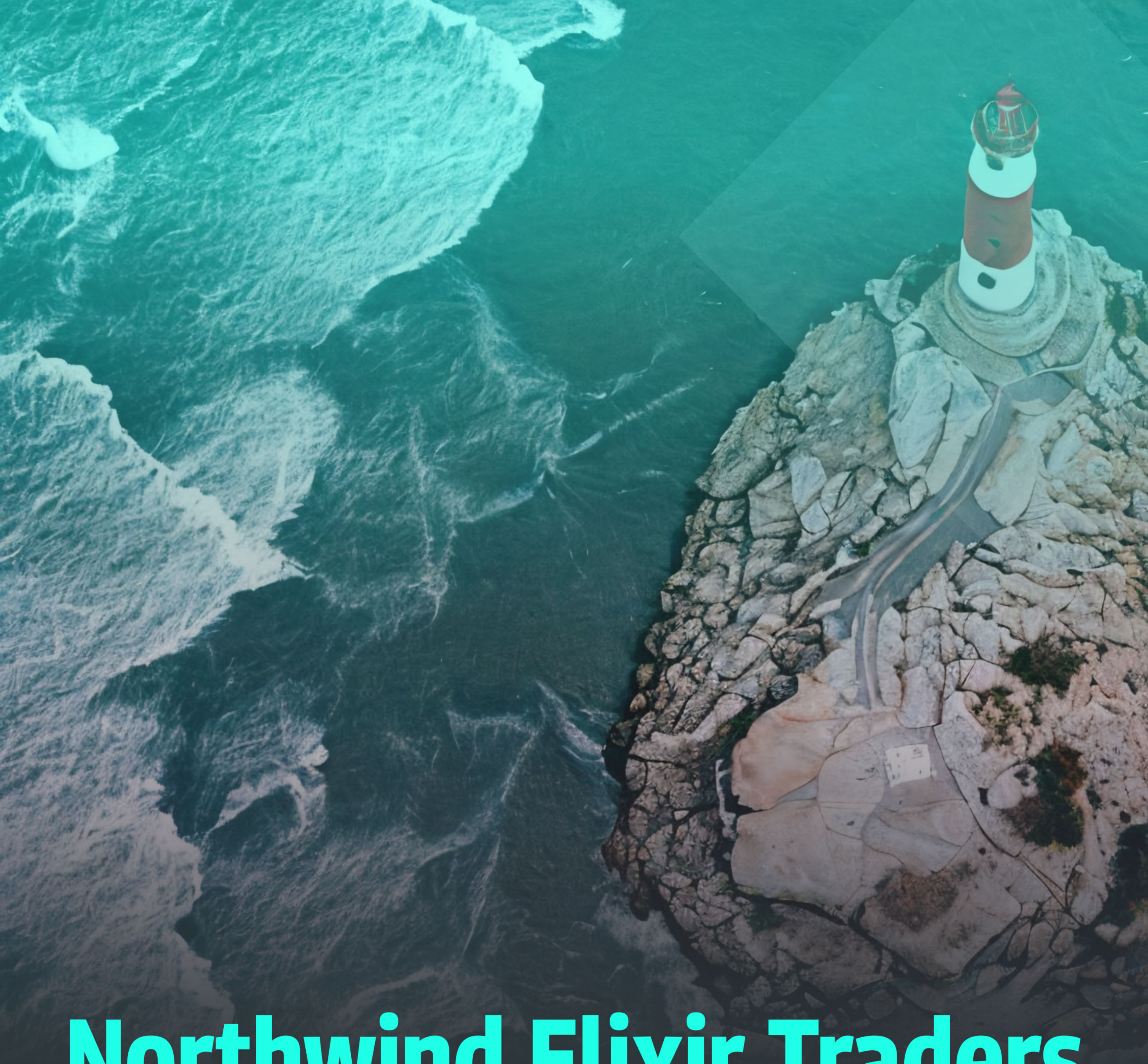

Northwind Elixir Traders

Learn Elixir and database modeling with Ecto and SQLite, all in one project

Isaak Tsalicoglou

OVERBRING LABS

Northwind Elixir Traders

Learn Elixir and database modeling with Ecto and SQLite, all in one project

Isaak Tsalicoglou

This book is available at https://leanpub.com/northwind-elixir-traders

This version was published on 2025-03-14

Tweet This Book!

Please help Isaak Tsalicoglou by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Do you remember the ”Northwind Traders” sample database that shipped with Microsoft Access way back when? Well, this book I just bought uses it to teach databases with Elixir and Ecto :)

The suggested hashtag for this book is #northwindelixirtraders.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#northwindelixirtraders

Contents

Before we begin

Northwind Elixir Traders

Learn Elixir and database modeling with Ecto and SQLite, all in one project

by **Isaak Tsalicoglou**, **OVERBRING Labs**, in Athens, Greece

This is version 1.0 of the book, dated 2025-03-14.

Announcements, comments and discussions related to this book can be found on the book's forum thread on elixirforum.com.

* * *

For more information, contact the author on ElixirForum, on LinkedIn, or by email.

About this book

This is a book for curious people. Exercise your Elixir skills and learn about working with databases by using Ecto's migrations, changesets, and more to reimagine a classic database that millions around the world have used before for learning. By not pursuing the "happy path", this immersive, exploratory, memorable, project-based tutorial will help you to fearlessly tackle real-world projects with Elixir and Ecto.

> *"This rare gem of a technical book is a must-read for anyone wanting to build practical skills in modeling databases with Elixir and Ecto. But it's not only about coding—it also emphasizes real-world problem-solving, learning by doing, and tackling development roadblocks head-on with confidence, in a relentless pursuit of understanding. Truly, an invaluable resource for everyone tired of cookie-cutter tutorials and theoretical fluff!"*–**Petros Papapanagiotou, PhD / Head of Development**

> *"With this excellent book about modeling data with Ecto and SQLite, I was up and running quickly without snags! I learned how SQLite compares with PostgreSQL and how to work between their differences. Despite Northwind Traders being a toy database, the modeling we do in the book is top notch. After thoroughly modeling the Northwind Traders DB with Ecto, we then craft a gang of queries to glean excellent business "insights" within the data. The author has a strong understanding and passion in this area of development and it shows. Northwind Elixir Traders is an excellent read and a phenomenal Elixir resource."*–**Benjamin Gandhi-Shepard, Designer / Developer at Solvm**

Step back into the nostalgic realm of 90s' database exploration and learning, with a modern twist! Remember the "Northwind Traders" database from the early days of MS Access? It's back and ready for a re-imagining in the dynamic world of the Elixir programming language, its Ecto database layer, and SQLite.

Join me on a learning exploration as we transform this familiar database into the captivating "Northwind Elixir Traders" database through migrations, table alterations, schema definitions, CRUD operations, queries, changesets, associations and all key features of Ecto that you need to know, to be able to use Ecto productively in real-world applications. Plus much, much more that will stretch your Elixir muscles and give you a good workout of its key concepts and out-of-the-box functionality.

Who is this book for?

This book has been written for anyone who wants to learn Elixir by practicing on an ambitious project, and anyone who already possesses some Elixir programming skill and basic knowledge of relational databases, and wants to understand Ecto better to a degree that will allow them to model any kind of data with these technologies.

You will learn about incrementally converting a database schema to Ecto structures and writing the code necessary to manage the data therein, gradually (and retroactively, and iteratively) refining the database model with constraints, changesets and associations and helper functions, and working with dynamic repositories to import, validate and sanitize data, and making simple and complex queries to generate insights.

What knowledge is required to get started?

- A basic understanding of Elixir's `Kernel`, in particular types of variables, flow control, the `Enum` module, how to define modules and functions, and how to write functional pipelines.
- A basic understanding of relational databases and how they are structured (tables, columns and their types, primary and foreign keys).
- Rudimentary skill in SQL and its syntax.
- Rudimentary skills in running commands in a shell.

What will we not cover?

I assume that your interest in this book means that you already know how to install Elixir with your operating system's native or package manager or with something like asdf, how to install the `sqlite3` binary, and how to use an editor and IEx, so none of those topics will be covered. We will also not cover testing with the `ExUnit` unit-testing framework, even though we will be testing our code and queries with code snippets in IEx, and with a handful of functions in the modules we develop.

What kind of problems will you learn to solve?

On a high level, you will learn to solve problems such as:

- Thinking through an Entity Relationship Diagram (ERD) and gradually transforming it into Elixir/Ecto code.
- Making decisions about data types when having to deal with existing data that isn't necessarily "clean".
- Dealing with SQLite's limitations that make us stray off the beaten path of Ecto's excellent documentation.
- Automating the importing of data in bulk from another database and verifying its data quality.
- Thinking about real-world boundary conditions when improving a business database like this one.

Especially the latter is informed by my experience in running, digitalizing, and growing a B2B industrial equipment trading business, for which in recent years I have implemented various small pieces of software that are related to the modeling of a database like this one, and much more.

What else will you learn?

My primary goal for you is to understand how to use Ecto productively. However, through numerous "side-quests" we will exercise Elixir skills that take you *far* beyond just learning to utilize Ecto. We will use whatever is required to accomplish our objective. This includes `Enum.reduce/3` and `Enum.reduce_while/3`, `use` vs. `import`, basic streams, basic stuff from the `Application`, `Module`, `Supervisor` and `Task` modules, and even some parts of Erlang's library, such as the `httpc` HTTP client to make `GET` requests (that we use in this book to fetch CSV data from an online source), or `timer` for timing the performance of our queries. We will even implement a Depth-First Search algorithm to determine the order in which tables should be imported from the original database. If something could aid us in reaching our goal, we will try it out.

Therefore, my secondary goal for you with this book in terms of Elixir programming skills is to enter with basic knowledge and exit with strengthened skills as a side effect. This includes refactoring code to make everything more composable. Nothing works as well as hands-on practice and repetition for learning, and this book is like an Elixir gym.

My tertiary goal with Northwind Elixir is to bolster your confidence and expand your research and development skills by helping you acquire an engineer's exploratory mindset and grit in problem-solving. There are numerous references and side discussions and even mini-essays/asides scattered across most chapters that aim to transmit to you some parts of my experience in the R&D of products of different kinds. Therefore, you will also learn about:

- the mentality and tools of working through a complex problem with pragmatism and an eye on managing complexity,
- the gradual implementation of a pedagogical example in a programming language that I absolutely love, and
- the engineer's mindset of dealing fearlessly with obstacles as they arise and aiming to deeply understand what's going on, before rolling up your sleeves to deal with them.

How is this book different to other books on databases and Ecto?

We will explore the subject matter with a hands-on, outcome-focused approach of gradually modeling an existing pedagogically-mature database schema and by venturing on side-quests to gain a solid understanding of Ecto's "gotchas" and the ramifications of using Ecto with SQLite instead of the default choice of PostgreSQL.

Different to a book that takes a more "completionist" approach, Northwind Elixir Traders starts with a specific objective in mind and helps you work towards it by using only the elements of Ecto that are required for the sub-tasks of achieving the objective, without pursuing the coverage of the entirety of Ecto's functionality as a reference manual. Databases and Ecto, combined, are an immense field of knowledge. If you have basic skills in those topics and want to get some practice in Elixir too, this is the book for you.

This book fosters truly "grokking" Ecto through practice and experimentation, and by exploring why things sometimes don't work as one would expect and diving deep into the documentation to help you figure things out. As such, *Northwind Elixir Traders* is the right choice of book for those who prefer applied learning that covers the essentials to get the job done, before diving deeper into every nook and cranny of Ecto with a more comprehensive learning resource or into Ecto's online documentation (which we will be referring to consistently).

Why SQLite, of all databases out there?

SQLite has been chosen in order to "spice things up" compared to Ecto's out-of-the-box default of using PostgreSQL, thanks to its ever-growing popularity and impressive, far better than just good-enough capabilities for some applications that your might want to build, such as Phoenix APIs or Phoenix LiveView websites with primarily read-heavy functionality. However, the book is written so that your acquired understanding of Ecto is transferable to using any other RDBMS that Ecto has an adapter for. If you want to use PostgreSQL instead of SQLite after finishing this book, you will be well prepared.

What makes this book special?

This *isn't* your typical Ecto tutorial—instead of pursuing the usual "happy path" reflected in its official documentation and "Getting Started" guide, we'll dive into the complexities and nuances of database implementation for business purposes, offering a refreshing exploration that goes beyond the ordinary and the expected.

Our adventure begins by embracing the simplicity of SQLite over the conventional choice of PostgreSQL. This deliberate trade-off in favor of simplicity against feature-completeness introduces challenges and roadblocks that will enrich your learning experience as we navigate the intricacies of schema design, migrations, primary and foreign keys, constraints and validation functions, relational structures between the different tables, importing data from existing databases, and executing queries that return meaningful insights.

Discover the joy of experimentation as we encounter unexpected hurdles and exercise our Elixir skills in figuring out why things don't work as expected, and how to still achieve our goal, regardless. Through these challenges and related "side-quests", you'll gain invaluable insights into problem-solving and critical thinking with Elixir and Ecto. In the process, you will also learn more about the highly versatile and popular SQLite database.

Embrace the engineer's ethos of fearless exploration and systematic problem-solving as we delve deeper into Ecto's capabilities, focusing on the art of reading documentation and adapting to unforeseen obstacles. With each chapter, you'll elevate your understanding of Ecto, empowering yourself to tackle real-world projects with confidence and finesse, instead of with copy-paste operations and "hopes and prayers" that it all works out in the end.

For a curious Elixir newcomer getting into the world of building databases for a microservice, a backend, or a Phoenix or Phoenix LiveView app, *Northwind Elixir Traders* confidently promises an immersive learning experience that's both enriching and unforgettable.

Our journey along the book's chapters

- **Chapter 1**, **The application and the repo**: We jump straight into setting up the project and explain why we need a supervision tree (and what it is).
- **Chapter 2**, **Creating the database schema**: We analyze the ERD of Northwind Traders, model the first table, persist, delete and query data (an appetizer of what's to come in **Chapter 14**), and discuss primary keys.
- **Chapter 3**, **Refining the table schema**: We extend our table with migrations and hit (and overcome) the first snags of SQLite's limitations compared to more feature-rich databases.
- **Chapter 4**, **Introducing constraints**: We learn about constraints and SQLite's limitations with them, and about database portability, changesets and validation rules, and validation errors.
- **Chapter 5**, **Changesets in modules with schemas**: We learn to implement custom validation functions and multiple changeset functions as we gradually build the modules that model our database.
- **Chapter 6**, **Uniqueness constraints**: We utilize unique indexes and constraints, gain new insights about Ecto and SQLite, and learn about Ecto migrations and migration rollbacks.
- **Chapter 7**, **Basic table associations**: We learn about SQLite's type affinity, about numeric fields and Northwind Traders' dirty data, about one-to-many associations with a foreign key, and also implement a custom validation function to overcome SQLite's limitations regarding foreign-key constraints.
- **Chapter 8**, **Associations with Ecto.Schema**: We take a break from coding with an interlude on exploration vs. exploitation in learning and in solving problems, we learn about the N+1 query problem and Ecto's preloading, about one-to-many associations and `has_many`, and we use all that to improve existing associations.
- **Chapter 9**, **Casting and putting associations**: We explore casting and putting an association, creating new records, and casting the reverse association so that we can access the data of associated database records through a struct/schema.
- **Chapter 10**, **Importing data from a dynamic repository**: We learn how to use a dynamic Ecto repo to connect to the original database on demand, and implement a module for importing data from the original in bulk and in an automated manner and validating that everything was done correctly.
- **Chapter 11**, **Modeling further tables, and data cleansing**: We go on a a final march through a smorgasbord of pending tasks; we learn about mapping our next working steps with SIPOC, about dealing with phone numbers, about importing, validating and converting countries' data from an external CSV source over HTTP `GET`, and then work through the rest of the tables (except for the **OrderDetails** table).
- **Chapter 12**, **Modeling a join table**: We model the centerpiece of the ERD, i.e. the join/association table that pulls everything together, and investigate Ecto's `many_to_many`.
- **Chapter 13**, **Cleaning up**: Having completed the modeling of all tables and imported all data from the original database, we review whether everything works well and identify some loose ends and tie them up, including revisiting old assumptions about our automation in determining the order in which we import tables and improving our functions with a Depth-First Search algorithm.
- **Chapter 14**, **Insights from data with queries**: We learn about the kinds of insights that a business might want to extract from the dataset. We also learn about the difference between Elixir vs. database functions, running simple queries and queries that span multiple tables, and creating composable queries. We stumble on an earlier assumption about the data type of price data and write migrations to make things right. We briefly explore parallelizing our queries, and write many more functions to extract the insights we need. We also implement filtering query results by date using keyword-list options.
- **Chapter 15**, **More and more-advanced queries**: Beyond looking at the entire dataset, we learn about queries that allow us to look at subsets of the data with windows and partitions. We implement functions that make it possible to link any schema of the Northwind traders ERD to *any* other schema, and use these functions in one-dimensional grouping queries, and in developing expanding-window and sliding-window queries partitioned by up to three primary-key values. We refactor extensively, with the goal of making our queries composable and capable of dealing with sub-queries too.
- **Chapter 16**, **Very complicated window queries**: After conquering window queries with partitions, ordering and frames that use SQL fragments with `ROWS`, we implement complicated window queries that circumvent to only severe limitation of SQLite thus far: its lack of support for `INTERVAL`.

- **Chapter 17, Towards "Northwind Elixir Traders 2.0"**: With everything finished, we say farewell to our job well done by listing ideas for improvements and next steps for our Northwind Elixir Traders and other Elixir applications that might build upon it.

Software versions used

- **Elixir**: version 1.18.2, installed using the asdf version manager. Initially, the code in the book was developed with version 1.14.0 from the `elixir` package for Debian 12. Either will work to progress through the book, except for places where `Enum.sum_by/2` is used, which is available starting from version 1.18.0, and the functions `DateTime.before?/2`, `DateTime.after?/2`, `Date.before?/2`, and `Date.after?/2`, which are available starting from version 1.15.0. In any case, I strongly recommend installing Elixir 1.18.2 (the current version at the time of this writing), since the new typing features also make debugging easier.
- **Ecto**: version 3.12.5.
- **Ecto SQLite3**: version 0.18.1 from Hex.pm.
- **SQLite**: version 3.40.1 from the `sqlite3` package for Debian 12.

To my knowledge, nothing above is specific to Debian 12 or Linux, and you should be able to work through this book on any popular operating system and CPU architecture of your choice. Since early 2025, and while polishing the book for its 100%-ready release, I have "replayed" the book and all code snippers executed in IEx with the versions mentioned above. To the date of this release, the replay has concluded for the entirety of the book. Should you face any issues. Should anything not work as expected up to that chapter, kindly contact me so that I can publish a new release with corrections.

License of the code in this book

All code that we will write throughout this book is licensed under the Apache-2.0 open-source license. You are free to use, modify, and distribute the code, provided that you include proper attribution and comply with the terms of the license. What this means for you, in short:

You can:

- **Use** the code for personal, commercial, or open-source projects.
- **Modify** and customize the code however you like.
- **Distribute** the original or modified code.
- **Use it in proprietary (closed-source) software** as long as you include the required notices.

You cannot:

- **Hold the author liable** if something goes wrong (no warranties).
- **Use the original project's logo or name** without permission.

However, you must:

- **Keep the license notice** in all copies of the code.
- **State any changes** you make if you modify the code.
- **Include a copy of the license** when redistributing.

Other aspects:

- Regarding the patent grants aspects of Apache-2.0: the book does not include patented technology.
- The git repository with cleaned-up code has been published on GitHub.
- The Apache-2.0 license **does not** extend to the text of the book.

Typographic conventions

This book is long and contains a lot of code and IEx output. It would have ended up way longer without the following conventions, established in order to better utilize the space on every page:

- Commands shown in IEx have been written less for readability, and more with the goal of enabling you to copy and paste them into IEx.
- The lines of output of commands in IEx have been wrapped, and are displayed differently in code blocks, compared to how they will appear on screen. This is especially true for output related to changesets.
- The formatting of the code has also been made more concise / horizontal, instead of pretty with vertical pipelines, and is neither everywhere the same as what `mix format` will generate, not "best practice" (nor is the organization of code between the `Joins` and `Insights` modules to be considered a good practice).
- Non-relevant IEx output and unchanged lines of code are replaced by an ellipsis (…).
- The names of the tables of the original Northwind Traders database are bolded; e.g., **Orders**.
- The names of the tables of the Northwind *Elixir* Traders database are lowercased, in a monospace font; e.g., `orders`.
- When mentioning a row/record/schema/module, its name is mentioned interchangeably with its struct; e.g., `%Order` or `Order`. These always refer to the Northwind *Elixir* Traders application.

Foreword

As I am writing this foreword, I have known Isaak for a little more than a quarter of a century. Since high school, Isaak and I shared the itch of technology and programming. Besides playing early video games on our PCs, we wanted to hack things to make them work, and to build, engineer, create! We were curious about and intrigued by the possibility of making computers work for us. And now, decades later, I have the privilege of introducing his book on Elixir—a language that captures that same spirit of curiosity and creation, allowing for many different ways to get a computer to do things for us.

At this point you might expect me to praise Isaak's years of battle-tested experience and deep expertise in all things related to databases, functional programming and Elixir. While he has indeed accumulated years of experience developing products and software, it is his endless appetite for knowledge and his methodical and passionate mindset that make what he has to say in this and other books worth reading.

Isaak is not the kind of person who will compromise by accepting mediocrity, shallow understanding, or bloated code. He attacks problems at their core, with passion and relentless pragmatism. He does not learn in order to be an expert or to follow the latest trend or hype (he has written books about that!). He learns because he is an engineer who thirsts for understanding and wants to solve problems without waste. Staying true to his high-school self, he wants to hack things to make them work, and to build, engineer, create. He wrote this book because he fell in love with Elixir and he wanted to absorb its potential, so that he can put it to productive use. This hands-on approach is what makes his work so impactful—not just as an engineer, but as an author as well.

This happens to be the kind of engineer I like to hire in my teams: not one who blindly memorizes patterns after having solved thousands of leetcode problems, and whose code is a minefield of technical debt that will one day explode in the team's face, but rather one who cares about the problem at hand and the impact of the solution—holistically, and down the road.

Such mentality is essential in an age in which software engineers are bombarded by a constant stream of new programming languages, frameworks and libraries, and by breathless hype regarding the looming (alleged?) threat of being made obsolete by super-efficient (though so far shallow-thinking) AI. Learning quickly and getting up to speed with the core capabilities of tools both new and old needs to be doable faster than ever before. However, the learning utilities at our disposal do not always suit this challenge well. We have documentation that cannot always keep up with development, plenty of principled theory-driven textbooks, repetitive tutorials with simplistic cookie-cutter patterns, blog posts with toy examples, LLMs that sometimes mislead by answering our questions with confident hallucinations, and crowd-sourced yet also strictly moderated Q&A platforms that end up promoting a kind of partisan groupthink in the popular "one true way" of tackling tasks.

The approach Isaak adopts in Northwind Elixir Traders stands out among these options. Beyond being about Elixir, Ecto, and SQLite, this book is also a case study on the train of thought of a dedicated learner who methodically breaks the problem down and focuses on pragmatic solutions without babbling, boasting, or hyping—and without feeling intimidated by things not working as intended the first time around. He wants to understand this topic in sufficient depth and breadth to accomplish his objective, and he is taking us along on this exploratory journey. It is not a smooth ride, like that of traditional technical books, where everything works out as if by magic, yet leaves you with the nagging feeling of countless unanswered *"what if?"* questions. At each step along the journey he tries things out based on his knowledge and intuition, sometimes faces the consequences of having been wrong, and then goes through the resulting pains and the challenges of getting things working, thus equipping you with the mindset of tackling development roadblocks head-on with confidence.

This book is inspired by the profound educational role that a database example of Microsoft Access played in the 90s and in the decades that followed. Back then, such examples provided an accessible playground for aspiring "computer professionals" *and* for curious, budding young developers like us—a way to learn without the modern conveniences of the Internet, by exploring an example thoughtfully developed by professionals. Isaak channels that same spirit of exploratory pedagogy into this book, capitalizing on the simplicity and timelessness of the Northwind Traders MDB file to explore Ecto's key features for accomplishing the objective, and to have productive fun coding in Elixir along the way.

* * *

Petros Papapanagiotou, PhD

Head of Development, Tech consultant

January 19th, 2025

Edinburgh, Scotland

Preface

Chapter 1: Our objective, the application and the repo

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Adding dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Creating the repository (the "repo")

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The need for a supervision tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 2: Creating the database schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Building the Northwind Elixir Traders database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Creating an empty migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Defining the content of the migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Deciding on column types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Putting it all together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Running the migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Preparing and using the data structure of a Category

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using the Category struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Wrangling the struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Failing to persist the struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Persisting data successfully

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Deleting records

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Basic queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Auto-incrementing integer primary keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Before moving on to the next chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 3: Refining the table schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

More options for the table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Looking for learning opportunities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Why `:utc_datetime`?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Digging deeper into the `create/1` macro

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Making changes, even if just because

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

To create is to destroy; to alter is to preserve

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Investigating the impact of our alterations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 4: Introducing constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Constraints with SQLite

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Considering database portability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Changesets: intended changes to data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Schemaless changesets, and casting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Changesets and validation rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Downstream of an invalid changeset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Turning validation errors into helpful messages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 5: Changesets in modules with schemas

In the previous chapter we learned about the fundamentals of changesets and validation functions without using the functionality offered by the `Ecto.Schema` module. It's time to extend that knowledge into changesets *within* a module that contains a schema definition. This will make it possible to use the Ecto repo and persist the data structures corresponding to the rows of various tables in the Northwind Elixir Traders database.

In the previous chapter we used schemaless changesets with the `Employee` struct, and in earlier chapters we already used the `schema/2` macro to define the schema of the `Category` struct. In order to be able to use `Ecto.Changeset` functions on a `%Category{}` without the use of a `practically_a_schema` tuple like the one we used in the previous chapter, we need to define a "changeset function" within the `Category` module. This changeset function will differ minimally to what we learned earlier; it will use the same `cast/3` and the various `validate_*` functions as before.

Its inputs will be an initial `%Category{}` struct and a `params` map containing the values that we want to cast and validate. No surprises there!

In `category.ex`, don't forget to add the line `import Ecto.Changeset` so that the `cast/3` and `validate_*` functions can be brought into the module's namespace and be used without the "`Ecto.Changeset.`" prefix.

Discussion: `use` vs. `import`

Or: **why `use Ecto.Schema` but `import Ecto.Changeset`? Why not `use` for both, or `import` for both?**

`use Ecto.Schema` calls the `__using__/1` macro of the `Ecto.Schema` module to perform the initial setup of the module as will be used by the `Category` module. Concretely, this means that it will:

1. Run `require Ecto.Schema` and thus provide access to the macros of `Ecto.Schema`, such as`schema/2` or `field/3`.
2. Inject (execute) the code contained in the `Ecto.Schema.__using__/1` macro, which (if you'll examine its source code linked above) initializes the attributes of the module, and sets values for some of them.

More information can be found in the documentation of the `Module.register_attribute/3` and `Module.put_attribute/3` functions, or `Module` in general, but this topic is way beyond the scope of this book.

Meanwhile, we do not need `Ecto.Changeset` to initialize any behavior by injecting any code into our `Category` module. We only want to gain access to its functions directly within our module's namespace. In fact, if you examine the module's source code you'll find out that `Ecto.Changeset.__using__/1` doesn't even exist, which is why this happens (unsurprisingly):

```
iex> use Ecto.Changeset
** (UndefinedFunctionError) function Ecto.Changeset.__using__/1 is undefined or private
    (ecto 3.12.5) Ecto.Changeset.__using__([])
```

Therefore, for `Ecto.Changeset`:

- We don't call `use`, because `Ecto.Changeset` doesn't even contain a `__using__/1` macro, anyway.
- We don't call `require`, because we want access to not just the macros of `Ecto.Changeset`.
- We don't call `alias`, because we don't simply want to avoid typing the `Changeset.` or any other prefix before using the functions (or macros) of `Ecto.Changeset`.
- We **do** call `import` so that we can simply write `cast(…)` instead of `Ecto.Changeset.cast(…)` (or `Changeset.cast(…)`, had we used `alias`).

Writing our first changeset function

In our earlier experimentation with defining constraints for the fields of the `categories` table in the `NorthwindElixirTraders.Category` module through the migration, we came up with a couple of rules:

- A value for `:name` is required, and its maximum length is 50 characters.
- A value for `:description` is optional, but should be at most 100 characters long, if defined.

If the lessons in the previous chapter have already become second nature, we can come up with the function pipeline of the changeset without writing a single line of code first:

1. `cast` the values in `params` and use both of the `:name` and `:description` fields as `permitted`.
2. `validate_required` on `:name`.
3. `validate_length` on the value of `:name` with `max: 50`.
4. `validate_length` on the value of `:description` with `max: 100`.

Instead of hard-coding the values 50 and 100 in the `validate_length/3` function calls within the changeset function, we can also extract those as module attributes `@name_mxlen` and `@desc_mxlen`.

We can also extract the list of atoms (or single atom) of the required fields into a `required` variable. We place this within the changeset function and don't set it as a module attribute, because we can define more than one changeset function within the module (more about this later in this chapter). Each changeset function will have a different purpose, and therefore also a different list of required fields. This is different to the maximum-string-length validations that, sensibly, safeguard data quality regardless of the changeset function used.

We can also add something that we have so far forgotten: timestamp fields! In previous chapters we investigated the use of the `timestamps/1` function of the `Ecto.Migration` module of Ecto SQL within the `CreateCategories` migration. This function created the `:inserted_at` and `:updated_at` columns to the `categories` table. Furthermore, we called that function with the `type: :utc_datetime` option in order to store the timestamp values in the `Etc/UTC` time zone (the default for SQLite storing dates as strings).

To add the corresponding fields to the schema of `Category`, we can use the `Ecto.Schema.timestamps/1` macro. To set UTC datetimes here, we'll set the `:type` option's value to `:utc_datetime`. Here's what `category.ex` could look like after implementing our first-ever changeset function within a module:

Figure 8.1. The Category module with our first changeset function

```elixir
defmodule NorthwindElixirTraders.Category do
  use Ecto.Schema
  import Ecto.Changeset

  @name_mxlen 50
  @desc_mxlen 100

  schema "categories" do
    field(:name, :string)
    field(:description, :string)

    timestamps(type: :utc_datetime)
  end

  def changeset(data, params \\ %{}) do
    permitted = [:name, :description]
    required = [:name]

    data
    |> cast(params, permitted)
    |> validate_required(required)
    |> validate_length(:name, max: @name_mxlen)
    |> validate_length(:description, max: @desc_mxlen)
  end
end
```

For clarity when comparing the code to the documentation of Ecto, I'm using the same argument names here, such as `data`, `params`, and `permitted`. It's also a personal convention for my Elixir apps, as it makes it easier to do a cursory check on the permitted and required fields of a changeset function. This is especially useful when you have many of those; I have occasionally wasted an embarrassing amount of minutes wondering why some value doesn't change when submitting a form in a Phoenix LiveView, only to realize that the field isn't part of `permitted`. Pulling the `permitted` and `required` lists to the top of the changeset function makes it easier to avoid banging your head against the wall unnecessarily.

Due to the `Ecto.Schema.timestamps/1` function call, the `%Category{}` struct now contains two extra fields: `:inserted_at` and `:updated_at`, i.e. the same names as those created in the migration by the `Ecto.Migration.timestamps/1` macro call. Note that the value of those two fields will be equal to `nil` when we create a `%Category{}` struct. As the documentation explains, their values will automatically be set to the current time *when inserting and updating values in a repository*.

```
iex> recompile
iex> %Category{} |> Map.keys()
[:__meta__, :__struct__, :description, :id, :inserted_at, :name, :updated_at]
```

We can finally try out different scenarios, such as starting with an empty `Category` struct, and using the changeset function to apply `nil` values to both of its fields with an empty map (`%{}`), that is: *make no changes at all.* Clearly, the changeset will be invalid (`valid?: false`) due to violating constraint #2 of the list above (a value for `:name` is required, as `:name` is in the `required` list):

```
iex> %Category{} |> Category.changeset(%{})
#Ecto.Changeset<
  action: nil, changes: %{}, errors: [name: {"can't be blank", [validation: :required]}],
  valid?: false, data: #NorthwindElixirTraders.Category<>
>
```

From now on, the `iex> recompile` line will be omitted from code blocks. Remember to call `recompile` in IEx after changing something in the codebase.

Another scenario: we start with a `Category` struct that already includes a valid name, then use a farcical German compound noun in the `params` map:

```
iex> too_long = "Geschwindigkeitsbegrenzungsinstandhaltungsundreparaturanlagenkomponenten"
iex> %Category{name: "Yolorific Yoloroonies"} |> Category.changeset(%{name: too_long})
#Ecto.Changeset<
  action: nil, changes: %{
    name: "Geschwindigkeitsbegrenzungsinstandhaltungsundreparaturanlagenkomponenten"},
  errors: [name: {"should be at most %{count} character(s)",
     [count: 50, validation: :length, kind: :max, type: :string]}],
  valid?: false, …
>
```

As expected:

- `:changes` contains the change of the `:name` field.
- `:errors` reports the invalid value of the `:name` field according to the `validate_length/3` constraint we applied.

Here's an interesting thing, though: does it matter that the initial value of `:name` ("Yolorific Yoloroonies"), satisfied the `validate_length/3` rule? What if we had used an invalid starting value, instead, and applied no changes, by using an empty map for `params`? Let's find out:

```
iex> %Category{name: too_long} |> Category.changeset(%{})
#Ecto.Changeset<action: nil, changes: %{}, errors: [],
 data: #NorthwindElixirTraders.Category<>, valid?: true, …>
```

`valid?: true`? *How?* ***Why?*** Actually, this is the expected behavior! Take a look at the documentation of `cast/4`:

> *Note that `cast/4` validates the types in the `params`, but not in the given `data`.*

Therefore, if you want to run an existing struct through the gauntlet of the changeset's pipeline, set `data` to an empty `Category` struct and provide the existing struct as the `params` argument, instead:

```
iex> params = %{name: too_long}
iex> %Category{} |> Category.changeset(params) |> Map.get(:valid?)
false
```

Finally, we can try the "happy path":

```
iex> %Category{} |> Category.changeset(%{name: "Yolorific Yoloroonies"})
#Ecto.Changeset<
  action: nil, valid?: true, changes: %{name: "Yolorific Yoloroonies"}, errors: [], …
>
```

For the reason mentioned earlier (the non-validating behavior of `cast/4` on its `data` argument), the following is *not* the "happy path", as it doesn't matter that the starting value of `:name` is valid; the resulting changeset has `valid?: true`, regardless:

```
iex> %Category{name: "Yolorific Yoloroonies"} |> Category.changeset(%{})
#Ecto.Changeset<action: nil, changes: %{}, errors: [], valid?: true, …>
```

Custom validation functions

We now know how to define a changeset function within a module that contains an Ecto schema. To be frank, we had it easy; with a mere two `:string` fields, the `Category` struct is as simple as it gets. However, other tables (such as the **Employees** table in the Northwind Traders ERD) include non-string and non-numeric fields. For example: the birth date of an employee.

Elixir's `Date` module provides functions to handle dates, but Elixir itself doesn't have a type that represents a date. A date is represented as a `%Date{}` struct or a `~D` sigil such as `~D[1992-03-16]` (which constructs a `%Date{}` struct). In the previous chapter we also saw a list of `Ecto.Changeset` functions with names beginning with `validate_`, but there was no such function with a name such as `validate_date` or `validate_time`.

So how, then, are we supposed to add constraints on the value of a schema field representing a date? Let's explore! We'll work on the `Employee` struct and bring it up to the state of the `Category` struct, with its own schema defined using`schema/2`, and with its own changeset function. To begin with, there are not many differences; two more string fields for `:photo` and `:notes`, plus the `:birth_date` field. This time, however, `:birth_date` will have the type `:date` instead of `:string`, since we later want to persist its value as a date using the Ecto repo.

Figure 8.2. The Employee module with a proper schema and a changeset function without birth-date validation

```
defmodule NorthwindElixirTraders.Employee do
  use Ecto.Schema
  import Ecto.Changeset

  @name_mxlen 50
  @notes_mxlen 500

  schema "employees" do
    field(:last_name, :string)
    field(:first_name, :string)
    field(:birth_date, :date)
    field(:photo, :string)
    field(:notes, :string)

    timestamps(type: :utc_datetime)
  end

  def changeset(data, params \\ %{}) do
    permitted = [:last_name, :first_name, :birth_date, :photo, :notes]
    required = [:last_name, :first_name, :birth_date]

    data
    |> cast(params, permitted)
    |> validate_required(required)
    |> validate_length(:last_name, max: @name_mxlen)
    |> validate_length(:first_name, max: @name_mxlen)
    |> validate_length(:notes, max: @notes_mxlen)
  end
end
```

As you can see, it's basically the same as the earlier definition of the `Category` module, with the most important difference being this line, where `:date` replaces the previous `:string` field type:

```
field(:birth_date, :date)
```

We can check that type validation works for that field by providing a string instead of a date:

```
iex> changeset_with_error = %Employee{} |> Employee.changeset(%{first_name: "Kumba", last_name: "Ya", birt\
h_date: "lulz"})
#Ecto.Changeset<
  errors: [birth_date: {"is invalid", [type: :date, validation: :cast]}], valid?: false, …
>
```

We can also check that it works by providing a date constructed with `Date.new!/4`, with the default calendar and an omitted 4th argument:

```
iex> changeset = %Employee{} |> Employee.changeset(%{first_name: "Kumba", last_name: "Ya", birth_date: Dat\
e.new!(1985, 5, 30)})
#Ecto.Changeset<
  changes: %{birth_date: ~D[1985-05-30], first_name: "Kumba", last_name: "Ya"},
  errors: [], valid?: true, …
>
```

Provided that the year, month and day-in-the-month arguments are valid (they are, here), `Date.new!(1985, 5, 30)` constructs a new `%Date{}` struct representing the date of May 30th, 1985. The same can be achieved with the `~D[1985-05-30]` sigil.

So, yes—the changeset function above only validates the *type* of the provided value through the `cast/4` call. It doesn't (yet) validate the value of this field; for example, we have specified no rules such as the birth date not being set in the future or earlier than a sensible date, such as 100 years ago from the moment of setting it. Let's build our own validation function for this purpose! What would be the requirements on it?

1. It must take a changeset as its first argument, and must also return a changeset, so that it can be chained with other `Ecto.Changeset` validation functions (including `cast/4`).
2. It must take the name of the field as its second argument, and validate its value according to certain rules that we will specify.
3. Before making any comparisons of the value to other specified dates, it must validate that the value of said field represents a date, i.e. conforms to the built-in `%Date{}` struct (which is handled by the built-in `Date` module).
4. It must check that the value of the field is not in the future or earlier than a certain time span compared to now; in other words, we are validating the implied age of the employee against a permitted age range.

5. It could (if we want to be thorough and generic) take a keyword list of options that will be used to parameterize the validation functionality; for example, with keywords like `:after` (later than), `:before` (earlier than), `:is` (exactly on a specific date).

The last requirement would mirror the options of the built-in `validate_number/3` function, which offers options such as `:less_than`, `:greater_than`, etc. A generic date validation function is already offered by the Ecto Commons package, within the `EctoCommons.DateValidator` module.

For now, we will not care about implementing a generic date validation function, but only one that satisfies the first 4 requirements, with option arguments for the age range. First, we'll check the changeset's `:errors` list to determine whether the value of `:birth_date` was a valid `:date`:

```
iex> changeset.errors[:birth_date]
nil
iex> changeset_with_error.errors[:birth_date]
{"is invalid", [type: :date, validation: :cast]}
```

We can pattern-match on the value of this snippet to decide what to do next. If the output is `nil`, then we proceed to checking that the value is within some specified range. Otherwise, the changeset already includes the error of the value of `:birth_date` being invalid, so we just return the changeset as-is.

If the type of the value of the `:birth_date` was deemed to be valid by the `cast/4` call (that performs type validation on the field), we can proceed to comparing the value against different cut-off dates—or, differently put, we can compare the difference (an integer) to today's date to integers representing upper and lower bounds of the number of days to today's date. We'll use `Ecto.Changeset`'s `get_field/3` function to get the value, and then utilize functions of the built-in `Date` module for the comparison:

- `utc_today/1` (without an optional non-default calendar) to get today's date.
- `diff/2` to calculate the difference between dates in days.

```
iex> today = Date.utc_today()
~D[2025-02-22]
iex> changeset |> get_field(:birth_date) |> Date.diff(today)
-14513
```

We can then check whether the integer resulting of this function pipeline is within the past 100 years (greater than -100 × 365 days), and not within the past 15 years (smaller than -15 × 365 days). The first comparison (max. 100 years of age) is a common-sense one, and the second one (min. 15 years of age) applies if Northwind Elixir Traders is a legal entity operating in the EU (but, really, treat this as an example, not as legal advice—I neither am, will be, or ever wanted to become a lawyer).

For example (not an example of amazingly readable code, but a succinct pipeline, nevertheless):

```
iex> changeset |> get_field(:birth_date)
~D[1985-05-30]
iex> (changeset |> get_field(:birth_date) |> Date.diff(today)
 |> then(&Kernel.and(Kernel.>(&1, -100 * 365), Kernel.<(&1, -15 * 365))))
true
```

By the way: did you already know that `and`, and operators such as `>` and `<` are actually functions of Elixir's `Kernel` module?

When such a check returns true, we leave the changeset untouched. Otherwise, we need to add a validation error to the changeset's `:errors` list with the `add_error/4` function of `Ecto.Changeset`. We can use the simplified version with arity 3 and simply provide a non-parameterized error message:

```
iex> changeset.valid?
true
iex> add_error(changeset, :birth_date, "date not within specified limits")
#Ecto.Changeset<
  action: nil, valid?: false
  changes: %{birth_date: ~D[1985-05-30], first_name: "Kumba", last_name: "Ya"},
  errors: [birth_date: {"date not within specified limits", []}], …
>
```

Note that by using the `add_error` function call, the resulting changeset *automatically becomes invalid* (`valid?: false`).

Alternatively, we might want to provide better user guidance by parameterizing the message through string interpolation within the message string, and providing a keyword list of the variables in the string, and their values. Everything worth doing is worth doing well, so let's also add a `:validation` option with a meaningful value to indicate the validation step that caused the error:

```
iex> add_error(changeset, :birth_date, "date '%{date}' not within specified limits",
date: get_field(changeset, :birth_date), validation: :age_range)
#Ecto.Changeset<
  action: nil, valid?: false,
  changes: %{birth_date: ~D[1985-05-30], first_name: "Kumba", last_name: "Ya"},
  errors: [birth_date: {"date '%{date}' not within specified limits",
     [date: ~D[1985-05-30], validation: :age_range]}], …
>
```

As we saw in the previous chapter, the `traverse_error/2` function could then be used to generate human-readable error messages. In that case, it would be more helpful to provide information about the alluded-to "specified limits" in the error message, instead of providing the value of the field that the user most likely is already aware of.

Putting all this together we can define this validation function as a private function within the `Employee` module:

Figure 8.3. **A function that validates that the calculated age based on a date field falls within a specified range**

```elixir
defmodule NorthwindElixirTraders.Employee do
  …
  defp validate_age_range(changeset, field, [min: age_mn, max: age_mx] = opts)
       when is_atom(field) and is_list(opts) do
    case {changeset.errors[field], get_field(changeset, field)} do
      {nil, field_value} when not is_nil(field_value) ->
        within_range? = field_value |> Date.diff(Date.utc_today())
          |> then(&Kernel.and(Kernel.>(&1, -age_mx * 365), Kernel.<(&1, -age_mn * 365)))

        if not within_range? do
          add_error(changeset, field,
            "date not within the specified span between '%{min}' and '%{max}' years ago",
            min: age_mn, max: age_mx, validation: :age_range)
        else
          changeset
        end

      _ ->
        changeset
    end
  end
end
```

And this is how we would add use this validation function as an extra step in our changeset function's pipeline:

```elixir
defmodule NorthwindElixirTraders.Employee do
  …
  def changeset(data, params \\ %{}) do
    …
    data
    |> cast(params, permitted)
    …
    |> validate_age_range(:birth_date, min: 15, max: 100)
  end
  …
end
```

Let's see it in action with a few examples. As always, the superfluous lines have been omitted with an ellipsis for brevity. When the type of the provided `:birth_date` field value does not conform to the field's type in the schema (`:date`), the `:errors` in the changeset only contain the entry resulting from the `cast/4` function call. This makes sense, due to the `case` statement within the `validate_age_range/3` function we implemented:

```
iex> params = %{first_name: "Kumba", last_name: "Ya", birth_date: "yolo"}
iex> Employee.changeset(%Employee{}, params)
#Ecto.Changeset<…,
  changes: %{first_name: "Kumba", last_name: "Ya"}, valid?: false,
  errors: [birth_date: {"is invalid", [type: :date, validation: :cast]}]
>
```

When the provided `:birth_date` field value is a valid date but fails to pass the check of the validation function (too old, or too young), we can report back a helpful, string-interpolate-able error message and the values to plug into it, as well as the name of the failed validation check (`:age_range`):

```
iex> Employee.changeset(%Employee{}, %{params | birth_date: Date.new!(2015, 5, 30)})
#Ecto.Changeset<…,
  changes: %{birth_date: ~D[2015-05-30], first_name: "Kumba", last_name: "Ya"},
  errors: [birth_date: {"date not within the specified span between '%{min}' and '%{max}' years ago", [min\
: 15, max: 100, validation: :age_range]}], valid?: false
>
```

On the "happy path" of the provided `:birth_date` field value being both valid and within the admitted range, no errors are added; since the conditions of every other validation function in the changeset function were satisfied, the changeset generated by the `cast/4` call is returned untouched:

```
iex> Employee.changeset(%Employee{}, %{params | birth_date: Date.new!(1985, 5, 30)})
#Ecto.Changeset<…,
  changes: %{birth_date: ~D[1985-05-30], first_name: "Kumba", last_name: "Ya"},
  errors: [], valid?: true
>
```

More than one changeset function

We mentioned earlier that it's possible to have more than one changeset function in a module. The reason for this could be that you want to cast and validate different fields of the schema for different purposes. If, for example, we were dealing with a `%User{}` struct representing a user of a Phoenix LiveView app, we might want to have changesets for these purposes:

- **Registering a new user with an email address and a password**, with values coming from the input form on a registration page of the app. In this case, we'd need to validate that:
  1. the string provided in the email address input field indeed resembles an email address, and
  2. the string provided in the password input field satisfies conditions such as a minimum and maximum length, and the presence of lowercase, uppercase, and special characters.
- **Changing the password of an existing user**, either from a "user settings" page of the web app (when the user is already authenticated) or from a "Forgot password" page (when the user has not yet logged in). In this case we'd need to apply the same rules as before for validating that the string provided by the user represents a strong new password.

Since in both of those cases we'd need to use the same rules for evaluating that the provided password is a strong one, we could abstract these rules into a private function within the module that would be used by both of these changesets.

In fact, this is what the `phx.gen.auth` Mix task of Phoenix does. It generates two changesets in the module that represents a user: a `registration_changeset/3` and a `password_changeset/3`. Both of these changesets use the private function `validate_password/2` that's generated within the module representing the user struct. Phoenix is outside of the scope of this book, but the linked code segments are great examples of the above lessons in action.

In our specific example of the `Employee` module, we could also define different changesets for:

- **Registering a new employee or modifying at least the core data (first and last name, and the birth date) of an existing record**; for this we'd use the currently defined changeset to possibly set values for all fields of the `Employee` schema, but leave the `:photo` and `:notes` field optional.

- **Modifying the notes on an existing employee record**; for this we'd only cast `:notes`, make the provision of a value mandatory with `validate_required([:notes])` and apply the same limitation on the maximum length with `validate_length(:notes, @notes_mxlen)`. We could also take the provided value, sanitize it (e.g., by removing any HTML markup) and then use `put_change/3` to set the value of `:notes` in the changeset's `:changes` map to the sanitized value. Note that you could also adapt a changeset function to derive a value for a field that's `cast/4` but not provided in `params`; e.g. you could populate `:notes` with information such as the date of the employee on the day the `Employee` records was created.
- **Modifying the photo of an existing employee record**; for this we'd only cast `:photo`, use `validate_required([:photo])`, and use `validate_format/3` with a regex that would check that the filename of the provided string representing the filename of the provided/uploaded photo has an extension such as `.jp{e}g`, `.png`, or `.webp`, and/or that the uploaded file has a maximum file size. It would also make sense to amend this changeset to truncate the filename (not the extension) to a maximum length and optionally prepend a random alphanumeric string, in order to overcome filesystems' limitations on the maximum file path length. Another option, in case you store blobs on S3-compatible object storage, would be to store both the prepended/truncated filename in `:photo`, and add another field that you populate with the item's key (string) in the S3 bucket after the upload has successfully completed. Not here though—as you'll later see, Northwind Traders' data timestamps long pre-date "the cloud".

In fact, across the first two of those changesets we could extract the validation and sanitization of the `:notes` field in a private function, so that we don't repeat code in two places.

We could also use `validate_format/3` on the `:first_name` and `:last_name` fields to only allow letters (including non-Latin ones) and thus disallow characters such as "<", ">", "/" and ";" that are typically used in HTML markup and JavaScript. In fact, since this function would be useful across different schemas' changesets on plain-text fields (names, locations, etc.), we could (actually, should) extract this function in a `NorthwindElixirTraders.Helpers` module used across multiple modules representing structs (rows) of various tables in our database.

Creating another migration

For now, let's tackle another missing piece of our work on the Northwind Elixir Traders database: the `employees` table. Again, we'll first generate a new migration file:

```
$ mix ecto.gen.migration create_employees
* creating priv/repo/migrations/20250222224043_create_employees.exs
```

And, without much ado, we'll edit it to define the `employees` table according to the existing `Employee` struct and its schema, setting the `null: false` option to the three columns for which a value must be provided:

Figure 8.4. The migration that creates the employees table

```elixir
defmodule NorthwindElixirTraders.Repo.Migrations.CreateEmployees do
  use Ecto.Migration

  def change do
    create table(:employees) do
      add :last_name, :string, null: false
      add :first_name, :string, null: false
      add :birth_date, :date, null: false
      add :photo, :string
      add :notes, :string

      timestamps(type: :utc_datetime)
    end
  end
end
```

We run the migration:

```
$ mix ecto.migrate
00:41:13.702 [info] == Running 20250222224043 NorthwindElixirTraders.Repo.Migrations.CreateEmployees.chang\
e/0 forward
00:41:13.705 [info] create table employees
00:41:13.706 [info] == Migrated 20250222224043 in 0.0s
```

Since it ran successfully, we can check the schema of the new `employees` table in the SQLite database:

```
$ sqlite3 northwind_elixir_traders_repo.db ".schema employees"
CREATE TABLE IF NOT EXISTS "employees" ("id" INTEGER PRIMARY KEY AUTOINCREMENT, "last_name" TEXT NOT NULL,\
 "first_name" TEXT NOT NULL, "birth_date" TEXT NOT NULL, "photo" TEXT, "notes" TEXT, "inserted_at" TEXT NO
T NULL, "updated_at" TEXT NOT NULL);
```

It looks familiar, as it's similar to the schema of the `categories` table. Unsurprisingly, based on our earlier discussion regarding SQLite column types, the `birth_date` column's type is TEXT, the same as that of the `inserted_at` and `updated_at` columns generated by the `Ecto.Migration.timestamps/1` function in the migration file.

Conditionally persisting data in the repo

In an earlier chapter we saw that we can use the `insert/2` function of `Ecto.Repo` to create new records in a database table. With the `Employee` struct newly backed by the `employees` table thanks to the `create_employees` migration file, we can now do this:

```
iex> gully_foyle = %Employee{first_name: "Gulliver", last_name: "Foyle", birth_date: ~D[2094-09-14]}
%NorthwindElixirTraders.Employee{
  __meta__: #Ecto.Schema.Metadata<:built, "employees">,
  id: nil, last_name: "Foyle", first_name: "Gulliver", birth_date: ~D[2094-09-14],
  photo: nil, notes: nil, inserted_at: nil, updated_at: nil}
iex> Repo.insert(gully_foyle)
00:42:05.240 [debug] QUERY OK source="employees" db=1.2ms decode=0.8ms idle=1101.3ms
INSERT INTO "employees" ("last_name","first_name","birth_date","inserted_at","updated_at") VALUES (?,?,?,?\
,?) RETURNING "id" ["Foyle", "Gulliver", ~D[2094-09-14], ~U[2025-02-22 22:42:05Z], ~U[2025-02-22 22:42:05Z
]]
{:ok,
 %NorthwindElixirTraders.Employee{
   __meta__: #Ecto.Schema.Metadata<:loaded, "employees">,
   id: 1, last_name: "Foyle", first_name: "Gulliver", birth_date: ~D[2094-09-14],
   photo: nil, notes: nil, inserted_at: ~U[2025-02-22 22:42:05Z], updated_at: ~U[2025-02-22 22:42:05Z]
 }}
```

Nothing new here, except that:

- The `:inserted_at` and `:updated_at` fields were automatically set to the current UTC datetime. Good.
- Without using a changeset, we skipped validations. Thus, we were able to successfully insert an `Employee` record with a `:birth_date` value that's in the future; something prohibited by our custom `validate_age_range/2` private function. Not great.

In the earlier chapter where we tried out `Ecto.Repo.insert/2` with a `%Category{}` struct before defining the `Category` schema to include these timestamps, the second item of the return tuple was the same as the `%Category{}` struct we passed as the function argument. Here, however, we have added the timestamps to the schema. So now we know and understand better: the second item of the return tuple of a successful `insert/2` operation with a struct as the argument contains the `%Employee{}` record *as it is persisted in the database*, i.e. with the aforementioned timestamp fields populated.

However, now that we have defined a changeset function within the `Employee` module, we can also pass a changeset to the `insert/2` function, and thus perform validations before attempting to insert the record. For example, here's what happens when we try to `insert/2` a changeset with `valid?: false` (because of a deliberately missing `:birth_date` field value, which is required):

```
iex> params = %{first_name: "Ben", last_name: "Reich"}
iex> %Employee{} |> Employee.changeset(params) |> Repo.insert
{:error,
 #Ecto.Changeset<…,
   action: :insert, valid?: false, changes: %{first_name: "Ben", last_name: "Reich"},
   errors: [birth_date: {"can't be blank", [validation: :required]}]
 >}
```

In this case, we observe the following:

- The return tuple is `{:error, …}`, and the second item of the tuple is the changeset we attempted to insert into the table.
- The value of the `:action` attribute of the returned changeset of the tuple is not `nil` anymore, but `:insert`, to indicate the repo action that was attempted.

And here's what happens when we attempt to insert a valid changeset: the second element of the `{:ok, …}` return tuple is the `Employee` record *as it was successfully persisted in the database*:

```
iex> %Employee{} |> Employee.changeset(Map.put(params, :birth_date, Date.new!(1976, 2, 15))) |>
Repo.insert
{:ok,
 %NorthwindElixirTraders.Employee{
   __meta__: #Ecto.Schema.Metadata<:loaded, "employees">,
   id: 2, last_name: "Reich", first_name: "Ben", birth_date: ~D[1976-02-15], photo: nil,
   notes: nil, inserted_at: ~U[2025-02-22 22:44:46Z], updated_at: ~U[2025-02-22 22:44:46Z]
 }}
```

We will examine repo actions in **Chapter 9**, while we learn about casting and putting associations.

Summary and outlook

In this chapter we learned about using changeset functions within modules with Ecto schemas, using built-in and custom validation functions, and basic persistence of data in the database using changesets. However, for all our discussions about validations of the values of a table record we want to persist, we left something out: making sure that we are not allowed to persist the same data multiple times. Hence, the next chapter will deal with uniqueness constraints to avoid multiple copies of the same records, before we proceed to connecting different tables of the Northwind Traders database and thus gradually recreate more complex and more educational database operations.

Chapter 6: Uniqueness constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Unique indexes and unique constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

New insights about Ecto and SQLite

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Ecto migration rollbacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 7: Basic table associations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Fields with numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

A one-to-many association using a foreign key

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Dealing with SQLite's limitations on foreign key constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

A custom validation function for a foreign key constraint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 8: Associations with Ecto.Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Interlude: exploration vs. exploitation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The N+1 query problem vs. Ecto's `preload/3`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

One-to-many relationships and `has_many`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Creating the flip-side of a one-to-many relationship

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Improving upon an existing association

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 9: Casting and putting associations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Casting an associated field (unsuccessfully)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Putting an association

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Creating a new record using `cast_assoc/3`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Casting the reverse association

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 10: Importing data from a dynamic repository

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Seeding the database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Fetching the original Northwind Traders data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Option 1: convert the original file to our conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Option 2: do as the web page instructs, and then use a second Ecto repo

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using a dynamic Ecto repo

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Implementing a data importer module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Importing in bulk (is cheaper)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Automatically prioritizing tables for importing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Validating that our data importing is robust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 11: Modeling further tables, and data cleansing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

SIPOC of the task of modeling a table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

To batch or not to batch? That is a valid question.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Step 1: take care of the migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Step 2: take care of the module, its schema, and its default changeset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Data cleansing: dealing with phone numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Detecting an international phone number

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Detecting an NANP-formatted phone number

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Converting to an international phone number

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: importing data from an online CSV data source

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Extracting country names and dial codes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Creating the countries table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The upside of no uniqueness constraint on `:alpha3` or `:dial`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Making sure that it all works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Finalizing our automatic generation of the `countries` table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Converting phone numbers to the international format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Short side-quest: building a query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Creating a validation function for phone numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Tearing down the database and re-importing data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Modeling the Customers and Orders tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Opportunities for improvement: all business entities in one table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Forging ahead with defining the `Customer` module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: `DATETIME` or `DATE` in the `orders` table?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Evaluating the need for a complex uniqueness constraint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Employees vs. customers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Shippers vs. customers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Shippers vs. employees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Disambiguating: why we need a `DATETIME` field

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Defining the new `Order` module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The cherry on top: country name validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Reaping the rewards: enabling the `has_many` calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Quality-assuring the entire importing process

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 12: Modeling a join table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using `belongs_to` and `has_many`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: the application life-cycle and compiled vs. loaded modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Importing the OrderDetails table data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using `many_to_many`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 13: Cleaning up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Discovering some loose ends of our table prioritization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Building a proper table dependency graph

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Implementing a Depth-First Search algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 14: Insights from data with queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Taking inspiration from a real situation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Elixir functions vs. database functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Simple joins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The woes of money amounts and floating-point numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: timing the execution of functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: rudimentary parallelization using the Task module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Floating-point values for prices? No, thanks.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Converting dollar prices in our database to cents (integers)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Multiple changeset functions are A-OK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Completing the switch from dollars to cents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Back to querying data and answering questions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Who are the biggest customers?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using a sub-query to create a composable query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: customers' share of revenue, and business resilience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-side-quest: using a `LEFT JOIN` to get the customers without orders

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Calculating the base data of the share of revenues vs. share of customers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Calculating the Gini coefficient

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Calculating the revenue share of any entity, Part I

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Virtual fields and fragments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Calculating the revenue share of any entity, Part II

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The vital few vs. the trivial many

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

It’s not only about revenues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Making our code more flexible

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Short side-quest: dynamic queries with dynamic fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Back to refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Time filtering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Making sure that date(time) comparisons are correct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using `where/3` for a datetime comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Implementing a date(time) filter in our queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using keyword options as a function argument

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: switching from positional to named bindings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Back to the date(time) filter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Dynamic fragments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Improving the date filter for monthly and annual breakdowns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Calculating metrics over time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Revenues and quantities by the customer's country

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 15: More and more-advanced queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Window functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Average product price per category

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Similarity search with `like/2` and `like/2`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Queryable to SQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Building an SQL fragment that rounds results

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Partitioning customers by country

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Dynamic queries with window functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Dynamic joins and the pains of positional bindings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using named bindings to traverse the ERD

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Refactoring query logic for consistency and composablity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: reducing function arity by pattern-matching on the query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Back to the dynamic window query—for real this time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The impact of `distinct/3`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Modifying the dynamic window query function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Defining windows, partitioning, and grouping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Expanding-window queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using the date-filtering feature once again

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Historical running total of revenue or quantity per customer, or anything else

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Disaggregating query results by a field

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Investigating the results of the window query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Generalizing our window query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Back to the running-total-revenues query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Sliding-window queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Aggregating `OrderDetail` revenues into `Order` revenues as a sub-query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using a sub-query as the data source of the window query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Generalizing the subquery of the sliding-window query

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Making the sliding-window query more flexible

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: plotting `:agg` vs. `:date` in IEx

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Single partitioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Partitioning at the right place, and to the right degree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Double (actually, triple) partitioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Side-quest: recursive disaggregation by more than one field

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Tying loose ends, and fulfilling a promise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 16: Very complicated window queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Adding zero rows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Recursive Common Table Expressions (CTEs)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Recursive CTE for a sequence of dates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Implementing the recursive CTE in Ecto

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Getting `Order` rows without actual orders by using a `LEFT JOIN`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Using the recursive CTE with other queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Nerdy data-sleuth side-quest (and Elixir gym): hidden patterns in data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Coalescing data to turn `NULL` into 0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

The Holy Grail: Rolling and running `n`-day aggregation with partitioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Performance woes, and indexing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

How to (not) make the queries’ foundation less blatantly expensive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Chapter 17: Towards “Northwind Elixir Traders 2.0”

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Ideas and exercises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Create the building blocks of a record-specific dashboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Add a product description—in different languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Add a `:status` field to some of the schemas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Normalize the database further

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Add a table for payment terms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Link products to each other

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Enable deviation from a product's price on an order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Enrich certain schemas with entity-specific fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Validate addresses with an external API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Handle documents associated with any type of database entity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Enrich every record with a `:notes` field, or with a series of notes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Implement an organizational chart

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Enable bundles of products

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Build out the buy side of the ERD

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

Summary and outlook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/northwind-elixir-traders.

About the Author

Isaak Tsalicoglou is a relatively recent aficionado of the Elixir programming language, its ecosystem, and its developer community. He discovered Elixir in 2021 and took the plunge, going *all-in* on Elixir in late 2022 after a serendipitous discussion with a great friend about the joys of functional programming. Since then, he's been applying his growing Elixir skill-set in the development of Apache-2.0-licensed Elixir packages, open-sourced web apps such as Changelogrex for the exploration of Linux kernel Changelogs, and of proprietary web apps such as Pragmata, for the management of spare parts for industrial equipment, and Bellweather, for the caching, analysis, and serving of meteorological data.

Isaak's budding fascination with Elixir, Ecto, Phoenix and Phoenix LiveView followed hot on the heels of his two prior years of experience developing, deploying and improving web apps and REST APIs with Django, Django Ninja, FastAPI, and backends with NocoDB for small-business use cases, such as the management of product information, inventory data, and commercial registry data, the generation of pricing recommendations and commercial offers, the historical analysis of invoices, and the tracking of order progress using QR codes.

Before that, parallel to and sometimes supporting his managerial roles and entrepreneurial activities, Isaak developed proprietary MVPs and tools in Python, such as NLP on web-scraped employee reviews, clustering and regression of web-scraped used-vehicle data, Machine Learning applications on hardware testing datasets of engineering design processes, post-processing and display of radar data, object detection and tagging of photos, and blob detection of aggregates in concrete slab cross-sections.

Even earlier, as a development engineer and R&D project manager, Isaak programmed extensively in Python, building numerous software tools for engineering design, FEM/CFD/numerical simulation, post-processing and analysis of turbines, shafts and other components. He also adapted, modularized and wrapped in-house C++ and FORTRAN tools for embarrassingly parallel computation on HPC infrastructure. In this role, he became skilled at technical writing, compiling numerous manuals on newly developed engineering design processes and technical reports on mechanical component design and verification.

Besides *Northwind Elixir Traders*, his first technical book so far, Isaak has also written extensively (articles and books) about all aspects of organizations and corporate professionals turning their knowledge about markets, customers, and technology into a competitive advantage through the judicious use of technology, vigorous collaboration across functions and locations, and an entrepreneurial mindset… but also writes about how hype, dogmatism, agency issues, culture clash and misaligned incentives prevent organizations from doing so.

Isaak studied Mechanical Engineering at ETH Zürich and also holds an MBA degree with Honors from IMD in Lausanne, Switzerland.

Through OVERBRING Labs, his consultancy and bespoke software development and product incubation and fractional development/management company in Athens, Greece, Isaak helps ambitious organizations, both domestically and abroad, to improve knowledge-work operations and to grow their business through both in-house and customer-facing digital means, by providing his services as a fractional manager of business, technology, and development projects, and as an expert in problem-solving, risk-management and QA related to software products and product-focused ventures. As part of this activity, Isaak is also the co-host of The Puzzle Podcast, a series of 10-minute episodes on the wicked problems and pieces of the "puzzle" of product innovation, development, and marketing.

Beyond that, Isaak is the General Manager of Breek.gr, the digital command-center for real-estate property management operations in Greece, a SaaS that he is in charge of incubating as Head of Software Development Projects of the Tethys family office.