Ash notifications back for pubsub broadcast, with return_notifications?: true. This is similar to when we bulk created Tunez notifications in SendNewAlbumNotifications. So many notifications flying around!

before_action or after_action?

cascade_destroy works by calling a bulk destroy action for the related resources, either in a after_action function hook (the default) or a before_action function hook. Which one you choose determines the order of the destroys: which should come first, deleting the main resource (the album) or deleting the related resources (the notifications)?

When using an after_action hook, the main resource will be deleted first (and then the related records, in the hook). But we know that you can't delete a record that has references pointing to it, it generates a foreign key violation error — that's why we used ON DELETE CASCADE in the first place! This is when we'd need to use the deferrable option on the database reference, as mentioned in the cascade_destroy documentation. deferrable: :initially will *defer* that foreign key check until the end of the transaction. As long as all the related records are also deleted before the end of the transaction, everything is A-OK.

18. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html#cascade_destroy/2

> This won't suit *all* cases, though. We haven't looked at policies yet, but as we'll now be deleting notifications via calling a destroy action, we'll have to update the policies for that action to include this new use case. What if the rules we want to encode, depend on the main resource? If the main resource has already been deleted, the policies might not behave as intended.
>
> If we switch our cascade_destroy to use a before_action instead by specifying after_action?: false, then all of these issues will go away!

That last paragraph also hints at another last change we need to make — we need to *read* the related notifications before we can delete them. Our Tunez.Accounts.Notification resource doesn't have any basic read action, so we can quickly add one.

10/lib/tunez/accounts/notification.ex

```
actions do
  defaults [:read, :destroy]
  # ...
```

The read action also needs an authorization policy, or it won't be allowed to run. Who should be allowed to access this action? The only users who should be bulk-managing notifications like this, to read all notifications for an album to delete them, are the users who are deleting the album.

Our policies around album deletion currently look like this:

10/lib/tunez/music/album.ex

```
policies do
  bypass actor_attribute_equals(:role, :admin) do
    authorize_if always()
  end

  # ...

  policy action_type([:update, :destroy]) do
    authorize_if expr(^actor(:role) == :editor and created_by_id == ^actor(:id))
  end
end
```

You *could* copy and paste these policy checks into policies for the read and destroy actions for Notifications, but if the logic changes, we'd have to remember to update it in all three places. Instead, we'll extract the logic into a calculation, so we can reuse it across dfferent resources.

Calculations: not just for user-facing data

There's probably some confused noises being made right now. So far we've seen calculations primarily for presenting data in a more user-friendly way — formatting seconds as minutes and seconds, or telling users how long ago their favorite albums came out (I'm sorry, Master of Puppets is *how old???*) But there's nothing saying that's *all* they can be used for.

Extracting reusable expressions is a perfectly valid use of a calculation. We can extract the logic of the Album policy check into a calculation, which we'll call `can_manage_album?`:

10/lib/tunez/music/album.ex

```
calculations do
  # ...

  calculate :can_manage_album?,
            :boolean,
            expr(
              ^actor(:role) == :admin or
              (^actor(:role) == :editor and created_by_id == ^actor(:id))
            )
end
```

We can then update the Album policy to use the new calculation:

10/lib/tunez/music/album.ex

```
policy action_type([:update, :destroy]) do
➤   authorize_if expr(can_manage_album?)
end
```

Ash will automatically load the calculation when the policy is run. We have existing tests for this policy in `test/tunez/music/album_test.exs`, so you can run them to double-check that the functionality still behaves as expected.

Now we can write the policies for the Notification resource — the `read` and `destroy` actions can be run if the album can be managed by the current user. This is a second policy check for the `destroy` action, so it can go in the same policy. If either of the checks pass, then the action will be authorized.

10/lib/tunez/accounts/notification.ex

```
policies do
➤   policy action(:read) do
➤     authorize_if expr(album.can_manage_album?)
➤   end

    # ...

    policy action(:destroy) do
➤     authorize_if expr(album.can_manage_album?)
      authorize_if relates_to_actor_via(:user)
```

```
  end
```

This is why we couldn't use an after_action when calling cascade_destroy. We'd first be deleting the album, and then the notifications, but the authorization policy for notifications depends on the deleted album! Oops.

And now we can delete albums again! Try following an artist, creating an album for them, seeing the notification, and then deleting the album. The notification will disappear! Magic!

I have some bad news for you, though…

We've successfully solved the issue around deleting albums, but introduced another problem. Now we can't delete *artists* that have albums that have notifications, for the same reason we couldn't delete albums that had notifications!

These kinds of changes can ripple through an app, and unfortunately, there's not much that can be done about it. We can either keep going and add cascade_destroy for albums when deleting artists, or we can undo our cascade_destroy changes for albums and accept that users may occasionally see phantom notifications for albums that have been deleted.

We'll opt to add cascade_destroy for albums in Tunez, but we'll do it *super* quickly. Updating the reference from albums back to artists:

10/lib/tunez/music/album.ex

```
defmodule Tunez.Music.Album do
  # ...

  postgres do
    # ...

    references do
➤     reference :artist, index?: true
    end
  end
```

Replacing the default destroy action with a new one to call cascade_destroy:

10/lib/tunez/music/artist.ex

```
defmodule Tunez.Music.Artist do
  # ...

  actions do
    defaults [:create, :read]

    destroy :destroy do
      primary? true
      change cascade_destroy(:albums, return_notifications?: true,
        after_action?: false)
```

```
    end

    # ...
```

And lastly, generating and running the migration to update the database reference:

```
$ mix ash.codegen remove_album_artist_cascade_delete
$ mix ash.migrate
```

And now we can delete artists again. Phew!

There are a few tests in the Tunez repo that cover this behaviour, to ensure that it works as expected. You can enable them in test/tunez/music/artist_test.exs and test/tunez/music/album_test.exs.

We Need to Talk About Atomics

There's one last topic we want to cover before we finish up — it's not related to what we've covered so far in this chapter, but we think it's pretty important. We've discussed little snippets about atomics all throughout this book, but haven't gone into much detail beyond the fact that they're used for running logic in the data layer, instead of in our app. What does that really *mean*, though?

Imagine we wrote a feature that counts the number of followers an artist has, and manually updates the number whenever someone follows or unfollows them. It might look something like this, in the Tunez.Music.Artist resource:

```
update :follow do
  change fn changeset, _opts ->
    count = Ash.Changeset.get_attribute(changeset, :follower_count)
    Ash.Changeset.change_attribute(changeset, :follower_count, count + 1)
  end
end
```

(Obviously we'd never write this code because we know that change modules are much better, but for demonstration purposes.)

You're browsing Tunez, and you see your favorite artist. Hey, they have 472 followers! Not bad! But you haven't followed them yet. Better do that now.

While you're reading through their album list and making sure there's no typos, three *other* users follow that artist. They now have 475 followers! But when you click the star icon to follow them, what happens? The follower count goes *down* to 473! Wait, what?

This action isn't *atomic* — it runs in code, and uses the data that's loaded in memory. When you loaded the artist record, the value was at 472, so the follow

action dutifully added one more and wrote the value 473 to the data layer. Oops.

What you really *meant* in the action was "add one to the current count, whatever it is", and the data layer is the source of truth for what the count is right now. Not when you loaded the page, but *right now*. When we say "run the change logic in the data layer", its instructing the data layer to increment the current value, not store an arbitrary new value that we calculated elsewhere. In SQL, it's this:

```
-- Not atomic!
UPDATE artists SET follower_count = 473 WHERE id = «uuid»;

-- Atomic! :)
UPDATE artists SET follower_count = follower_count + 1 WHERE id = «uuid»;
```

Basically, if we're using data from the resource in a change function/module, we really want to be doing it atomically, or consciously decide not to do so.

What does this mean for Tunez?

When we wrote code to store previous names for an artist on page 53, we used both the `name` and `previous_names` attributes that existed when we loaded the `Tunez.Music.Artist` record. This creates a race condition like our `follower_count` example — if two users edited the same artist's name at the same time, the name that the first user used would not be added to the second user's submitted `previous_names`, it would be lost unless we rewrote the logic of the change to be atomic.

It's the same thing with using 'manage_relationship' when updating records on page 184: Ash currently needs to use the existing records, to be able to figure out what data needs to be created, updated, or deleted, so this can't be run atomically.

Our final case of the 'MinutesToSeconds' change module on page 201 *can* be made atomic, but perhaps not in the way you think. The module doesn't use data from the Track record when it was loaded — it only uses data that was submitted from the Album form. So it's *already* atomic, but we need to tell Ash that the change can be used that way.

To enable atomic behaviour for a change module, we need to implement the `atomic/3` callback in the `Tunez.Music.Changes.MinutesToSeconds` module. It doesn't need to do anything fancy, it can call the existing `change/3` function, and return the changeset in an `:ok` tuple:

```
defmodule Tunez.Music.Changes.MinutesToSeconds do
  # ...
```

```
  @impl true
  def atomic(changeset, opts, context) do
    {:ok, change(changeset, opts, context)}
  end
end
```

We might think twice in this case about removing require_atomic? false from the update action of the Tunez.Music.Track resource, though. Because this action is called as part of a manage_relationship function call, Ash will try to atomically update each track *just in case the data in memory is out of date*, leading to a classic n+1 query problem.[19]

What if we really wanted to store the follower count, though?

Sometimes you really do need incrementing fields. Or maybe our tradeoff for UpdatePreviousNames is unacceptable, and it *has* to be atomic. For cases like this, there are a few different approaches.

Ash provides an atomic_update/3 built-in change function,[20] that can be used for cases like the incrementing follower count. Using atomic_update, it could be written like this:

```
update :follow do
  change atomic_update(:follower_count, expr(follower_count + 1))
end
```

This uses an expression to define what needs to change, and that's something the data layer knows how to deal with. As a bonus, it's even shorter and easier to read than the inline change version!

For more complex logic, there's the atomic/3 callback[21] that can be implemented in change modules. We've seen how we can use it to mark known-good changes as atomic, but it can do a whole lot more. It has a pretty intimidating typespec — it can return a *lot* of different things. We've seen one example already: an :ok tuple means "this changeset is already atomic, nothing else needs to be done".

If a change *can* be done atomically, we can return an :atomic tuple, with :atomic as the first element, and a map of atomic changes to make as the second element. The follower_count example could be written like this, if we wanted to extract it to a module for reuse:

```
@impl true
```

19. https://www.pingcap.com/article/how-to-efficiently-solve-the-n1-query-problem/
20. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html#atomic_update/3
21. https://hexdocs.pm/ash/Ash.Resource.Change.html#c:atomic/3

```
def atomic(_changeset, _opts, _context) do
  {:atomic, %{follower_count: expr(follower_count + 1)}}
end
```

Ash actually has an increment change built-in,[22] and you can see exactly how it's implemented.[23] The atomic version can't be written as a single expression, due to the overflow_limit option, but it always returns a single atomic tuple.

It does have one thing we haven't seen before — the use of the atomic_ref[24] function. This will get a reference to the attribute, *after any other changes from our action have been made*, ready to use in an expression. We'll see what this really means, when we rewrite the UpdatePreviousNames change.

Rewriting UpdatePreviousNames to be atomic

This change runs in the Artist update action, to record all previous versions of an artist's name attribute.

10/lib/tunez/music/changes/update_previous_names.ex

```
def change(changeset, _opts, _context) do
  Ash.Changeset.before_action(changeset, fn changeset ->
    new_name = Ash.Changeset.get_attribute(changeset, :name)
    previous_name = Ash.Changeset.get_data(changeset, :name)
    previous_names = Ash.Changeset.get_data(changeset, :previous_names)

    names =
      [previous_name | previous_names]
      |> Enum.uniq()
      |> Enum.reject(fn name -> name == new_name end)

    Ash.Changeset.force_change_attribute(changeset, :previous_names, names)
  end)
end
```

It *can* be written atomically using in a single expression, if we lean on PostgreSQL array operations and functions[25] to do some of the heavy lifting. To embed SQL directly in an expression, we can use a fragment.[26]

```
expr(
  fragment(
    "array_remove(array_prepend(?, ?), ?)",
    name, previous_names, ^atomic_ref(:name)
  )
)
```

22. https://hexdocs.pm/ash/Ash.Resource.Change.Builtins.html#increment/2
23. https://github.com/ash-project/ash/blob/main/lib/ash/resource/change/increment.ex
24. https://hexdocs.pm/ash/Ash.Expr.html#atomic_ref/1
25. https://www.postgresql.org/docs/current/functions-array.html
26. https://hexdocs.pm/ash/expressions.html#fragments

This expression uses both name and atomic_ref(:name), so we can really see the difference. name is the name as it exists in the database, but atomic_ref(:name) is the name with any changes we've made as part of this action. If we're changing an artist's name from "Hybrid Theory" to "Linkin Park", name will refer to the "Hybrid Theory" value, but atomic_ref(:name) will refer to the "Linkin Park" value.

And if previous_names is "{Xero}", a PostgreSQL array with one element (the string "Xero"), then this expression will boil down to the SQL fragment:

```
array_remove(array_prepend('Hybrid Theory', '{Xero}'), 'Linkin Park')
```

If the name is being updated, this will add the old name and remove the new name from the list (as the new name is no longer a "previous" name!), which is the same logic that we wrote in the non-atomic version of the change.

The expression needs one more thing before it can be used in an atomic/3 callback. The fragment returns an array in PostgreSQL land, but Ash has no way of knowing that. To tell Ash that yes, this expression is okay and will always return a valid value, we need to wrap *it* in an :atomic tuple.

All together, the atomic/3 callback looks like this:

10/lib/tunez/music/changes/update_previous_names.ex

```
@impl true
def atomic(_changeset, _opts, _context) do
  {:atomic,
    %{
      previous_names:
        {:atomic,
        expr(
          fragment(
            "array_remove(array_prepend(?, ?), ?)",
            name, previous_names, ^atomic_ref(:name)
          )
        )}
    }}
end
```

The change is now fully atomic! We can remove the change/3 version of the change from the UpdatePreviousNames module, and we can remove the require_atomic? false from the update action of the Tunez.Music.Artist resource,because the whole action can now be run atomically.

10/lib/tunez/music/artist.ex

```
update :update do
  accept [:name, :biography]
  change Tunez.Music.Changes.UpdatePreviousNames
end
```

Wrapping Everything Up

And that's it! Congratulations! You've made it to the very end, and there was no monster at the end of the book. Well done!

We hope you've enjoyed this tour through the foundations of the Ash framework, learning how it can help speed up your development and write more efficient apps using declarative design. You've built a full application (it's tiny but mighty!) and you should be proud!

What should you do next? Build some more apps! It's one thing to follow a guide, carefully crafted to explain all the concepts as you go along, but its quite another to build something of your own. You could build one of the ideas we considered building in this book — a web forum, a Q&A site, a project management tool, a time tracker. Or you could add some more features to Tunez. Some cool ideas we wanted to cover (but ran out of space!) are things like:

- Moving notification generation to a background job, using Oban and AshOban
- Using pubsub to live-update each artist's follower count in the catalog
- The ability to rate albums and artists
- Extracting that rating ability to an extension, so it can be re-used and you can rate all of the things!!111!!!!
- User friendships, and getting notifications when your friends rate an album
- A more advanced artist search, using `AshPhoenix.FilterForm`
- Genre tags for artists, to show how a tagging UI could be built

The possibilities are literally endless. And that's just for Tunez, you probably have a lot of awesome ideas of your own, which we'd love to see you build! If you're keen to learn more about Ash, or just want to chat, join us in the Ash community.[27] We're a friendly bunch!

And above all else, have fun, and good luck!

27. https://ash-hq.org/community

Bibliography

[Jur15] Saša Jurić. *Elixir in Action*. Manning Publications Co., Greenwich, CT, 2015.

[LM21] Andrea Leopardi and Jeffrey Matthias. *Testing Elixir*. The Pragmatic Bookshelf, Dallas, TX, 2021.

[TD24] Bruce A. Tate and Sophie DeBenedetto. *Programming Phoenix LiveView*. The Pragmatic Bookshelf, Dallas, TX, 2024.

[TV19] Chris McCord, Bruce Tate and José Valim. *Programming Phoenix 1.4*. The Pragmatic Bookshelf, Dallas, TX, 2019.

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head on over to https://pragprog.com right now, and use the coupon code BUYANOTHER2025 to save 30% on your next ebook. Offer is void where prohibited or restricted. This offer does not apply to any edition of *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With up to a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-author/ today to learn more and to get started.

Thank you for your continued support. We hope to hear from you again soon!

The Pragmatic Bookshelf

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/ldash
Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up-to-Date
https://pragprog.com
Join our announcement mailing list (low volume) or follow us on Twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy
https://pragprog.com/news
Check out the latest Pragmatic developments, new titles, and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*