

The Phoenix Tutorial

Jeffrey Biles

Introduction

This book will focus on teaching you the fundamentals of web development.

Web development is a powerful tool that can help you in many endeavors—whether it's building a startup that can change the world, getting a job to provide for your family, making a web app as a creative expression, or using web development as a secret weapon to increase your effectiveness in your current career, I think that you'll be glad you invested the time into learning this valuable skill.

We're going to learn web development using what I believe are some of the best tools available: Elixir (a blazing-fast programming language with friendly syntax), Phoenix (a modern web framework that combines functional programming with the best of Ruby on Rails), and Ecto (a maintainable system for interacting with the database). (More on that later in this introduction)

What are we building?

We'll be taking a **hands-on approach** to learning, which means you'll be practicing each concept quickly after you learn it, applying it in a real application when possible. In addition to the code given in each chapter (which you should type into your computer and run), there will also be exercises, which I highly recommend you do—if you want to remember what you're learning.

Throughout this book we're going to be building the Star Tracker app, which boldly goes where no resource exchange app has gone before.

Note: this is an early version of the app. There will be some stylistic and usability improvements made before we get to that part of the tutorial.

We'll be creating a way to buy and sell resources (mostly the elements that make up stars, hence the name) on a centralized market. You click a button, you buy and sell a resource. We'll use LiveView to automatically show the user when

Why this book, and not Programming Phoenix or another book?

The screenshot shows a web application interface for the Phoenix Framework. At the top, there's a navigation bar with a red bird logo, the text "Phoenix Framework", and user information "Spock", "550 GPL", and a "Log out" link. Below the header, the main content area has a title "Hydrogen Show". Underneath the title, there's a list of items:

- Price: **2**
- Amount available: **271**
- Personally owned: **200**

On the right side, there are several buttons for buying and selling Hydrogen:

BUY 1	SELL 1	Cost: 2
BUY 10	SELL 10	Cost: 20
BUY 100	SELL 100	Cost: 200
Custom Amount:		
BUY 12	SELL 12	Cost: 24

Below the item list, there are several log entries:

- Data **sold 10 Hydrogen** at 16:22pm UTC on Apr 30, 2019
- Data **sold 10 Hydrogen** at 16:22pm UTC on Apr 30, 2019
- Data **bought 10 Hydrogen** at 16:22pm UTC on Apr 30, 2019
- Data **bought 10 Hydrogen** at 13:31pm UTC on Apr 30, 2019

At the bottom left, there's a "Edit Back" link.

Figure 1: Insert picture of finished app here

someone else makes a trade - without a page refresh, and without us writing a single line of Javascript.

It's not super fancy, but it gives us many opportunities to solidify the core concepts you'll need to start creating your own Phoenix apps (and understand the more advanced Elixir and Phoenix books, should you choose to read them).

Why this book, and not Programming Phoenix or another book?

There are lots of great resources for intermediate and advanced developers to learn Elixir and Phoenix, but relatively little for beginners. In addition, the resources that do exist tend to rush past the core web framework features in order to get to the cool advanced features that set the framework apart, like Channels and OTP.

This book aims to be your first Elixir and Phoenix book (and maybe even your first web development book), but not your last. We'll go into the basics of the language and the framework, giving you a solid foundation so you can crank out simple web apps and be prepared to read and understand the books written on more advanced topics.

That said, this is not meant to be your first introduction to programming.

Prerequisites

While I don't expect you to have familiarity with Elixir or web programming, I do expect you to know how to put together a simple program in *some* language. If you don't meet that standard, here are some good (free) resources:

- Intro to Computer Science (A video course that teaches Python basics)
- How to Design Programs (this Lisp book is how I learned)
- Invent With Python (Book series that involves making lots of games)

Why Phoenix and Elixir?

This section uses advanced terminology in order to compare Phoenix and Elixir to other tools. If you don't understand something here that's okay—just read to get the general gist of it. There will be a summary at the end.

The current king of the backend Web Framework world is Ruby on Rails—it's what is taught at hundreds of Coder Bootcamps and used at thousands of startups (as well as some large companies). However, many of the leading lights of the Ruby and Rails worlds have moved on, and a significant number of them have moved to Elixir and Phoenix. Why is that?



RailsConf is in Phoenix. Presumably because that's where the Rails Community was already going.

RETWEETS 7 LIKES 31

Figure 2: Twitter joke about Rails and Phoenix

The answer is that Elixir combines the power, speed, and reliability of the Erlang VM with the beauty and expressiveness of Ruby syntax. In one stroke it fixes the biggest complaint about each language—speed for Ruby and syntax for Erlang.

The Elixir ecosystem has also learned a lot from both languages. From Ruby it takes many of the most popular tools and combines them into one tool with a consistent interface: Mix. From Erlang it takes... the entire Erlang ecosystem; you can use any Erlang package you want in Elixir (just be prepared to look at some not-as-beautiful code).

Phoenix itself has benefited greatly from seeing what Rails has done. It's taken what worked well (convention over configuration, REST, the Rails MVC style, friendly build tooling, etc.) while doing some much-needed modernization. Some of the modernizations are made available simply by escaping the inertia of the Rails codebase and creator DHH's opinions. Others are unlocked due to the increased power and features of Elixir as compared to Ruby.

So here are the biggest advantages of Phoenix and Elixir, in fancy listicle form:

- SPEED

Elixir is an order of magnitude (~10X) faster than Ruby right out of the box. This not only gets you faster response times and lets you handle more requests per box, but it eliminates a whole class of performance hacks and opens the door for doing things that would not be wise in Ruby.

- Reliability

Elixir is built on the Erlang VM, which has created systems with “nine nines” of reliability- that is, it was working 99.999999% of the time. This reliability isn’t necessarily because individual parts crash less often, but because it can recover seamlessly from crashes- a “self healing” network of processes. We won’t get into this mechanism in detail in this book.

- Real-time features

Channels/Sockets that are just as easy as in Node (the current Sockets king), but in Elixir they’re more performant. This means that “real-time” communication between server and client is a breeze compared to other languages and frameworks.

- Functional Programming Style

This can occasionally cause frustration for those coming from an Object-Oriented language (such as Ruby), but once you become acclimated it means fewer bugs and more readable code. You also get really cool features like Pattern Matching and piping that are not available in Object-Oriented languages.

- LiveView

Tired of sprinkling Javascript all over your app? LiveView lets you do form validations, auto-completes, complex interactions, and even real-time gaming, all from the server. But it doesn’t just eliminate the burden of managing two separate frameworks... it’s going to do for real-time apps what Rails did for request/response CRUD apps.

Technobabble: When to use Phoenix vs Other web frameworks?

Phoenix is still a newer technology, but it's well past 1.0 and stable enough to run production apps (and the underlying technologies—Elixir and Erlang—are rock solid).

If you're starting a new project, choose Phoenix! But if you have a legacy project then there are more tradeoffs to consider.

I'd say that if you are having performance problems which can't be solved with basic solutions (such as fixing n+1 database queries) then it may be worth it to switch to Phoenix. High availability requirements are another good reason. It also may be worth it if your project involves lots of concurrency or sockets—stuff that most languages and frameworks do poorly, but Elixir and Phoenix do really really well.

Another huge reason to switch to Phoenix is if you want real-time interactive apps. Phoenix LiveView lets you have that without having to manage two frameworks + an API layer, like you would if you used, for example, Rails and React. It's also fantastic for dashboards and other apps that need up-to-date information from the server.

If you're investigating Phoenix, then it's likely that you fall into one of those camps. However, if you're happy with your current tech, or if you're unhappy for different reasons, consider whether the benefits of Phoenix will be worth the cost of a rewrite.

Why Phoenix? (for beginners)

If you're a beginner that didn't quite understand all of the last section, here's the tl;dr on why Elixir/Phoenix/Ecto can be a better choice than Ruby/Ruby on Rails/ActiveRecord or other popular web programming tools:

- Phoenix is an order of magnitude faster (due to Elixir's use of the Erlang VM)
- Your Phoenix code is significantly less likely to descend into spaghetti code (due both to Elixir's functional programming and Phoenix and Ecto's learning many lessons from how web apps can go wrong)
- LiveView means you don't have to learn React
- You'll be prepared to take advantage of Elixir's advanced features *when their use becomes necessary*

Conclusion

I hope you're as excited as I am to begin! Turn the page and we'll start installing Elixir.

Changelog

0.3.0

- **New Section Added: Generating a Resource**
- **Fixed syntax highlighting**
- Readability improvements, including increasing font size and adding a border around images
- Fixed error with “em” tag
- Updated text of “use, import, alias” to reflect that it’s in an appendix

0.2.0

- **New Section Added: Customizing Your First Phoenix Page**
- Added preview of app in introduction
- Fixed typos/grammar in previous sections

0.1.4

- Updated Introduction to include LiveView
- Updated to Elixir 1.8.1
- Fixed typos/grammar
- A bit of time has passed since the last update, but things should be going super fast from here on out.

0.1.3

- **Website Created**

0.1.2

- Added a Captain’s Log for snake_case vs PascalCase
- Added technobabble for assignment vs pattern matching
- Added a note on tab auto-complete in iex
- Renamed `recombinePhraseRequired` function to `recombine`
- Updated Arguments exercise to be easier to understand
- Updated Maps exercise to not accidentally resemble a Ruby on Rails convention
- Added paragraph on destructuring Maps
- Removed errant camelCases and incorrect references to hashes
- Updated String section for clarity
- Fixed some typos/grammar stuff

0.1.1

- Rewrote the introduction
- Rewrote immutability section, starting with simpler examples and showing an example of mutable data
- Rewrote the case statement section, starting with a simpler example
- Demarcated all code samples in exercises as code (so they are easier to copy/paste)
- Indent iex output lines so they're easier to read
- Fixed many typos
- Turned all usages of "method" to "function"
- Made pattern matching exercises a bit more clear
- Added a captain's log about the command line conventions
- Added a captain's log about the interpreter
- Added a captain's log about loading elixir files
- Added a captain's log about function ordering
- Added flavor bits to captain's logs, to hint at a story
- Added acknowledgements
- Added changelog
- Added copyright and license
- Pdf version: links in footnotes
- Pdf version: Give codeblocks a background color
- Pdf version: Decreased margins so codeblocks fit
- Many small improvements/rewordings

0.1

- **Introduction and first section created.**

ACKNOWLEDGEMENTS

Acknowledgements

A huge thanks to those who have offered feedback on the manuscript: Derek Wood, Franco Barbeite, Giuseppe Caruso, Richard Poole, Matthew Davis, Peter Karth, and Joep Stender. This book would be much worse without them.

Copyright and License

The Phoenix Tutorial. Copyright 2017-2019 by Jeffrey Biles.

All source code in *The Phoenix Tutorial* is available under the MIT License. The book itself and the non-code writing is, unless specifically stated otherwise, copyright Jeffrey Biles.

The MIT License

Copyright (c) 2017-2019 Jeffrey Biles

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Installing Elixir

Hello World

Welcome to The Phoenix Tutorial. I'm so glad you decided to join us!

Captain's Log: The Introduction

Stardate: far future.

I have just discovered the first fragments of this fascinating manuscript detailing the Phoenix technology that underlies our fleet.

At first I wasn't sure what I was reading, since it seemed to jump straight into technical instructions, but then I went back and read the Introduction and it all made sense.

I'll be recording my observations, both for myself, and so that others may continue my research should anything... happen.

As we said in the intro, we'll be taking a **hands-on approach** to learning. And that means that before we can do anything, you're going to have to get Elixir installed on your computer.

Installation

Installing Elixir is the most system-dependent installation step of this entire book. Each function shared will install both Elixir and Erlang, the language which Elixir builds on top of.

Check which operating system you are using and follow the associated instructions.

Captain's Log: Command Line Notation

When we find a dollar sign at the beginning of a line of code, it appears to signify that everything following should be typed out on the reader's "command line".

The command line was the 21st-century equivalent of our ship computer; you tell it a command (by typing with your fingers—such barbarians!) and it does exactly what you ask. The most common name for this command line was "bash" (as I said—violent barbarians!), but there were several other variants in use.

I will continue reporting my findings as I uncover more of this fascinating 21st-century document.

Mac

1. Install or update homebrew

```
$ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

or

```
$ brew update
```

2. Install Elixir

```
$ brew install elixir
```

Windows

Download the Installer and run it: <https://repo.hex.pm/elixir-websetup.exe>

If you're using Windows, make sure you have the Bash shell installed (<https://msdn.microsoft.com/en-us/commandline/wsl/about>). This new feature, available on Windows 10 and above, will allow you to use the same shell commands as Linux and Mac users.

Other

You can find instructions for other platforms (mostly Linux flavors) at <http://elixir-lang.org/install.html>.

If your Linux is Debian-based, be sure to explicitly install Erlang so you can get all the necessary packages (instructions here: <http://www.phoenixframework.org/docs/installation#section-erlang>)

Installing Hex

From here on out things get (generally) platform-agnostic. There are a couple of extra commands for those on Debian-based systems, which can be found on the Phoenix Installation Guide: <http://www.phoenixframework.org/docs/installation>

Hex is the Erlang package manager, which lets us specify and download our dependencies. Mix is our Elixir task runner. We'll use Mix (which came with Elixir) to install Hex.

```
$ mix local.hex
```

We'll talk more about both of these later.

Conclusion

We've got Elixir installed now and we're ready to start learning! In the next couple chapters we'll go over the basics of the language, with a preference towards what is most commonly used in Phoenix.

Getting Started with Elixir

Elixir is an awesome language that is filled to the brim with exciting features. In part 1 we're only going to cover the basics—enough to get started with Phoenix. Then in subsequent parts of this book we'll add on more language constructs as needed.

The Elixir Interpreter

In order to show off the language features, we'll want to jump down to something simpler than a full Phoenix app. To do that, we're going to use the Elixir Interpreter from the command line.

Open it up by typing `iex` in your command line.

```
$ iex
```

Now you should see something like the following:

```
$ iex
Erlang/OTP 21 [erts-10.3.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [hip...
```



```
Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

This is good! That means it's working.

This guide uses Elixir 1.8.1, but it should work as long as the first number (major version) is '1' and the second number (minor version) is 8 or greater. The third number is called the patch version, if you're curious.

Now you can type Elixir code in and it'll run right there (after you hit the Enter key).

```
iex(1)> 2 + 3
      5
```

Boom. Type in `2 + 3`, hit enter, and it shows the result `5` in the line below.

Your interpreter will show the results all the way to the left, but I've indented the results in this book so input/output lines are easier to differentiate and so output is easier to match up with the input that produced it.

When we build our Phoenix app we'll spend the majority of our time interacting with it in other ways, but we'll always have the option to drop back to the interpreter and play around.

```
iex(1)> 2 + 3
      5
iex(2)> (2 + 3) * 3
      15
iex(3)> 2 + (3 * 3)
      11
iex(3)> num_tacos = 3
      3
iex(4)> num_burritos = 3 * 5
      15
iex(5)> num_tacos + num_burritos
      18
```

The above demonstrate numbers, basic math, order of operations with parentheses (this will apply to more than just math), and the assignment and use of variables.

Captain's Log: Command Line Interpreter

Our scientists have discovered how to access the Command Line Interpreter (iex) that this manuscript speaks of.

At first we thought that we had forever trapped our computers in the interpreter, but we eventually discovered that we could exit by hitting Ctrl + C twice in a row. This is good for when you're done, or if you get the interpreter into a weird state you don't understand.

Exiting and re-entering also solved another mystery- the numbers in the interpreter prompt. A few more superstitious in our crew thought them at first some sort of sinister countdown, but they are simply the number of lines since the interpreter started.

Interpreter Exercises

What is displayed when you type each of these into the command line interpreter? After thinking about it, test your answer.

1. `2 * 10`
2. `(2 * 10) + 3`
3. `2 * (10 + 3)`
4. `num_tacos = 2 + 3; pepper_per_taco = 3; num_tacos * pepper_per_taco`

(If you can't guess what ; does, just go ahead and play around with it. Playing and testing your hypotheses is cheap in the command line!)

Strings

There are two types of Strings in Elixir.

```
iex(1)> "hello universe"  
"hello universe"  
iex(2)> 'hello universe'  
'hello universe'
```

For the purposes of this book we'll be using double-quoted strings (sometimes called Binaries). Single-quoted strings (sometimes called Char Lists) are mostly used when interfacing with old Erlang libraries.

Technobabble: Binaries vs Char Lists

Single-quoted strings are Lists of Characters and double-quoted strings are UTF-8 encoded binaries represented by a series of graphemes. Even though we think of both of them as “Strings”, we use different sets of functions on each.

For a fun demonstration of a quirk of Char Lists, try putting [111, 111, 112, 115] in the command line.

Here's some basic inline usages of strings

```
iex(1)> "hello" <> "universe"  
"hellouniverse"  
iex(2)> "hello" <> "universe"  
"hello universe"  
iex(3)> place = "universe"  
"universe"  
iex(4)> "hello #{place}"  
"hello universe"
```

Aside from these inline usages, most action on strings happens via functions being applied to them. Here's an example:

```
iex(5)> phrase = "hello vast universe"  
"hello vast universe"  
iex(6)> String.split(phrase, " ")  
["hello", "vast", "universe"]  
iex(7)> upcase_phrase = String.upcase(phrase)  
"HELLO VAST UNIVERSE"  
iex(8)> phrase <> " -> " <> upcase_phrase  
"hello vast universe -> HELLO VAST UNIVERSE"
```

The string usage above is simple but allows us to demonstrate two important facets of how Elixir works.

First, when we call `split` and `upcase`, we're not changing the original phrase. Instead, we're creating a new copy of the phrase, which can be assigned to a variable (such as `upcase_phrase`). The original string is “immutable”—that is, it cannot be changed (although a new immutable value could be assigned to the variable `phrase`). We'll go over immutability in more detail in chapter 1.5.

Second, instead of calling a function which is stored on the string itself (`phrase.split(" ")`) we take our function and apply it to the string (`String.split(phrase, " ")`). This may seem like a trivial difference, but in fact it is vital to understanding the functional nature of Elixir.

Functional vs. Object-Oriented

In an object-oriented language the phrase would be an instance of the class `String`, and would therefore have available to it all the `String` functions. Hence you could call `phrase.split(" ")`.

In a functional language, the phrase is the data, which can be acted upon by functions. These functions are organized, so the most common functions which can act upon strings are stored in the `String` module, but what's important is we're separating out the data from the function instead of throwing them all in one “object”.

If this doesn't make sense to you, that's okay—just move on. We'll cover modules in the next chapter (and make our own), and the functional style of coding will become clearer to you as you write more Elixir.

String exercises

What is displayed when you type each of these into the command line? After thinking about it, test your answer.

1. `String.split("hello amazing universe", " ")`
2. `String.split("hello amazing universe", "i")`
3. `setting = "stun"; "set phasers to #{setting}"`

Enum

The `Enum` module is meant for working with `Enumerable`s. Although there are others, the most common type of `Enumerable` is a list, so that's what we'll be working with today.

```
iex(1)> split_phrase = ["hello", "vast", "universe"]
          ["hello", "vast", "universe"]
iex(2)> Enum.count split_phrase
          3
iex(3)> Enum.join split_phrase, "_"
          "hello-vast-universe"
iex(4)> Enum.member? split_phrase, "universe"
          true
iex(4)> Enum.member? split_phrase, "univers"
          false
```

Once again we have our functions which are organized by module (`Enum`), which are then applied to the data (`split_phrase`).

We're also demonstrating a parentheses-free function syntax. It's equivalent to having parentheses, except it's a bit cleaner (but, in some cases, more ambiguous. In those cases parentheses should be added back for clarity).

Char Lists vs Binaries

As an aside, remember how we have single-quoted strings (arrays of Chars) and double-quoted strings (binaries)? Those types end up being important. We use **Enum** functions on single-quoted Strings, and **String** functions on double-quoted Strings.

```
iex(1)> Enum.count 'hello'  
5  
iex(2)> Enum.count "hello"  
** (Protocol.UndefinedError) protocol Enumerable not implemented for "hello"  
  (elixir) lib/enum.ex:1: Enumerable.impl_for!/1  
  (elixir) lib/enum.ex:146: Enumerable.count/1  
  (elixir) lib/enum.ex:467: Enum.count/1  
iex(3)> String.split 'hello universe'  
** (ArgumentError) argument error  
  (stdlib) :binary.split('hello universe',  
    [" ", "", "", "", "", "", "", "", "", "", "", "", "", "", "",  
     <<194, 133>>, " ", "\t", "\n", "\v", "\f", "\r"], [:global, :trim_all])  
  (elixir) lib/elixir/unicode/properties.ex:344: String.Break.split/1  
iex(4)> String.split "hello universe"  
["hello", "universe"]
```

Want to see something cool? Press ‘tab’ in iex to trigger an auto-complete. So you type in “Stri”, hit ‘tab’, and it’ll complete it to “String”.

Even better, if there are multiple options, it will show you all of them

that match your search. Try hitting tab after typing in “String.” (make sure to include the dot)- it’ll show you all of the functions available on that module.

Enum Exercises

What is displayed when you type each of these into the command line? After thinking about it, test your answer.

1. `Enum.member?(["hello", "vast", "universe"], "hello")`
2. `Enum.member?(["hello", "vast", "universe"], "hello vast universe")`
3. `Enum.join(["hello", "vast", "universe"], "! ")`
4. `Enum.join(["hello", "vast", "universe"], "joining with this string is a very bad idea")`

Chaining functions together

Let’s say we wanted to count how many words are in a certain phrase. We don’t have a “word count” function, but we can get there by applying multiple functions in a row.

There are several ways we could do this. Here’s the first.

```
iex(1)> phrase = "boldly going where no man has gone before"  
"boldly going where no man has gone before"  
iex(2)> split_phrase = String.split(phrase)  
["boldly", "going", "where", "no", "man", "has", "gone", "before"]  
iex(3)> count = Enum.count split_phrase  
8
```

If you’re unsure of where you’re going and what to check at every step, that’s probably the best way to go. However, for production code it can look a bit messy. Here’s another way that’s a bit more compact:

```
iex(1)> Enum.count(String.split("boldly going where no man has gone before"))  
8
```

Here we’re nesting the function calls so the result of one gets fed into the other. This is more compact, but it can still look messy.

A great solution to this is the pipe syntax.

The Pipe Syntax

In the pipe syntax, we can take the results of one function and “pipe” it as the first argument of the next function.

```
iex(1)> "boldly going where no man has gone before" |> String.split |> Enum.count  
8
```

It’s a couple more characters than the previous one, but it’s far more readable. You have the subject first (the phrase), and then you pass that phrase to `String.split`, then pass the results of that to `Enum.count`.

If you have multiple arguments, you just pass the second (and third and fourth, etc.) arguments with the function.

```
iex(1)> "boldly going where no man has gone before" |> String.split() |> Enum.join("-")  
"boldly-going-where-no-man-has-gone-before"
```

When you’re piping functions, be sure to use parentheses for your arguments—the compiler can usually infer them, but the pipe syntax makes that more difficult and so it’s best to be explicit.

Working with pipes is a core part of the Elixir experience, and so most functions you encounter will be designed with piping in mind—they’ll take *data* as their first argument, and then configuration for their other arguments. You can see this with `Enum.join`, which takes in the list as the first argument and then “`-`” as the second argument.

Pipe Exercises

What is displayed when you type each of these into the command line? After thinking about it, test your answer.

1. `"boldly going where no man has gone before" |> String.length`
2. `"boldly going where no man has gone before" |> String.split(" ") |> Enum.member?("boldly")`
3. `"boldly going where no man has gone before" |> String.upcase |> String.split(" ") |> Enum.join("... ")`

Conclusion

Already each individual line of Elixir should be looking less mysterious than before, but there’s still plenty to learn. In the next chapter we’ll go over how to create and use custom functions.

Functions

So far in our exploration of Elixir we've typed our code directly into the Elixir interpreter, but for complex functions – not to mention building an entire program – we'll want a place to store our functions.

In this chapter we'll learn how to store our code by defining modules and functions.

Storing Code: defmodule and def

Start by creating a file. I'm going to call mine `basic_elixir.ex`, but the only part of this name that you are required to copy is the `.ex` file extension, which signals that this is an Elixir file.

Then we'll create the minimum viable module in this file:

```
defmodule LearningElixir do
  def hello do
    "boldly going where no man has gone before"
  end
end
```

Our module, defined by `defmodule` (DEFine MODULE), is `LearningElixir`. Attached to that we have a `do` and its closing `end`. This pair of “delimiters” indicates that anything in between them is part of the `LearningElixir` module. Some languages use whitespace or curly braces as their delimiters, but Elixir has copied Ruby and uses the more descriptive `do` and `end` keywords.

Our function `hello` is defined by `def`. It also has a `do/end` delimiting block. Inside that block is the string “boldly going where no man has gone before”, which is what is returned when we call that function.

Let's go ahead and load that file in the Interpreter.

```
iex(1)> import_file "basic_elixir.ex"
{:module, LearningElixir,
 <<70, 79, 82, 49, 0, 0, 4, 52, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 134,
  0, 0, 0, 13, 21, 69, 108, 105, 120, 105, 114, 46, 76, 101, 97, 114, 110, 105,
  110, 103, 69, 108, 105, 120, 105, 114, 8, ...>>, {:hello, 0}}
iex(2)> LearningElixir.hello
"boldly going where no man has gone before"
```

Captain's Log: Loading Elixir Files

It took much experimentation, but we have finally discovered how to load code into the ship's interpreter.

Our error was that we were in the wrong folder- the computer didn't know how to find `basic_elixir.ex`. Our solution: be in the same folder as the file when we start the Elixir interpreter.

Our scientists have reported that we may be able to access the file from a different folder by specifying the path but, truth be told, I prefer the simple way.

We load the file into the Interpreter with the command `import_file`. We can ignore the output for now; only notice that we see both `LearningElixir` and `hello` in various forms.

Then we call our function with `LearningElixir.hello` and get back the expected string.

Note that we precede the function name with the name of the module within which it resides. Apply the logic to our commands from the previous chapter and we can see that `String.split` was calling the `split` function on the `String` module.

Next let's define a new function, which has the code that we had previously inputted directly into the command line:

```
defmodule LearningElixir do
  def phrase do
    "boldly going where no man has gone before"
  end

  def recombine do
    phrase |> String.split |> Enum.join("-")
  end
end
```

We've renamed `hello` to `phrase`, then used it in our `recombine` function. Note that since we're within the `LearningElixir` module we don't need to precede `phrase` with `LearningElixir-` `phrase` is currently directly available because it's "within scope".

Captain's Log: Scope

Scope can be a scary word, but here's a basic way to think about it. If you're in your ship's common room, and you say "I would like to sit on the couch", you don't have to specify which of the millions of couches you're sitting on. You're in the common room scope, so when you're trying to think of couches, the one in the ship's common room comes to mind first.

It's the same reason that if you talked to someone in the old United States about "the civil war", they'll immediately think of the American Civil War, not the Spanish Civil War, the American Revolution, or the Klingon Civil War. That's because they're scoped to the United States.

Scope isn't as big a deal in Elixir as it is in Object-Oriented languages, but it's still important to understand.

The rest of our `recombine` function is just like what we previously did directly in the interpreter.

Let's load and call this in the interpreter.

```
iex(3)> import_file "basic_elixir.ex"
warning: redefining module LearningElixir (current version defined in memory)
  iex:1

warning: variable "phrase" does not exist and is being expanded to "phrase()", please use parentheses to remove the ambiguity or change the variable name
  iex:7

{:module, LearningElixir,
 <<70, 79, 82, 49, 0, 0, 5, 40, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 182,
  0, 0, 0, 18, 21, 69, 108, 105, 120, 105, 114, 46, 76, 101, 97, 114, 110, 105,
  110, 103, 69, 108, 105, 120, 105, 114, 8, ...>>, {:recombine, 0}}
iex(4)> LearningElixir.recombine
"boldly-going-where-no-man-has-gone-before"
```

The result of `LearningElixir.recombine` is what we would expect if we straightforwardly combined what we previously knew.

Less expected are the two warnings.

The first warning is the result of importing the same file (and thus defining the same module) twice in one interpreter session. If we had restarted the interpreter in between loading the file then this warning would not show up.

The second warning is because `phrase` is somewhat ambiguous- it could be either a variable or a function. The interpreter automatically (and correctly) expands it to `phrase()`, which is the less ambiguous way to call a function.

There are two ways to solve the ambiguity:

```
defmodule LearningElixir do
  def phrase do
    "boldly going where no man has gone before"
  end
```

```
def recombine1 do
  phrase() |> String.split |> Enum.join("-")
end

def recombine2 do
  LearningElixir.phrase |> String.split |> Enum.join("-")
end
end
```

The first is exactly what's suggested by the warning: add in the parentheses. The second makes explicit the fact that `phrase` is defined on the `LearningElixir` module, thus removing the ambiguity. I personally prefer the second solution (it makes the function more portable because you don't have to worry about scope), but either works.

Finally, now that we're in a proper file we don't have to define everything on one line.

```
defmodule LearningElixir do
  def phrase do
    "boldly going where no man has gone before"
  end

  def recombine do
    LearningElixir.phrase
    |> String.split
    |> Enum.join("-")
  end
end
```

This version of `recombine` does exactly the same as our last version, but now instead of having everything in one line we have the pipes lined up vertically. This can be very convenient for seeing at a glance how a function is composed.

Exercises

1. Create the `LearningElixir` module, with the `phrase` and `recombine` functions in it. Import it on the command line, then run `LearningElixir.recombine`.
2. Within that module, create the `upcase_phrase` function, which returns the phrase, but all in upper case letters. Use the `phrase` function in your solution- you're cheating if you just type out the phrase manually in upper case.

```
> iex(1)> LearningElixir.upcase_phrase
      "BOLDLY GOING WHERE NO MAN HAS GONE BEFORE"
```

We introduced the relevant `String` function in the last chapter.

3. Within the same module, create the “`upcase_dasherized_phrase`” function, which returns the phrase, dasherized AND in all upper case.

Conclusion

Now you know how to create a module that organizes your functions. This increases your ability to organize your code aesthetically (lining up pipe operators) and opens up many new possibilities. However, our current understanding is still very limited.

In the next chapter, we’ll show how to create more flexible functions by giving them arguments- and by introducing our first instance of Pattern Matching, the functional programming design pattern that you’ll soon grow to love.

Arguments and Pattern Matching

Using Arguments

The `recombine` function is pretty cool, but it always uses the default phrase defined elsewhere in the module. What if we wanted to allow the user the option of putting in their own phrase?

First we'll see how to *make* them input their own phrase:

```
def recombine(phrase) do
  phrase
  |> String.split
  |> Enum.join("-")
end
```

This is the first function we've created that requires an argument, but we've used functions with arguments before. When we call it we must feed it a phrase as input (with or without parentheses)

```
iex(1)> LearningElixir.recombine("hello world")
"hello-world"
iex(2)> LearningElixir.recombine "hello world"
"hello-world"
```

Next we'll figure out how to make the phrase argument *optional*.

There are several ways to do that. The first solution we show, having a default value, will be typical of non-functional languages. The second solution, which is modeled after a common Javascript pattern, will show us how different the two language's capabilities are and introduce us to the concept of Arity. Finally, the third solution will take advantage of Elixir's functional abilities and give us our first taste of Pattern Matching.

Default Value

```
def recombine(phrase \\ LearningElixir.phrase) do
  phrase
  |> String.split
  |> Enum.join("-")
end
```

Here the argument has a *default value* of `LearningElixir.phrase`. When defining an argument, you can put `\\"` after an argument name and then give a default value- in this case, `LearningElixir.phrase`

```
iex(1)> LearningElixir.recombine
  "boldly-going-where-no-man-has-gone-before"
iex(2)> LearningElixir.recombine "hello universe"
  "hello-universe"
```

Note: We skipped the `import_file "basic_elixir.ex"` line this time for brevity's sake, and we will continue to skip it, but you should keep calling `import_file` when you make changes.

This is a decent pattern for simple situations like what we have here, but is suboptimal once the complexity of your function starts to grow (specifically, the number and complexity of arguments passed to the function).

Second Solution: Arity

Another way you could attempt to do this is to use the or (`||`) functionality.

```
def recombine(phrase) do
  phrase = phrase || LearningElixir.phrase

  phrase
  |> String.split
  |> Enum.join("-")
end
```

In many languages, if you don't give a default value for an argument the default will automatically be `nil`. However, this is not the case in Elixir. Here's what happens if you try that:

```
iex(1)> LearningElixir.recombine()
** (UndefinedFunctionError) function LearningElixir.recombine/0
is undefined or private.
Did you mean one of:

  * recombine/1

LearningElixir.recombine()
```

It's saying that `LearningElixir.recombine/0` is undefined or private... what does that mean? Didn't we define `recombine`? And what's that 0 afterwards? The answer, as you may have guessed from this section header, is "arity".

"Arity" is a fancy word for the number of arguments a given function requires. So we've defined `LearningElixir.recombine/1` (a version of `recombine` with 1 argument), but not `LearningElixir.recombine/0` (a version of `recombine` with 0 arguments). This can be annoying when we're used to looser languages that

default to `nil` automatically, but it's really helpful for when we accidentally forget an argument- Elixir will help us catch that bug right at the start.

We can, of course, “cheat” the arity system by providing default values:

```
def recombine(phrase \\ nil) do
  phrase = phrase || LearningElixir.phrase

  phrase
  |> String.split
  |> Enum.join("-")
end
```

But the existence of the concept of arity, combined with Pattern Matching, opens up a better possibility.

Pattern Matching

Pattern Matching is a way to define a function multiple times and then run a specific definition based on the arguments given. One way to pattern match is by arity.

```
def recombine(phrase) do
  phrase
  |> String.split
  |> Enum.join("-")
end

def recombine do
  LearningElixir.phrase |> LearningElixir.recombine
end
```

Here we're defining two versions of `recombine`- the first with an arity of 1 and the second with an arity of 0. When we call it with an argument, we get the first version of `recombine`. When we call it without an argument, we get the second version- which then calls the first version and feeds it the phrase we defined earlier.

```
iex(1)> LearningElixir.recombine "hello world, universe"
      "hello-world,-universe"
iex(2)> LearningElixir.recombine
      "boldly-going-where-no-man-has-gone-before"
```

Pretty cool, right? There are lots of other ways you can employ pattern matching beyond simple number of arguments, and we'll go over those as they come up.

Exercises

1. Use pattern matching to add a 2-arity version of `recombine` which lets you input the join string.

```
iex(1)> LearningElixir.recombine("hello world, universe", " vast ")
           "hello vast world, vast universe"
iex(2)> LearningElixir.recombine("hello universe")
           "hello-universe"
```

2. Redefine the 1-arity version of `recombine` in terms of the 2-arity version.

That is, make it so that the 1-arity version calls the 2-arity version instead of repeating code. You'll note that we already defined the 0-arity version to call the 1-arity version.

Captain's Log: Function Ordering

Our scientists have arranged the above functions of different arities in many orders, and have found no difference in how the program runs.

However, in other situations the order does matter- Elixir will check the functions in order and use the first one that matches. So we have made it a habit to put the more specific cases earlier in the file and the more general cases later in the file.

Conclusion

Learning how to define functions with arguments (and using pattern matching to define multiple versions of a function) make our functions much more flexible and useful.

In the next chapter we'll introduce our first complex data type- Maps. Playing around with them will also lead us to the concept of Immutability- something very important in the functional programming world.

Maps and Immutability

Maps

Maps (not to be confused with the `Enum.map` function) are a very common programming construct, although they might be called “hashes” or “dictionaries” in other languages (or even, confusingly, “objects” in Javascript).

Here’s a basic map:

```
defmodule LearningElixir do
  def my_map do
    %{
      "name" => "Enterprise",
      "type" => "CodeShip",
      "mission" => "Code Boldly"
    }
  end
end
```

Captain’s Log: PascalCase vs snake_case

At first we thought the way the names were constructed was mere happenstance, a whim of the creators, but we have since determined a consistent pattern.

Modules, such as `LearningElixir`, are PascalCase. That means that each word that makes up the name is capitalized, and runs together with the other words with no separation.

Functions and variable names, such as `my_map`, are snake_case. That means that the words are entirely lower-case, and are separated by underscores.

It’s started with a `%{`, ended with a `}`, and in between consists of key-value pairs separated by commas. Each key-value pair has a key (such as “name”) before the `=>` symbol (sometimes called the “rocket”) and a value (such as “Enterprise”) after the `=>` symbol.

We can interact with this map using the functions in the `Map` module.

```
iex(1)> Map.get(LearningElixir.my_map, "mission")
          "Code Boldly"
iex(2)> Map.get(LearningElixir.my_map, "bad_key")
          nil
```

Our first (and most common) Map function, `Map.get`, takes two arguments: the map (`LearningElixir.my_map`) and a key (“mission”). It will then grab the value attached to that key in the map. If the key given doesn’t exist in the map, it will return `nil`.

You can use brackets as shorthand for `Map.get`.

```
iex(1)> LearningElixir.my_map["mission"]
"Code Boldly"
```

Another way to get an item from a map is to destructure it:

```
iex(1)> %{"mission" => mission} = LearningElixir.my_map
      %{mission: "Code Boldly", name: "Enterprise", type: "CodeShip"}
iex(2)> mission
      "Code Boldly"
```

More on destructuring in the next chapter.

The next most common Map function is to add new values to the map with `put`.

```
iex(1)> Map.put(LearningElixir.my_map, "captain", "Picard")
      %{"captain" => "Picard", "mission" => "Code Boldly",
        "name" => "Enterprise", "type" => "CodeShip"}
iex(2)> LearningElixir.my_map
      %{ "mission" => "Code Boldly", "name" => "Enterprise",
        "type" => "CodeShip"}
```

When we use `put`, it uses the second and third arguments as the key and value respectively, and adds them to the map. The result is seen after the first line. However, you’ll notice that when we call `LearningElixir.my_map` again, the new key-value isn’t there.

Immutability

That’s because data in Elixir is “immutable”. That means that each piece of data will never change. What `put` does is create a *new* piece of data and then assigns it to a variable.

To understand what “immutable” means, we’ll have to contrast it with data in a mutable language. Here’s some Javascript:

```
> var mutableArray = ["Don't", "try", "this", "at", "home"];
   undefined
> mutableArray.shift()
   "Don't"
> mutableArray
   ["try", "this", "at", "home"]
```

Don't worry if you don't understand the syntax of Javascript. The important thing is that you understand the differences between mutable and immutable languages, not the specifics of this example.

Above, we're assigning 5 strings to the variable `mutableArray`. Then we call the `shift` method on the `mutableArray`, which returns the first value in the array and has the *side effect* of removing that string from the array. The array now only has 4 elements.

Contrast that with immutable data, where there will never be side effects- no matter what you call on an array with 5 items, it will always be an array with 5 items.

Of course, the return value of the function can be something other than 5 items.

```
iex(1)> immutable_array = ["Immutability", "is", "great", "don't", "you", "agree"]
      ["Immutability", "is", "great", "don't", "you", "agree"]
iex(2)> Enum.slice(immutable_array, 4, 2)
      ["you", "agree"]
iex(3)> immutable_array
      ["Immutability", "is", "great", "don't", "you", "agree"]
```

So even though you call `Enum.slice` with `immutable_array` and get back an array with 2 items, `immutable_array` is still what it started as. There are no side effects.

Of course, there's a trick you can play with the data- take the result of the calculation and *immediately reassign it* to the variable you used.

```
iex(1)> immutable_array = ["Immutability", "is", "great", "don't", "you", "agree"]
      ["Immutability", "is", "great", "don't", "you", "agree"]
iex(2)> immutable_array = Enum.slice(immutable_array, 4, 2)
      ["you", "agree"]
iex(3)> immutable_array
      ["you", "agree"]
```

Oh no! Even naming it `immutable_array` didn't stop that disaster! How is this immutable data?

The array that `immutable_array` originally pointed to is still 5 items long. It's just that we told `immutable_array` to point to a new, different array- one that was the result of calling `Enum.slice` on the original `immutable_array`.

Imagine if you could start calling yourself Germany and then the entire country of Germany could no longer be found. Germany would still exist – you didn't mutate it – but everyone that was looking for Germany found you instead. All planes flying to Germany would go straight to your house. A disaster.

That's what it's like reassigning a variable.

One way around this is to keep on assigning stuff to new variables:

```
iex(1)> phrase = "boldly going where no man has gone before"
  "boldly going where no man has gone before"
iex(2)> phrase2 = String.split(phrase, " ")
  ["boldly", "going", "where", "no", "man", "has", "gone", "before"]
iex(3)> phrase3 = Enum.join(phrase2, "... ")
  "boldly... going... where... no... man... has... gone... before"
iex(4)> phrase
  "boldly going where no man has gone before"
iex(5)> phrase3
  "boldly... going... where... no... man... has... gone... before"
```

That method, however, can get tedious. That's one reason why the pipe (`|>`) construct is so popular in Elixir- it allows you to pass on the output of a function and use it as the first argument in the next function call, without the bother of naming it

```
iex(1)> phrase =
... (1)> "boldly going where no man has gone before" |>
... (1)> String.split(" ") |>
... (1)> Enum.join("... ")
  "boldly... going... where... no... man... has... gone... before"
iex(2)> phrase
  "boldly... going... where... no... man... has... gone... before"
```

Remember: in the command line we put the pipe at the end of the line, to let the Elixir interpreter know that we have more coming on the next line.

Now let's apply this to `Map.put`:

```
iex(1)> my_map = LearningElixir.my_map |>
... (1)> Map.put("captain", "Picard") |>
... (1)> Map.put("spock replacement", "Data")
  %{"captain" => "Picard", "mission" => "Code Boldly", "name" => "Enterprise",
    "spock replacement" => "Data", "type" => "CodeShip"}
iex(2)> my_map
  %{"captain" => "Picard", "mission" => "Code Boldly", "name" => "Enterprise",
    "spock replacement" => "Data", "type" => "CodeShip"}
```

We were able to feed the results of each line into the first argument of `Map.put`, then assign the whole thing to `my_map` without intermediate steps or explicit reassigments. This approach gives us a syntax as convenient as mutability, but with the stability and long-term simplicity of immutability.

Exercises

What will be the results from running each of the following functions?

```
def one do
  map = %{"hello" => "universe"}
  Map.put(map, "discarded", "data")
  map
end

def two do
  map = %{"hello" => "universe"}
  Map.put(map, "information", "data")
end

def three do
  %{"hello" => "universe"}
  |> Map.put("exploration_style", "bold")
  |> Map.put("starship", "Enterprise")
end

def four do
  %{"hello" => "universe"}
  |> Map.put("exploration_style", "bold")
  |> Map.put("starship", "Enterprise")
  |> Map.get("hello")
end
```

Be sure to check your work by copying the functions into a file and then running them.

Conclusion

In this chapter we introduced our most complex data-type to date: Maps. We learned how to get data from a map, and then how to update the data on a map- through creating a new copy of the map with updated data, because data in Elixir is immutable.

In the next chapter we'll introduce two more data types- Atoms and Tuples. They'll let us expand our range with pattern matching capabilities significantly, especially when combined with the `case` statement.

Atoms, Tuples, and Case Statements

In this chapter we'll cover three more basic Elixir concepts that are used often in Phoenix apps. We'll also combine them (and build on the concepts we've previously learned) in order to discover even more ways of pattern matching.

Atoms

Here's an example of an atom: `:name`.

Atoms are kind of like Strings, except that they can't be manipulated. "Name" could be fed into `String.upcase |> String.slice(1, 3)`, which would output "AME". `:name` will always and forever be `:name`, unless it is first explicitly turned into a String. Hence the name: an atom is irreducible.

At first glance, Atoms could be dismissed as a less-capable replacement for Strings... but their power is in their limitations, both in the signaling value to humans (it will not be manipulated) and in the conveniences that it allows.

One popular use of atoms is as the key in a Map. Here's what `my_map` from the last section would look like with a simple replacement:

```
defmodule LearningElixir do
  def my_map do
    %{
      :name => "Enterprise",
      :type => "CodeShip",
      :mission => "Code Boldly"
    }
  end
end
```

Now when we want to access the code, we use an atom instead of a String:

```
iex(1)> LearningElixir.my_map[:name]
"Enterprise"
iex(2)> LearningElixir.my_map["name"]
nil
```

But here's the first convenience that atoms unlock:

```
iex(3)> LearningElixir.my_map.name
"Enterprise"
```

This is much nicer than the clunky `["string"]` and `[:atom]` syntax.

You can also make your Map syntax nicer with atoms:

```
defmodule LearningElixir do
  def my_map do
    %{
      name: "Enterprise",
      type: "CodeShip",
      mission: "Code Boldly"
    }
  end
end
```

That looks much cleaner than what we had before, and works the exact same way when being called as the `:name => "Enterprise"` syntax.

So we've got a bit of convenience and syntax nicety- why else would we use atoms?

The primary benefit of atoms comes in their signaling value. A String can be manipulated- capitalized, split, reversed, etc. Elixir Strings are technically immutable (unchanging), but reassignment and piping make it easy for them to be manipulated without technically being mutated. On the other hand, if an atom starts as `:name`, it will go through the entire system as `:name` (unless explicitly turned into a String and then turned back into an atom).

Captain's log: changing an Atom

I have noticed that these Atoms, though supposedly indivisible, can be turned into a String or a Char List. This is, of course, a terrible idea, and should never be attempted. Not even for a handy plot point! Like the fourth wall, the integrity of atoms is very important to preserve.

Atoms are a perfect fit for keys in a Map construct. They're also great for pattern matching.

Pattern Matching with Atoms

```
defmodule LearningElixir do
  def my_map do
    %{
      name: "Enterprise",
      type: "CodeShip",
      mission: "Code Boldly"
    }
  end
end
```

```

def my_map(:voyager) do
  %{
    name: "Voyager",
    type: "Intrepid",
    mission: "Make it back"
  }
end

def my_map(:ds9) do
  %{
    name: "Terok Nor",
    type: "Station",
    mission: "Protect Bajor"
  }
end
end

```

Now we're matching not just on arity (number of arguments), but also on the value of the argument given (for versions with arity of 1).

```

iex(1)> LearningElixir.my_map
  %{mission: "Code Boldly", name: "Enterprise", type: "CodeShip"}
iex(2)> LearningElixir.my_map(:ds9)
  %{mission: "Protect Bajor", name: "Terok Nor", type: "Station"}
iex(3)> LearningElixir.my_map(:ds)
** (FunctionClauseError) no function clause matching in LearningElixir.my_map/1

The following arguments were given to LearningElixir.my_map/1:

# 1
:ds

iex:9: LearningElixir.my_map/1

```

Pattern matching is common with atoms, but it could be done with any datatype- Strings, numbers, even Maps.

One of the other common pattern-matching data-types is a Tuple.

Tuples

Tuples are simply collections of values, surrounded by curly braces:

```

iex(1)> my_tuple = {"Babylon", 5}
  {"Babylon", 5}

```

As you can see, the values can be of any type- including multiple types within one tuple. They can also be of any length, although lengths of 2 to 4 are most common.

You can access an element within a tuple using the `elem` function:

```
iex(2)> elem(my_tuple, 0)
      "Babylon"
```

As you can see, it's zero-indexed.

You can also decompose a tuple into its component parts:

```
iex(3)> {name, num} = my_tuple
      {"Babylon", 5}
iex(4)> name
      "Babylon"
```

This is a form of pattern matching, where the intent is to destructure the tuple into its component parts (sometimes this specific technique is just called “destructuring”).

This is similar to the Map destructuring we covered briefly in the last chapter. The biggest difference is that for a map you can destructure to as many or as few keys as you like, but in a tuple you have to destructure to the whole tuple. However, there are some conveniences- if you don't need one part of it, you can just use the underscore symbol instead of coming up with a throwaway name:

```
iex(5)> {name, _} = my_tuple
      {"Babylon", 5}
iex(6)> name
      "Babylon"
```

We're essentially throwing away the number by using the underscore. This saves us the trouble of naming it, and also communicates to future readers of our code which parts of the tuple will be used.

Technobabble: Assignment and Pattern Matching

We've called `name = {"Babylon", 5}` an “assignment”, but we called `{name, _} = {"Babylon", 5}` “pattern matching”. What sets them apart?

The truth is- they are both pattern matching. They're both attempts to make the left side equal to the right side.

When the left side is just one variable, then it's very easy to “match” to it- almost anything will do. This is “assignment” in Elixir and most other languages.

When there's a tuple or on the left side the attempt is more complex, and the attempt could fail, but it's essentially doing the same thing. A successful “destructuring” is when a complex left side of the equation successfully pattern matches with the right side.

It's possible for a destructuring attempt to fail.

```
iex(1)> {name, 5} = {"Babylon", 5}
          {"Babylon", 5}
iex(2)> name
          "Babylon"
```

In this first example, we're telling Elixir that "Babylon" is the variable `name`, and the number 5 is the number 5. Elixir agrees!

This second example is where things go wrong.

```
iex(1)> {name, "five"} = {"Babylon", 5}
** (MatchError) no match of right hand side value: {"Babylon", 5}
      (stdlib) erl_eval.erl:453: :erl_eval.expr/5
      (iex) lib/iex/evaluator.ex:257: IEx.Evaluator.handle_eval/5
      (iex) lib/iex/evaluator.ex:237: IEx.Evaluator.do_eval/3
      (iex) lib/iex/evaluator.ex:215: IEx.Evaluator.eval/3
      (iex) lib/iex/evaluator.ex:103: IEx.Evaluator.loop/1
      (iex) lib/iex/evaluator.ex:27: IEx.Evaluator.init/4
iex(2)> name
** (CompileError) iex:2: undefined function name/0
```

We're saying that "Babylon" is the variable `name`, and Elixir still agrees. But then we tell Elixir that the number 5 is the string `"five"`, which is false. Therefore Elixir throws an error.

You can see that if one part of the destructuring fails, they all fail. `name` isn't assigned.

Tuples in Function Definitions

Tuples can also be used to pattern match in function definitions:

```
defmodule LearningElixir do
  def take_action({:ok, _}, ship) do
    "Great job, #{ship}"
  end

  def take_action({:error, error_message}, ship) do
    "Problem with #{ship}.  #{error_message}"
  end
```

```
    end
end
```

Here both variations on the function have two arguments, the second of which is a ship and the first of which is a tuple. The tuple has two values- an atom and a string. The atom is either :ok or :error, and the string is used when the action is a failure but thrown away when the action is successful.

```
iex(1)> LearningElixir.take_action({:ok, "Make it so"}, "Enterprise")
      "Great job, Enterprise"
iex(2)> LearningElixir.take_action({:error, "Shields are at 38 percent!"}, "Enterprise")
      "Problem with Enterprise. Shields are at 38 percent!"
```

Matching a tuple with an :ok or :error atom as the first value is very common in Phoenix apps.

Conditionals with case

There are many other ways we could go about coding `take_action`. Probably the most fitting is the `case` statement.

Here's a super simple example of a case statement:

```
def rabbit_counting(number) do
  case number do
    0 -> "none"
    1 -> "one"
    2 -> "two"
    3 -> "three"
    4 -> "four"
    _ -> "many"
  end
end
```

A case statement has two parts- the expression and the clauses. In this case statement, the expression is `number`, and there are six clauses.

Each clause has a head and a body. The head (such as 1 or `_`) is before the arrow `->`, and is what matches against the expression. The body is what's after the arrow `->`. When a head matches, the body of that clause is returned.

So if you pass in 1, you get "one" returned. If you pass in 4, you get "four" returned.

What happens if you pass in 5? The underscore (`_`) in the head acts as a default, accepting anything. So if we pass in 5, it will be caught by the underscore and return "many". Be careful about that, though- if you put the underscore as

the first clause, then it will catch *everything*, not letting any of the other more specific clauses match.

Now let's apply the case statement to our tuple pattern-matching example:

```
defmodule LearningElixir do
  def take_action(tup, ship) do
    case tup do
      {:ok, _} -> "Great job, #{ship}"
      {:error, error_message} -> "Problem with #{ship}. #{error_message}"
    end
  end
end
```

If `{:ok, _}` matches, then `"Great job, #{ship}"` will be returned, with `ship` being filled in with whatever was passed in to the function.

Case statements of this form are very common in Phoenix apps, so you'll have plenty of time to get used to how they work.

Other Conditionals

There are two other common way of doing conditionals in Elixir: `cond` and `if`. However, they aren't used nearly as often as `case` within Phoenix, so if we use them then we'll introduce them at that time.

This may be a bit of a shock to people from other languages who are used to using `if` for everything, but in Elixir `if` is used sparingly. Most of the common usages of `if` can be done better by some sort of functional conditional- usually, but not always, involving pattern matching.

Exercises

1. For the following code:

```
{:ok, phaser_setting, _} = {:ok, "stun", "thank you"}
%{name: my_name} = %{rank: "Captain", name: "Picard"}
```

What is the value of `phaser_setting` and `my_name`?

Hint: Rememmrber the Map destructuring from the previous chapter

2. Type out our final version of the `take_action` function. What happens when you call it as follows? Why?
 - `LearningElixir.take_action({:err_bear, "Doctor, why is tummy glowing? This seems serious."}, "Enterprise")`
 - `LearningElixir.take_action("no tuple here", "Enterprise")`

- c. LearningElixir.take_action({"ok", "I have made it so"}, "Enterprise")
- d. LearningElixir.take_action({:ok, "I have made it so", "another part of the tuple, how fun"}, "Enterprise")
- e. LearningElixir.take_action({:ok, "I have made it so"})
- f. LearningElixir.take_action({:ok, "I have made it so"}, "Enterprise")

Learning the common error modes is important- better that you do it now while your program is small.

- 3. Modify `take_action` so that the ship is passed as a third part of the tuple.

```
iex(1)> LearningElixir.take_action({:ok, "I have made it so", "Enterprise"})
"Great job, Enterprise"
iex(2)> LearningElixir.take_action({:error, "Phasers not set to stun", "Enterprise"})
"Problem with Enterprise. Phasers not set to stun"
```

Conclusion

Now, with the introduction of atoms, tuples, and condition statements, we're finally starting to see functions that might look at home in a Phoenix app. In fact, our example for the case statement was inspired by the auto-generated Controller in Phoenix 1.2 and before (they changed the generator in Phoenix 1.3, but it still depends on the concepts we've introduced in this chapter).

Great job! You've already learned quite bit!

End of Part 1

That's most of the Elixir you need to know in order to understand a basic Phoenix app.

There's plenty of Elixir that I've left out, including some stuff that's used within Phoenix. I've left out several commonly-used things like `Enum.map` and anonymous functions because we're going to get pretty far into our app before we start needing them. And of course there's all the advanced topics- Channels, Metaprogramming, OTP, etc.- because while they're really cool, they have no place at the start of a beginner's guide. Finally, I've left out some of the more project-focused stuff- Import, Use, configuration files, various mix commands- because those are better understood while in the context of a full Phoenix app.

What that means is that our crash course in Elixir is finished- it's time to get started building a Phoenix app!

Captain's Log: more Elixir resources.

I have heard tell of great stores of knowledge, locked away in interconnected datapads and inscribed upon dead lumber. Here are the rumors which I hear most strongly:

- Programming Elixir 1.6- An excellent book for programming language nerds and others who want to geek out on details of the Elixir language.
- Programming Phoenix 1.4- A book which quickly and efficiently introduces the major parts of the Phoenix framework. It's meant for advanced coders who are familiar with Elixir and with backend web frameworks such as Rails or Laravel.

Were I a younger captain I might go seeking after these sources of knowledge myself, but I am lucky to have discovered even this one document, and in my age I know that deciphering it is much more important than some fool's errand. We must know how the Phoenix-class ships were constructed!

Now that we understand the basics of Elixir, it's time to install Phoenix and create our first app.

Installation

We're going to be using Phoenix version 1.4.4 in this tutorial. We download it straight from github using Mix (for now, just copy this line. We'll go over more about how version control and git/github work in the next chapter):

```
$ mix archive.install hex phx_new 1.4.4
```

Node

We rely on Webpack to compile static assets, and Brunch relies on Node.

Go to the NodeJS site and download version 5.0 or later. The current stable release (recommended) is 10.15.3 LTS.

Postgres

Postgres is the database we'll be using for this tutorial, although you're free to choose another SQL database if you're more comfortable there.

Installation guides are found on the Postgres wiki.

I personally use PostgresApp on my mac.

Your First App

Much of this section is taken from the Up and Running docs.

In general, the first few chapters in Section 2 are gluing together a bunch of Getting Started Guides, but after that we'll get to the more exciting Original Content.

We'll first use Mix to generate a new Phoenix app, named star_tracker:

```
$ mix phx.new star_tracker
```

This will create a bare-bones app

```
* creating star_tracker/config/config.exs
* creating star_tracker/config/dev.exs
....
* creating star_tracker/lib/star_tracker_web/controllers/page_controller.ex
* creating star_tracker/lib/star_tracker_web/templates/layout/app.html.eex
* creating star_tracker/lib/star_tracker_web/templates/page/index.html.eex
....
* creating star_tracker/assets/static/images/phoenix.png
* creating star_tracker/assets/static/favicon.ico
```

You can start looking around your file structure. We'll take our first quick tour in chapter 2.3, and start giving detailed explanations of what everything does starting in chapters 2.4 and 2.5.

You'll be prompted to fetch and install "dependencies". Say Yes with Y (or hitting enter to choose Y as default).

```
Fetch and install dependencies? [Yn] Y
```

It will then give you instructions for finishing installation

```
Fetch and install dependencies? [Yn] Y
* running mix deps.get
* running mix deps.compile
* cd assets && npm install && node node_modules/webpack/bin/webpack.js --mode development
```

We are all set! Go into your application by running:

```
$ cd star_tracker
```

Then configure your database in config/dev.exs and run:

```
$ mix ecto.create
```

Start your Phoenix app with:

```
$ mix phx.server
```

You can also run your app inside IEx (Interactive Elixir) as:

```
$ iex -S mix phx.server
```

We'll follow those instructions

```
$ cd star_tracker
```

The first command Changes Directories to the newly-created folder. cd is a unix shell command, but don't let the name scare you- "Unix" shell commands can

be used on Linux, Mac, and Windows 10+ (with Bash).

Here's some basic commands to get you started

Command	Description	Example
ls	lists files and folders in directory	ls -a
cd	change directory	cd star_tracker
mkdir	create directory	mkdir new_folder
mv	move file	mv old_location new_location
cp	copy file(s)	cp old_file new_file
rm	remove file	rm old_file

In Unix terminology . means “current directory”, .. means “one directory up”, and ~ means the home directory. So to move up a directory we would input cd .., and to look at the files in the home directory we would input ls ~.

There's lots more Unix shell commands we could learn, but those should be enough to get you started. Now back to installing Phoenix!

```
$ mix ecto.create
Compiling 13 files (.ex)
Generated star_tracker app
The database for StarTracker.Repo has already been created
```

Ecto is the library that Phoenix uses to interface with the database. We'll be learning a lot more about Ecto when creating our main app.

This particular command creates the database (Postgres by default) for our app.

If you get an error, it's probably because you either don't have Postgres installed or don't have Postgres running. See the instructions for whichever installation of Postgres you chose.

Finally, we run our app.

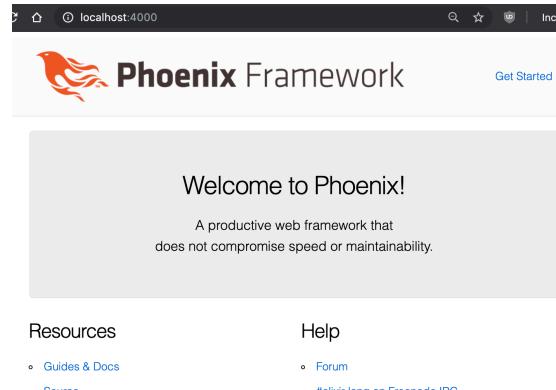
```
$ mix phx.server
[info] Running StarTrackerWeb.Endpoint with cowboy 2.6.3 at 0.0.0.0:4000 (http)
[info] Access StarTrackerWeb.Endpoint at http://localhost:4000

Webpack is watching the files...

Hash: 1fc94cc9b786e491ad40
Version: webpack 4.4.0
Time: 486ms
Built at: 2019-5-2 09:05:06
.....
```

In your web browser, visit <http://localhost:4000/> (it's the same address as

`http://0.0.0.0:4000`). There you should see the Phoenix Welcome page.



And that's your first Phoenix app! Congratulations!

In the next chapter we'll go over how to save your code with Git and Github, then how to deploy your site, making it publicly available on the internet.

Beam Me Up (Version Control and Deployment)

This chapter contains a lot of material that isn't directly related to Elixir and Phoenix, but is nonetheless extremely important to know. It also contains a lot of material that is specific to certain services and operating systems- meaning that I won't be able to provide good instructions for Windows on Linux, and the instructions for these services could change independently of the Phoenix framework's upgrade cycle.

For those reasons, I'll be linking to several external guides that cover these very important subjects much more thoroughly than I could, and which will be more likely to stay up to date and cover all operating systems. What I'll be writing are the parts that won't change either with time or with switching between operating systems.

If you're already familiar with version-control systems, feel free to skim until you get to the end, where we discuss deployment.

So let's dive in!

What is Version Control?

Let's say you're make a mistake. A horrible, life-changing mistake. In real life you'd just have to deal with the consequences, be they jail, debt, injury, or a lifetime of shame. But in code, you can make (most of) your mistakes disappear with just a few taps at the command line... IF you're using version control.

Using version control, you can go back in time, create alternate timelines, combine the best parts of alternate timelines, see your entire history, and even work together with other people separate from you in time and space. This chapter will teach you the absolute basics – we'll be treating commits kind of like save files in a videogame – but there's a whole world of advanced tricks that are really useful, especially when you collaborate with other coders.

The version control system we'll be using is called "Git".

Install Git

To see if you have `git` already installed, type `git` on the command line. If it shows you the most commonly used commands, then congratulations- you have `git` already installed. If it says something like "`command not found: git`" then you'll need to install it. This is a pretty good guide: <https://www.atlassian.com/git/tutorials/install-git>.

Initializing a Git Project

First, let's check the status of our project according to git. Make sure you're in the `star_tracker` folder and then type the following:

```
$ git status  
fatal: Not a git repository (or any of the parent directories): .git
```

This is an error, but it's one we expect. We don't have a git repository yet! Let's make one.

```
$ git init  
Initialized empty Git repository in /Users/jeffreybiles/phoenix/star_tracker/.git/
```

Your filepath will be different, but it should say something similar. It creates a `.git` folder, where it will store all your git data.

The dot in the folder name means it's "hidden", so it may not show up by default. You can fix that by using the `-a` option in your `ls` command.

```
$ ls  
README.md assets deps mix.exs priv  
_build config lib mix.lock test  
$ ls -a  
. .gitignore assets lib priv  
.. README.md config mix.exs test  
.git _build deps mix.lock
```

We won't worry about the specifics of the `.git` folder in this book, just know that deleting it is a very bad idea unless you know what you're doing.

Our First Commit

Let's ask git about our status again.

```
$ git status  
On branch master  
  
No commits yet  
  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
.formatter.exs  
.gitignore  
README.md  
assets/  
config/
```

Our First ComBEAM ME UP (VERSION CONTROL AND DEPLOYMENT)

```
lib/
mix.exs
mix.lock
priv/
test/

nothing added to commit but untracked files present (use "git add" to track)
```

We have lots of “untracked” files, which means that git currently has no record of them. Let’s go ahead and add all of them to our git project.

```
$ git add .
```

The dot here means “current directory”, and since we’re in the root directory of the project this command translates into “add everything”.

```
$ git status
On branch master

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)

new file:   .formatter.exs
new file:   .gitignore
new file:   README.md
new file:   assets/.babelrc
new file:   assets/css/app.css
new file:   assets/css/phoenix.css
.....
new file:   lib/star_tracker/application.ex
new file:   lib/star_tracker/repo.ex
new file:   lib/star_tracker_web.ex
.....
new file:   lib/star_tracker_web/views/page_view.ex
new file:   mix.exs
new file:   mix.lock
.....
new file:   test/test_helper.exs
```

Calling for the status shows that git now has all these new files ready to be committed – or, in the parlance of git, “staged”.

Let’s take these staged files and commit them.

We’ll use the `git commit` command, and pass the “message” option with `-m` and a string that will be used for your commit message.

```
$ git commit -m "Our first commit"
[master (root-commit) 6e6ab4a] Our first commit
47 files changed, 12588 insertions(+)
create mode 100644 .formatter.exs
create mode 100644 .gitignore
create mode 100644 README.md
create mode 100644 assets/.babelrc
create mode 100644 assets/css/app.css
create mode 100644 assets/css/phoenix.css
.....
create mode 100644 lib/star_tracker/application.ex
create mode 100644 lib/star_tracker/repo.ex
create mode 100644 lib/star_tracker_web.ex
.....
create mode 100644 lib/star_tracker_web/views/page_view.ex
create mode 100644 mix.exs
create mode 100644 mix.lock
.....
create mode 100644 test/test_helper.exs
```

So now we've committed 47 files with a total of 12588 lines of code between them (your numbers may be slightly different depending on the point version of Phoenix that you use, but they shouldn't be too far off). Thanks git for those handy stats, and thanks Phoenix for not making us write all that code ourselves!

Let's see what our status is now.

```
$ git status
# On branch master
nothing to commit, working tree clean
```

This tells us that we've committed all the changes. Everything is saved.

Your Second Commit

Let's make a small change, then create a new git commit.

We'll make our change in README.md. Go in that file and at the end add the following:

```
## Custom Instructions
```

We're going to add our custom instructions here.

It's some nonsense stuff, but the important part is that, once you save the file, we'll have more changes.

Looking at His README UP (VERSION CONTROL AND DEPLOYMENT)

Go ahead and check the git status again:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Now let's add the file.

```
$ git add README.md
```

Notice that now we're adding a specific file to the staged changes, instead of adding all available files like we did previous with `git add ..`. The effect would be the same in this case, since that's the only file changed, but I wanted to show that you could be more selective in adding to a commit.

Let's check out status.

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md
```

It shows that we've modified the README file.

Now let's commit it.

```
$ git commit -m "update the readme to add custom instructions"
[master 17f7669] update the readme
 1 file changed, 4 insertions(+)
```

Congrats! You made your second commit!

Looking at Historical Logs

Now that we've made some commits, let's look at our history by typing in `git log`

```
$ git log
commit 17f76691b8ebdc192f0e7028f22c04912bdc9efe (HEAD -> master)
Author: jeffreybiles <bilesjeffrey@gmail.com>
```

Stashing your ~~BREAK ME~~ UP (VERSION CONTROL AND DEPLOYMENT)

```
Date: Thu May 2 09:59:21 2019 -0500
```

```
update the readme to add custom instructions
```

```
commit 6e6ab4adb3198b2f0d6b370a63aa6a2d8d217c40
```

```
Author: jeffreybiles <bilesjeffrey@gmail.com>
```

```
Date: Thu May 2 09:45:30 2019 -0500
```

```
Our first commit
```

The specifics will be different for you, but the format should be similar. You'll see

- the commit hash. Think of this like the key for your save file.
- the author. This will always be you during this project, but this is very useful data when collaborating.
- the time the commit was made.
- the commit message.

All of these are useful, but the commit message varies the most wildly in usefulness. If you have nice commit messages that are clear about what has changed in that commit, then navigating your history will be a breeze.

When you're doing looking at the log, press q to exit.

Stashing your Mistakes

Sometimes you make mistakes. Horrible, horrible mistakes.

Let's make one of those mistake... let's delete our entire config directory.

```
$ rm -rf config
```

Oh no! It's gone! Our app doesn't run!

First, let's check out status:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:  config/config.exs
    deleted:  config/dev.exs
    deleted:  config/prod.exs
    deleted:  config/test.exs
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Cool, so git still knows about the files, even though they're deleted.

Git can do even better... it can make it like the mistake never happened. It's like going back to your last save file.

```
$ git stash -u
Saved working directory and index state WIP on master: 17f7669 update the readme
```

If you check the status again, you'll see that the files are no longer deleted.

```
$ git status
On branch master
nothing to commit, working tree clean
```

And if you check your files, you'll see the config directory is back. Hurrah!

By the way, the `-u` option makes it work with files that were added or deleted. Without that, it would just handle files that were modified.

Putting your git repository online

You've got the git repository on your computer, but the real power comes when you can share it with the world and work with other people.

The de-facto git management service is Github. That's what I'll be using. However, there are plenty of alternatives such as bitbucket and gitlab. Use whatever you're most comfortable with.

Create a Github account and repository

Go to github and signup: <https://github.com/join>.

You'll be fine with a free subscription.

Add your Public Key to Github # TODO

I'll be doing this and the next three sections when I create the videos, for logistical reasons.

Create your first repository # TODO

Push to Github # TODO

Deployment

“Deployment” is the process of making your app available to the wider world. In this case, it’s getting it to a web server where people can view your website (currently a barely-customized default app) by putting a URL in their browser.

For the purposes of this tutorial we recommend using either Render or Heroku. They both use git to receive your code, and do a lot of the nitty-gritty of server management for you; perfect for beginners or people who are too busy to bother with handling the fine details.

Both of these options have good online instructions, and I don’t really have much to add beyond a comparison of the pros and cons of each. However, seeing someone else follow the instructions could be useful, so I’ll record a video of me deploying an app in the video course.

Heroku

Heroku is the standard “easy deploy” choice. Phoenix even has official docs (<https://hexdocs.pm/phoenix/heroku.html>). You won’t really go wrong using Heroku as a starter, and when you’re just getting started out it’s free.

There are three downsides to using Heroku:

- the hoops you’re required to jump through have become slowly more arcane
- it’s not built for Phoenix, so you’re going to lose some scalability
- it can get expensive later on

Render

Render is a newer alternative that gets rid of some of the cruft that’s built up around Heroku. In general it’s cheaper, but to get started you’ll need their \$5/month DB plan.

One of the big selling points is that any push to github’s master branch will automatically deploy your app... no extra setup, no extra commands. For a lean team that has a good gitflow-like setup, this can be really nice.

Their official instructions (<https://render.com/docs/deploy-phoenix>) are good, and the founder is very responsive on slack.

Others

There are plenty of other deployment options. If you have the technical expertise, you can set up on a custom server and start running stuff. There's also specialized options like gigalixir. If you really want video instructions for a specific setup, send me an email and let me know: jeff@happyprogrammer.net.

Onward!

See you in the next chapter, where we start customizing our app.

Customizing Our Page

Now that we've got our foundational Elixir knowledge, our app is generated, and we're all set up on github and Heroku, it's time to start building our app. We'll start with some basic customization work, getting used to some of the building blocks of a Phoenix page.

If you've worked on html pages before then much of this chapter will be review—but I'd encourage you to still at least skim the chapter in order to pick up the Phoenix-specific parts, and to put the given code in your own application in order to keep up your project in sync with ours.

Technobabble: What about client-side Javascript applications?

The best part of If you do decide to use a client-side Javascript web framework, 80% of the knowledge in this book should still apply, and Phoenix is great at creating APIs. However, Phoenix LiveView has shrunk the range of cases where a full client-side web framework is required. If you want your page to work offline, or to have extra-rich interactions, then you'll still want a client-side web framework, but if you just want some basic interactivity (the “sprinkles of javascript” use case) then Phoenix LiveView has you covered.

The index page

We'll start with the autogenerated index page (`lib/star_tracker_web/templates/page/index.html.eex`). This is what shows up when you visit `localhost:4000`.

```
<section class="phx-hero">
  <h1><%= gettext "Welcome to %{name}!", name: "Phoenix" %></h1>
  <p>A productive web framework that<br/>does not compromise speed or maintainability.</p>
</section>

<section class="row">
  <article class="column">
    <h2>Resources</h2>
    <ul>
      <li>
        <a href="https://hexdocs.pm/phoenix/overview.html">Guides &amp; Docs</a>
      </li>
      <li>
        <a href="https://github.com/phoenixframework/phoenix">Source</a>
      </li>
      <li>
```

```

        <a href="https://github.com/phoenixframework/phoenix/blob/v1.4/CHANGELOG.md">v1.4 Changelog</a>
    </li>
</ul>
</article>
<article class="column">
    <h2>Help</h2>
    <ul>
        <li>
            <a href="https://elixirforum.com/c/phoenix-forum">Forum</a>
        </li>
        <li>
            <a href="https://webchat.freenode.net/?channels=elixir-lang">#elixir-lang on Freenode</a>
        </li>
        <li>
            <a href="https://twitter.com/elixirphoenix">Twitter @elixirphoenix</a>
        </li>
    </ul>
</article>
</section>
```

Let's delete all that, replace it with something simpler and more manageable:

```

<div class="phx-hero">
    <h2>Star Tracker</h2>
    <p class="lead">Track and trade resources related to star formation.</p>
</div>
```

You'll notice that the page shown at `localhost:4000` has changed automatically- this is because Phoenix uses livereload, a cool tool that saves us time and a click each time we change something in development.

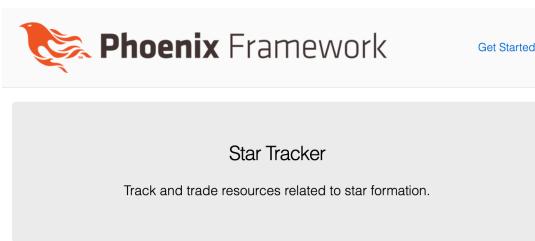


Figure 3: Picture of customized page

Let's break down what's happening here.

First, the file ending is `.html.eex`. The `.html` part means that it's an html web page. The `.eex` part means that it's the templating language "Embedded Elixir". Soon we'll be using Elixir to customize our templates on the fly, using

information from cookies, the database (both explained later), and other sources to create situation-specific html pages that are served to the user.

Then we have the html itself. Embedded Elixir is a superset of html- that is, an html page is a valid Embedded Elixir page, but then Embedded Elixir adds things on top. The current page is just plain html.

Previously On: HTML

The basic building block of HTML are “elements”. In the example above we see the “div”, “h2” (headline 2), and “p” (paragraph) elements. Each of those elements has an opening tag (“`<h2>`”), a closing tag (“`</h2>`”), and some content in between (“Star Tracker”). The opening tag is the element name in between a less-than sign and a greater-than sign. The closing tag is the same, but with a slash (“`/`”) added right before the element name.

There are some elements, such as “`
`” (break) that are standalone and do not have content or a closing tag. These are sometimes written as self-closing (“`
`”). When in doubt, use a closing tag.

Elements can have “attributes”. The attribute we see in the example above is “class”. Above we can see that the div has been given the class of “phx-hero” and the paragraph element has been given the class of “lead”. These are used by the css (we’ll go over that later) to style the code and change how it looks, and sometimes by other parts of the code as well.

We’ll try to briefly touch on everything as we go, but if this information about HTML is new to you and you want to get more familiar before moving on with this book, you can learn more at TODO: Find a good beginner html resource

So that’s the Index template... but you’ll notice that there’s other stuff showing on the page. Even if you delete everything in the index template there will still be content on the page. That’s because of the layout template.

The layout template

Wrapping everything is the app layout template. You can find it at `lib/star_tracker_web/templates/layout/app.html.eex`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```

<meta charset="utf-8"/>
<meta http-equiv="X-UA-Compatible" content="IE=edge"/>
<meta name="viewport" content="width=device-width, initial-scale=1.0"/>
<title>StarTracker · Phoenix Framework</title>
<link rel="stylesheet" href="<%= Routes.static_path(@conn, "/css/app.css") %>">
</head>
<body>
  <header>
    <section class="container">
      <nav role="navigation">
        <ul>
          <li><a href="https://hexdocs.pm/phoenix/overview.html">Get Started</a></li>
        </ul>
      </nav>
      <a href="http://phoenixframework.org/" class="phx-logo">
        " alt="Phoenix Fr">
      </a>
    </section>
  </header>
  <main role="main" class="container">
    <p class="alert alert-info" role="alert"><%= get_flash(@conn, :info) %></p>
    <p class="alert alert-danger" role="alert"><%= get_flash(@conn, :error) %></p>
    <%= render @view_module, @view_template, assigns %>
  </main>
  <script type="text/javascript" src="<%= Routes.static_path(@conn, "/js/app.js") %>"></sc
</body>
</html>

```

A lot of this should look familiar if you've worked on any web apps before.

You've got your enveloping `html` tags, which contain the `head` and `body` tags.

`head` contains a title (shown in the browser tab), a link to the css stylesheet, and lots of `meta` tags. We'll leave those all be for now- customizing them is mostly for SEO (Search Engine Optimization) purposes and is the same across web frameworks.

The `body` tag contains what's actually shown on the page.

First, we've got a `header` tag that contains navigation links.

Then after that, we've got a `main` tag with role `main` and class `container` that contains and wraps all our other content. Within the container div we've got a header, two alert paragraphs, and a render block (`<%= render @view_module, @view_template, assigns %>`) which will render everything else in the app.

Finally, we've got a script tag that contains any javascript we've written (we'll explain later when we get there).

Go ahead and delete the header so our `body` tag looks like this:

```
<body>
  <main role="main" class="container">
    <p class="alert alert-info" role="alert"><%= get_flash(@conn, :info) %></p>
    <p class="alert alert-danger" role="alert"><%= get_flash(@conn, :error) %></p>
    <%= render @view_module, @view_template, assigns %>
  </main>
  <script type="text/javascript" src="<%= Routes.static_path(@conn, "/js/app.js") %>"></script>
```

Now we're left with the two flash paragraphs and the renderer. The flash paragraphs only show up when we want to display a flash message. The renderer is where the rest of the app is placed - it's where the index page we worked on before goes.

To demonstrate that, let's put some marker text before and after the render block.

```
<main role="main" class="container">
  <p class="alert alert-info" role="alert"><%= get_flash(@conn, :info) %></p>
  <p class="alert alert-danger" role="alert"><%= get_flash(@conn, :error) %></p>
  <p>Before Render</p>
  <%= render @view_module, @view_template, assigns %>
  <p>After Render</p>
</main>
```

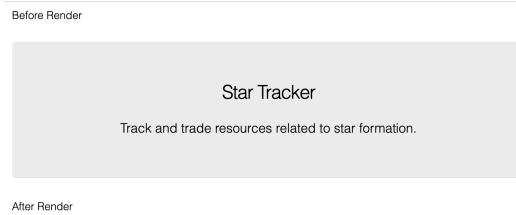


Figure 4: Index page wrapped

So that's the basic layout of the page. The only thing left to go over is the `<%= %>` construct.

Embedded Elixir

The `<%= %>` is the Embedded Elixir part of “Embedded Elixir templates”. Any time you see that you know that your Elixir app is feeding custom information into the template.

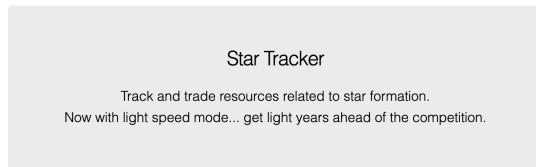
`<%= render @view_module, @view_template, assigns %>` is what displays

the index page (or another page, depending on the URL). Other Embedded Elixir blocks in the layout put in Javascript, CSS, and alert messages.

It's okay if you don't understand all of that right now- in the upcoming chapters, we'll be starting with simple Embedded Elixir and building up from there.

Exercises

1. Update your app to what we have (except for the “Before Render” and “After Render” stuff... leave that out)
2. After the lead paragraph element in the index page, add a new paragraph that says “Now with warp-speed mode... get light-years ahead of the competition”.



Extra Resources

- TODO: find a good HTML course

Conclusion

In this chapter we looked at the index page and the default app layout page (we can have alternate layouts within an application). We reviewed HTML, customized both of the aforementioned pages, and got our first taste of Embedded Elixir.

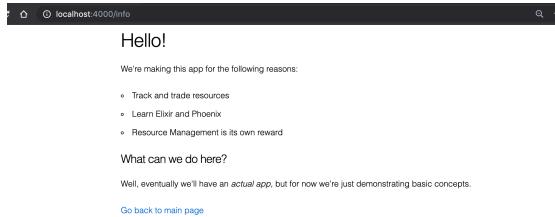
This chapter was either 80% review or a firehose of new information, depending on your background in web apps. As we go on through the book those difficulty levels should both adjust to the middle, but while we're in these opening parts it's best to recognize where you are and either skim or do extra research (see the Extra Resources section) based on how you're feeling about the material.

Next we're going to create our own custom page. This will let us trace through the basic path of a Phoenix web request and introduce us to the Router and a Controller.

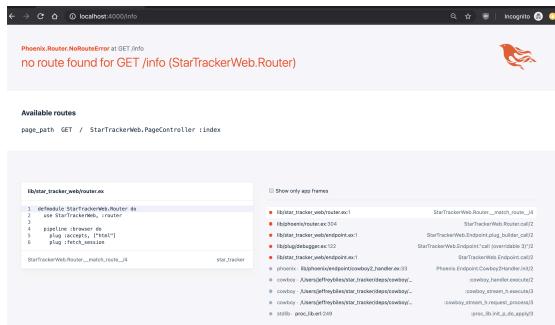
A New Page

In this chapter we're going to add our first custom Route – an Info page – giving us practice in the basics of html we learned last time, as well as our first look at a more full cycle of a request in Phoenix.

Here's what it will look like at the end of this chapter:



But this is what it would look like right now if you tried to visit `localhost:4000/info`:



The error message is “no route found for GET /info (StarTrackerWeb.Router)”.

The Router

Let's crack open our Router file, in `lib/star_tracker_web/router.ex`, and start exploring. It should look like this:

```
defmodule StarTrackerWeb.Router do
  use StarTrackerWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
```

```

end

pipeline :api do
  plug :accepts, ["json"]
end

scope "/", StarTrackerWeb do
  pipe_through :browser

  get "/", PageController, :index
end

# Other scopes may use custom stacks.
# scope "/api", StarTrackerWeb do
#   pipe_through :api
# end
end

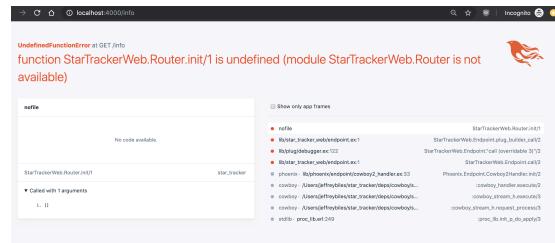
```

We'll go over this line by line before making our edit.

The Router Name

First, notice the module name: `StarTrackerWeb.Router`. Many many module names in our app will start with `StarTrackerWeb`- it's a good indication that what we're using is local to our Phoenix app. Then the second part- `Router` tells us the specific usage of this module.

Both parts of the name are very important, and Phoenix will freak out if you change either one. Try it! Change it to `StarTrackerWeb.Diverter` or something and Phoenix will immediately start asking where `StarTrackerWeb.Router` went.



If you start getting messages like that (“Module `StarTrackerWeb.X` is not available”) the most probable cause is that you misnamed something.

Let's change it back and then move on.

Macros

The next line is `use StarTrackerWeb, :router`. We'll go into detail in the next chapter on how that works, but for now just know that that's how we get the `pipeline`, `plug`, `scope`, `pipe_through`, and `get` macros used later in the file.

Technobabble: Macros

Macros are a cool advanced Elixir feature that give us more power and syntactical freedom than regular functions and let us define a DSL (Domain Specific Language).

While we won't be defining our own Macros in this book, we'll be taking advantage of lots of them that are built into Phoenix- the items from `StarTracker.Web, :router` are just the first.

If you're an advanced coder, I'd encourage you to research Macros for yourself. A good resource for this is Metaprogramming Elixir by Chris McCord (the creator of the Phoenix framework). It's a short but advanced book- if you had any trouble with chapters 2.2-2.4, I recommend waiting until the end of this book, and possibly reading a more detailed Intro to Elixir book first.

The first of those macros is `pipeline`. We define two of them: `browser` and `api`. Each has a series of `plugs` – a set of stacked instructions to run on each request (we'll go over plugs later in the book) – that provide helpful functionality for the specific type of request we're running.

The Routes Themselves

Next we see the following:

```
scope "/", StarTrackerWeb do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
end
```

The `scope` macro takes two arguments and a block.

The first argument is the base url- `/` here, so effectively nil- and the second argument is the app that will serve in this scope- `StarTrackerWeb`. Everything else in the scope will be prefixed with that (for example, `PageController` will actually be `StarTrackerWeb.PageController`). The block is everything between `do` and `end`, and where we use `pipe_through` and `get`.

`pipe_through :browser` says that within this scope, we'll be using the `browser` pipeline that was defined earlier in the file.

`get` takes 3 arguments- the url, the controller, and the function. Here the url is `/`, the controller is `PageController`, and the function is `:index`. What this means is that if a GET request is sent to the url `/`, then we'll respond with the `index` function on `PageController`.

Previously on: Request Types

GET is only one of several types of requests available. It's the most common, but other common types include POST, PUT, and DELETE.

Generally GET is used when you want information from the server but aren't requesting that the server make any changes.

We'll cover the other request types later when we start using them.

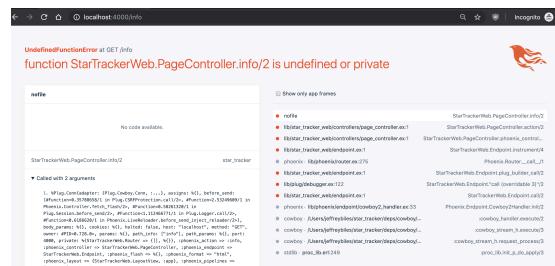
Our New Route

Let's define our own route now- `info`. It'll be a `get` request, since we don't need the server to make any changes. We'll want the url to be `/info`, we can re-use the `PageController`, and we'll call our function `:info`.

```
scope "/+", StarTrackerWeb do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
  get "/info", PageController, :info
end
```

Now if we try to visit `/info`, we'll get a different error!



It says "function `StarTrackerWeb.PageController.info/2` is undefined or private". Time to define it!

The Controller

First, let's look at our current code for `PageController`:

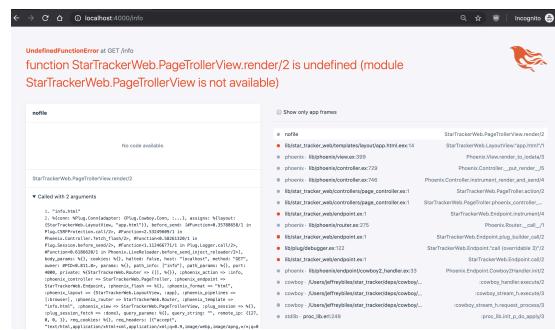
```
defmodule StarTrackerWeb.PageController do
  use StarTrackerWeb, :controller

  def index(conn, _params) do
    render conn, "index.html"
  end
end
```

The `defmodule` is `StarTrackerWeb.PageController`- the naming of which is, once again, is very important. Try changing the name if you don't believe me.

Don't Change the Name

Even if you also change the name in the `get` function, it will still complain. Let's change it in both places to `PageTroller` (both in the Controller `defmodule` and in the `get` function for `:index`) and see what happens.



The error is “function `StarTrackerWeb.PageTrollerView.render/2` is undefined (module `StarTrackerWeb.PageTrollerView` is not available)”. It's looking for a `StarTrackerWeb.PageTrollerView` module that doesn't exist. If we *really* wanted to change the name, we'd have to go change two more things: the view and the name of a templates folder.

But we don't want to change the names, so go ahead and change them back if you messed around with them. There's rarely a good reason to stray from the conventions that Phoenix recommends.

Our Current Controller Function

We'll once again skip over the line with `use` (`use StarTrackerWeb, :controller`), leaving it to the next chapter, and move on to our `index`

function.

```
def index(conn, _params) do
  render conn, "index.html"
end
```

Each Phoenix Controller function takes two arguments: connection and parameters (`conn` and `_params` in this example). We'll go over the connection in more detail in later chapters, but right now we just need it to feed to the `render` function. `_params`, on the other hand, is not needed. Starting an argument name with `_` is a great way to signal to future readers of your code that you don't intend to use it, while still being more descriptive than just a plain `_`. If we decided to use that argument, we would change it to `params`.

We then use the `render` function and feed it two arguments: the connection and then a string, `"index.html"`. The string indicates where we'll get the template to display our page. This is partly Phoenix Magic; through naming conventions it knows that `index.html` in the `PageController` means `lib/star_tracker_web/templates/page/index.html.eex`, and it also knows to use the `StarTrackerWeb.PageView` as the View (we'll cover Views later).

Our New Controller Function

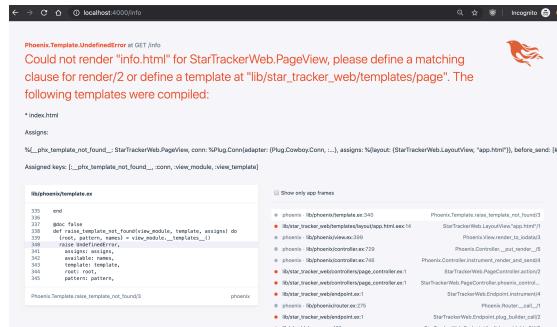
Our controller function won't be too much different.

```
defmodule StarTrackerWeb.PageController do
  use StarTrackerWeb, :controller

  def index(conn, _params) do
    render conn, "index.html"
  end

  def info(conn, _params) do
    render conn, "info.html"
  end
end
```

As you can see, the only differences are the name and the location of the template file. This is enough to give us a new error message when we try to visit `/info` in the browser.



The error is “Could not render”info.html” for StarTrackerWeb.PageView, please define a matching clause for render/2 or define a template at “lib/star_tracker_web/templates/page””. It’s pretty clear what we need to do: define a template.

The Template



We can do better than that though- we can put words on the page!

```
<h1>Hello!</h1>

<p>We're making this app for the following reasons:</p>

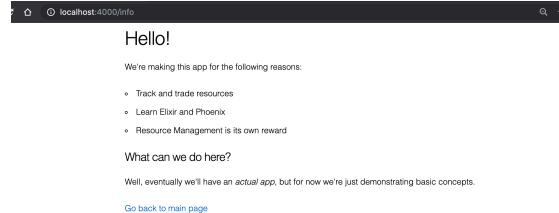
<ul>
  <li>Track and trade resources</li>
  <li>Learn Elixir and Phoenix</li>
  <li>Resource Management is its own reward</li>
</ul>

<h3>What can we do here?</h3>

<p>Well, eventually we'll have an <em>actual app</em>, but for now we're

<a href="/"><div class="btn btn-primary">Go back to main page</div></a>
```

This is all plain html, but it gets the job of filling out our page done.



Previously On: HTML

We're introducing some new HTML elements/tags here.

First is the ul/li combo. “ul” stands for “unordered list”, and “li” stands for “list item”. You’ll see several “li”’s nested within one “ul”. The default styling is a bullet pointed list, although you can change that.

“h3” is like “h2”, but smaller. “h1” through “h6” are available, with “h1” being the biggest and “h6” the smallest.

“em” stands for “emphasis”. By default this does the same thing as italics (*<i>*).

“a” stands for “anchor”. It’s a link. We won’t often use the bare “a” tag in Phoenix – we’ll prefer the `link` helper we’ll introduce in chapter 2.6 – but this html is good enough to get our page working.

And with that, we have our page!

Exercises

1. Bring your local app up to where we are.
2. There are many places where naming is important to Phoenix, but other places where it isn’t. Try changing naming in the following two places:
 - a) The Controller function name: try changing `:info` to `:about` in the `get` helper in the Router, and then change the function name in the Controller to `about` as well.
 - b) The template name: change `info.html` to `information.html` in the Controller function, then change the template filename to match (`web/templates/page/information.html.eex`).

You’ll know you’ve succeeded once you’ve made the changes and the page at `/info` still works as before. If you want to check that it’s actually doing some-

thing and not just coasting off an old version of the app, feel free to check halfway through each change- when you've changed one of the parts but not the other. You should, at that point, see an error message.

Conclusion

In this chapter we've create a new page at the url `/info`. To do this we had to create a new Route in the Router, a new function in the Controller, and a new template. Although we'll expand on it later, Route -> Controller -> Template is the basic path that all requests take when being served by Phoenix.

In the next chapter, we'll explore further the connections between the Router, the Controller, and the Template.

Curious where all of those handy macros come from, like `get` or `pipeline`? Check out the appendix section where we explain `use`, `import`, and `alias`.

Passing Data

Working with and responding to dynamic data is one of the core responsibilities of a Phoenix app. Without that, it would be just a very complex way of writing static HTML!

While dynamic data will eventually come from your users, your database, and other apps, for this chapter we're going to focus on the initial task of passing data in between your Controller, Template, and the URL.

Controller to Template

We'll start with the end in mind: displaying dynamic data to your user.

```
<h1>Hello <%= @name %>!</h1>

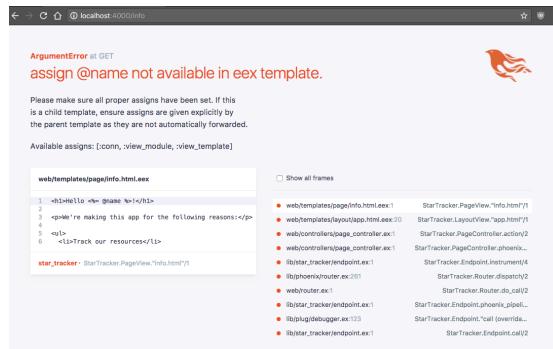
<p>We're making this app for the following reasons:</p>

<!-- ... -->
```

Here's the updated start of our info page. We've added `<%= @name %>` as part of our heading.

The `<%= %>` construct is how we embed Elixir into our HTML. Anything inside there is Elixir code. In this case we're displaying the variable `@name`.

Of course, that variable doesn't exist yet, so we'll get the following error:



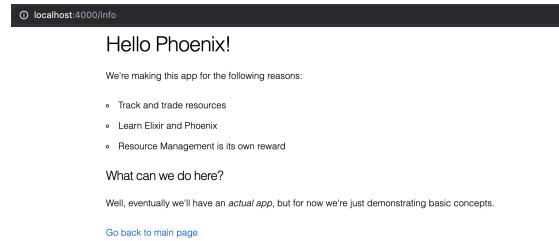
The error says “assign `@name` not available in eex template”, so let's make it available!

We'll do so in the Controller, by adding a Map to the end of our `render` function.

```
def info(conn, _params) do
  render(conn, "info.html", name: "Phoenix")
end
```

We have a map with one key-value pair- `:name` and “Phoenix”. This will be enough to let our page display.

Note: Maps can be used as arguments without the `%{}` wrapping them. The line is equivalent to `render(conn, "info.html", %{name: "Phoenix"})`



```
localhost:4000/info
Hello Phoenix!
We're making this app for the following reasons:


- Track and trade resources
- Learn Elixir and Phoenix
- Resource Management is its own reward


What can we do here?
Well, eventually we'll have an actual app, but for now we're just demonstrating basic concepts.
Go back to main page
```

And that's how we pass data from Controller to Template!

Router to Controller

The Router takes the URL and uses that to choose a Controller and a Controller action. However, you can also make it capture more information from the URL and then pass it along to the Controller action.

Our current Router looks like this (for brevity's sake, we're only showing the relevant scope):

```
scope "/", StarTracker do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
  get "/info", PageController, :info
end
```

We can choose to add a parameter to the `info` route. Let's call that parameter ‘`:name`’:

```
scope "/", StarTracker do
  pipe_through :browser # Use the default browser stack

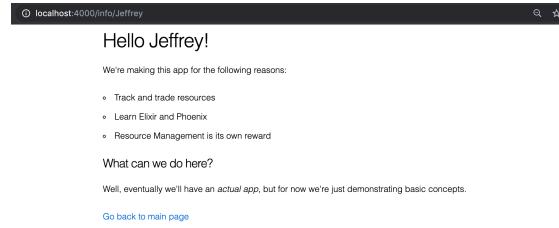
  get "/", PageController, :index
  get "/info/:name", PageController, :info
end
```

So our new route is `/info/:name`. `:name` is an atom instead of a plain String, marking it as a parameter. You can type anything in there and it will map to the `:name` parameter- for example, `/info/Jeffrey`.

Let's hook it up in the Controller.

```
def info(conn, params) do
  render conn, "info.html", name: params["name"]
end
```

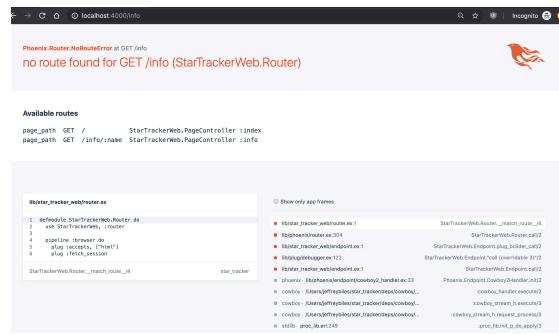
We're now using the `params` argument (and taking away the leading underscore). The `params` is a Map of all the URL parameters- in this case, just "name". So you call `params["name"]` to get the information from your URL, and feed it to the `name` key in the Map you feed to `render`.



We're now using the Router to pass data from the URL to the Controller, and then on down to the Template.

Default Parameters

However, there's a problem: if we go back to our plain `/info` route (without a parameter), we'll run into an error.



That can be solved by having two routes defined in our router:

```
scope "/", StarTracker do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
  get "/info", PageController, :info
  get "/info/:name", PageController, :info
end
```

So now we can get the `/info` route working both with and without parameters... but it doesn't look quite right.

```
localhost:4000/info
Hello !
We're making this app for the following reasons:


- Track and trade resources
- Learn Elixir and Phoenix
- Resource Management is its own reward


What can we do here?
Well, eventually we'll have an actual app, but for now we're just demonstrating basic concepts.
Go back to main page
```

The extra space may seem like a small thing, but it will look like a bug to a user.

There are several ways we could handle this. The first is to use a default name in the absence of a name passed in the URL.

```
def info(conn, params) do
  name = params["name"] || "Stranger"
  render conn, "info.html", name: name
end
```

Then, on the `/info` route, it will tell display "Hello Stranger", and on the `/info/Jeffrey` route it will display "Hello Jeffrey".

```
localhost:4000/info
Hello Stranger!
We're making this app for the following reasons:

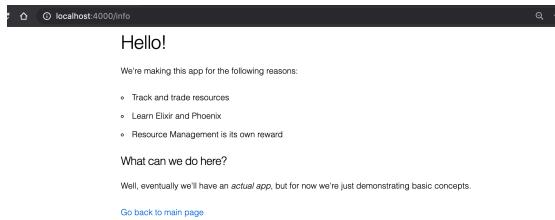

- Track and trade resources
- Learn Elixir and Phoenix
- Resource Management is its own reward


What can we do here?
Well, eventually we'll have an actual app, but for now we're just demonstrating basic concepts.
Go back to main page
```

If we don't want to come up with a default name we can take measures to remove the space before the exclamation point. The simplest in this case is to use an `if` in the template.

Template Conditionals

```
<%= if @name do %>
  <h1>Hello <%= @name %>!</h1>
<% else %>
  <h1>Hello!</h1>
<% end %>
```



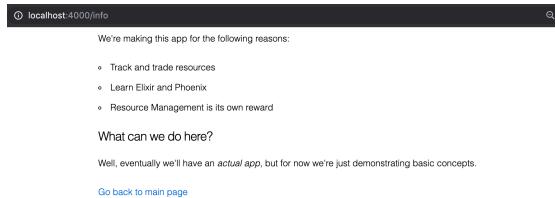
I know I've ragged on `ifs`, but Embedded Elixir plays by slightly different Best Practice rules than regular Elixir. It's still best to limit the amount of if-branching (especially nested ifs), but the strictures are a bit looser.

An `if` statement in the template starts off with this line: `<%= if @name do %>`. The conditional between the `if` and the `do` will vary, but everything else will always be part of the `if` statement. The block of html immediately after the `if` is what is displayed if the conditional (`@name` in this case) is “true”. I say “true” in quotes because it will be “true” for anything except `nil` or `false`- the conditional `@name` could be the boolean `true`, a number, a Map, a List, or (as is the case here) a String.

There's an optional `<% else %>` statement. If this is present, then the block of html immediately after it will be displayed when the conditional is `false` or `nil`.

Finally, it's closed out by `<% end %>`.

You may have noticed that the starter `<%= if @name do %>` has an equals sign at the start, while `<% else %>` and `<% end %>` don't. That's because the equals sign generally signifies that something will be displayed to the user (I guess the `else` doesn't get it because the equals sign in front of `if` already takes care of it- honestly not sure). Try removing the equals sign and you'll see that the entire header disappears.



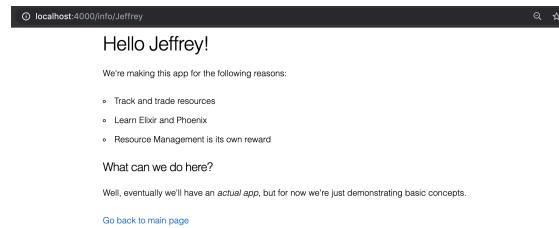
Conclusion

We've now seen how data is passed from the Route URL to the Controller to the Template. In the next chapter, we'll learn about links (one of the most important building blocks in a web page) and each loops (a common way of displaying

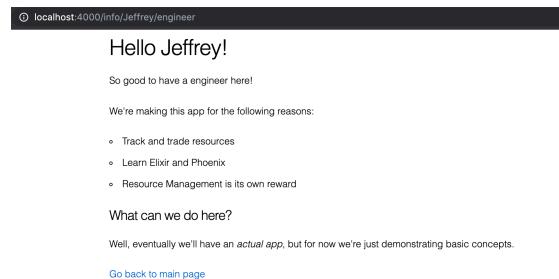
custom data arrays), and how to combine them to let the user customize the URL without ever touching the address bar.

Exercises

1. Bring your app up to speed with the code we wrote in the chapter.
2. Make it so the name is always capitalized, no matter how it's written in the URL



3. Add a second parameter to the URL- :position. After the header add a line welcoming them- if their position was "Engineer", it would say "So good to have an Engineer here".



We won't make you cover all 4 possible cases of names and positions being specified or not specified- multiple optional parameters are not a pattern we'll want to use in our Phoenix apps.

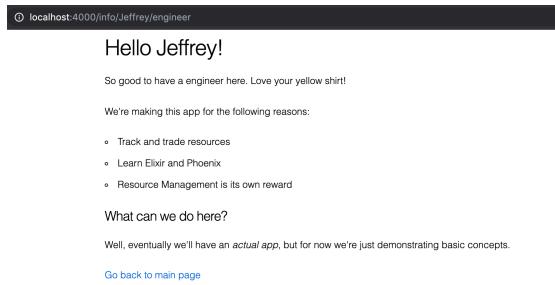
4. Add a shirt color based on their position. Here's a handy chart:

- commander -> red
- operations -> yellow
- engineer -> yellow
- security -> yellow
- scientist -> blue
- doctor -> blue
- other -> grey

Note: This is only true for the recent fleet operations- red/yellow used to be switched. The security personnel, operations, and engineering all wore red shirts, mostly to hide the security personnel's

blood. Once our interactions with alien planets became more one-sided, the majority of security personnel were eliminated (by not hiring them! As opposed to how they used to be eliminated), and command took on the symbolic risk of the red uniform.

So if they're an Engineer, then it should say "So good to have an engineer here. Love your yellow shirt!"



Choosing the URL

In the last chapter we saw how changing the URL could effect what's shown in the page. However, we were stuck editing the URL by hand when using the site-something that's rarely done. In this chapter we'll learn about the link helper-and then about how to use arrays and `each` to automatically generate a bunch of links.

Links

Previously, we'd used a link with a bare "anchor" tag as follows:

```
<a href="/"><div class="btn btn-primary">Go back to main page</div></a>
```

We can add more of these to link to various versions of this page:

```
<a href="/info/José">José</a>
<a href="/info/Chris">Chris</a>
<a href="/info/Jeffrey">Jeffrey</a>
<hr>
```

At this point, you have a list of 3 names, clicking any of which takes you to a different version of the info page.

```
① localhost:4000/info/José
```

Hello José!

José Chris Jeffrey

This is pretty good right now, but if we had more names we wanted to include, a dynamic list of names, or a more complex definition for each link, then things would get unwieldy fast. That's why we're introducing the `for` helper.

The `for` helper

The `for` helper is a way of iterating over data within a template, letting you create more dynamic templates.

Here's a simple version:

```
<%= for name <- ["José", "Chris", "Jeffrey"] do %>
  <%= name %>
<% end %>
```

First, notice the similarities to the `if` helper.

```
<%= if (xs) do %>
  Stuff
<% end %>

<%= for (x <- xs) do %>
  Stuff
<% end %>
```

So they start off with the equals version of the Embedded Elixir tag (`<%= %>`), then the name of the helper (`if` or `for`), then there's some stuff in parentheses (the parentheses were added to help you see the similarities- we don't actually recommend using those in your application, as they simply clutter things up), then `do`. That starts off a block of html which is used by the helper (displayed 0 or 1 times by the `if` helper, depending on the condition, and displayed n times by the `for` helper, where n is the number of items in the array), which is closed by `<% end %>`.

The interesting differentiator is in what comes between the helper name and `do`. Let's look at our `for` helper again.

```
<%= for name <- ["José", "Chris", "Jeffrey"] do %>
  <%= name %>
<% end %>
```

On the right side of the backwards arrow (`<-`) we have an array: `["José", "Chris", "Jeffrey"]` Then on the left side of the backwards arrow we see the variable (`name`) that the items in the array will be assigned to. Within the block we can use that variable (`<%= name %>`).

And with that we'll be looping over all 3 names and displaying them dynamically.

localhost:4000/info/José

Hello José!

José Chris Jeffrey

You'll notice that there are no longer links. Let's fix that.

```
<%= for name <- ["José", "Chris", "Jeffrey"] do %>
  <a href="/info/<%= name %>"><%= name %></a>
<% end %>
```

Here we've brought back our anchor tag, but we've dynamically inserted the `name` variable inside of it; once for the url and once for the displayed text.

Benefit: easily change how data is displayed

We'll see more benefits to this arrangement once we start pulling data from databases and external sources, but we can already see one benefit: changing the way the data is displayed is much easier. Let's put them in a list instead of displayed horizontally.

```
<ul>
  <%= for name <- ["José", "Chris", "Jeffrey"] do %>
    <li><a href="/info/<%= name %>"><%= name %></a></li>
  <% end %>
</ul>
```



We just had to add the `ul` tag around the `for` helper, and then the `li` tag around the anchor tag inside the block. Without the `for` helper, we would have to add the `li` tag around each anchor tag individually- a bad situation if we had lots of items in the list or we wanted to do something more complex than just adding one tag.

From the Controller

We don't have to define the list within the template. In fact, most of the time the data in the template will be fed into it from the controller. Let's do that with our list of names.

```
def info(conn, params) do
  names = ["José", "Chris", "Jeffrey"]
  render conn, "info.html", name: params["name"], names: names
end

<ul>
  <%= for (name <- @names) do %>
    <li><a href="/info/<%= name %>"><%= name %></a></li>
  <% end %>
</ul>
```

Within the controller we're now feeding the template two variables: `name` and `names`. They're usable in the template as `@name` and `@names`.

Within the template we then use `@names` to feed the `for` helper.

Name Collisions

You'll notice that `name`, `@name`, and `@names` are pretty similar. `name` comes from the `for` helper, and can only be used inside of it, while `@name` and `@names` come from the controller. I happen to think these are the best variable names in this particular situation, but it's useful to keep in mind that such similar variable names could cause confusion.

The link helper

We don't have to construct anchor tags by hand every time. Phoenix has a handy `link` helper.

```
<%= link name, to: "/info/#{name}" %>
```

The above is equivalent to

```
<a href="/info/<%= name %>"><%= name %></a>
```

So it's a little shorter. There are other benefits to it that we'll see later on, such as being able to choose a different HTTP function than "get", but for now it's just shorter and more "Phoenix-y".

So let's analyze each part of it.

```
<%= link name, to: "/info/#{name}" %>
```

First, the structure of the helper. It doesn't have a `do` at the end, so this helper is self-contained rather than having an html block and an `<% end %>`. We also know, because of the equals sign at the start, that it will display something. `link`, of course, tells us which helper we're using.

The first argument is what will be displayed - in this case, the value in the `name` variable. The second argument is a Map containing one key-value pair. The key is `to`, and the value is the URL that we're linking to. You'll notice that we use Elixir-style string interpolation (`"#{name}"`) rather than Embedded-Elixir-style string interpolation (`<%= name %>`). That's because this is already inside an Elixir block (the `link` helper).

There are other options available within the hash: `method`, which changes the HTTP method used, and other keys are passed directly to the html (class, data, etc.)

It's worth noting that there is second form of the `link` helper that does have a `do` block (for more complex link text). There's also other ways to define the url in the `to` option. We'll cover those when we need them.

Conclusion

Both the `link` and `for` helpers will be seen often throughout a typical Phoenix app. In addition to their uses as such, they've also served as good examples of how template helpers work in Phoenix- you can generalize based on those examples to the point where any Phoenix template you look at will be considerably less mysterious.

We're done covering the building blocks that we can comfortably tackle in an isolated page. In the next chapter we'll use Phoenix generators to generate some pages and finally start working with the "full" Phoenix stack, including the database.

Exercises

1. Update your app to where we are. Save and commit to git.
2. Use the `for` helper to dynamically display the 3 reasons we have listed for creating this app ("Track our resources", "Learn Elixir and Phoenix", and "Inventory Management is its own reward"). Pass them in from the controller as `@reasons`.
3. Start with your code from the chapter 2.5 exercises where you added a position variable. Then use nested `for` helpers (one `for` helper inside another) to create 3 links for each name ("Chris the commander", "Chris the engineer", "Chris the scientist"), with each linking to the page with the name and position shown.

localhost:4000/info/Jeffrey/scientist

Hello Jeffrey!

- José the commander
- José the engineer
- José the scientist
- Chris the commander
- Chris the engineer
- Chris the scientist
- Jeffrey the commander
- Jeffrey the engineer
- Jeffrey the scientist

So good to have a scientist here. Love your blue shirt!

Generating Resources

So far we've been introducing concepts piece by piece, building up a slowly more complex page. However, the interconnected nature of Phoenix means that if we kept on doing that, we'd spend quite a bit of time with incomplete or broken features. Instead of doing that, we're going to jump ahead and use a Phoenix generator to create a complete resource, then go over customizing it piece by piece.

For this chapter, we're going to generate the resource, input several instances of it into the database, then look at the various ways we can interact with the default generated pages (without looking at the underlying code of those pages). In the later chapters of the section we'll look at the code that's been generated, understand how it works, and see what will happen when we change it.

The generate task

We'll be doing the code generation by running a mix task from the command line- specifically `mix phx.gen.html`. `mix` is Elixir's task runner (more on it later), and `phx.gen.html` is the name of the task we're running. We're going to be adding a few more options to this task, giving us the complete command below:

```
mix phx.gen.html Trade Material materials name:string amount:integer
```

The first argument, `Trade`, is the *context*. They're a useful organizational tool which we'll say more about later; in this tutorial, we'll have two contexts: `Trade` and `Auth`.

The next two arguments, `Material` and `materials`, are the singular and plural forms of the name of the resource we're generating. Phoenix chose to be explicit about this rather than trying to guess the plural, like in some other popular frameworks, because being explicit in this case results in less confusion. This is an outgrowth of Phoenix's general philosophy: abstract away what you can, but when an abstraction could cause confusion, prefer being explicit over obtaining that last inch of magical convenience.

So we have the capitalized singular and the lower-case plural. The other two options are both fields for our resource. The part before the colon is the name of the field, while the part after the colon is the type. In the instance of `name:string`, the field is called `name`, while it will be stored in the database as a `string`. `string` will also determine part of the code generated in the templates.

Run the task.

```
$ mix phx.gen.html Trade Material materials name:string amount:integer
* creating lib/star_tracker_web/controllers/material_controller.ex
* creating lib/star_tracker_web/templates/material/edit.html.eex
```

```
* creating lib/star_tracker_web/templates/material/form.html.eex
* creating lib/star_tracker_web/templates/material/index.html.eex
* creating lib/star_tracker_web/templates/material/new.html.eex
* creating lib/star_tracker_web/templates/material/show.html.eex
* creating lib/star_tracker_web/views/material_view.ex
* creating test/star_tracker_web/controllers/material_controller_test.exs
* creating lib/star_tracker/trade/material.ex
* creating priv/repo/migrations/20190506123826_create_materials.exs
* creating lib/star_tracker/trade.ex
* injecting lib/star_tracker/trade.ex
* creating test/star_tracker/trade_test.exs
* injecting test/star_tracker/trade_test.exs
```

Add the resource to your browser scope in lib/star_tracker_web/router.ex:

```
resources "/materials", MaterialController
```

Remember to update your repository by running migrations:

```
$ mix ecto.migrate
```

Then follow the instructions given by the mix command. First we'll add the resources line to web/router.ex:

```
scope "/", StarTracker do
  pipe_through :browser # Use the default browser stack

  resources "/materials", MaterialController

  get "/", PageController, :index
  get "/info", PageController, :info
  #...
end
```

This adds several available urls to our application, which we'll be visiting later in this chapter.

Then we'll run the ecto migration:

```
$ mix ecto.migrate
Compiling 17 files (.ex)
Generated star_tracker app
```

```
[info] == Running 20190506124749 StarTracker.Repo.Migrations.CreateMaterials.change/0 forward
[info] create table materials
[info] == Migrated 20190506124749 in 0.0s
```

This changes the database, creating the table `materials`- which is where we'll store our materials data.

Adding materials

Run `mix phx.server` to get everything running again.

```
$ mix phx.server
[info] Running StarTrackerWeb.Endpoint with cowboy 2.6.3 at 0.0.0.0:4000 (http)
[info] Access StarTrackerWeb.Endpoint at http://localhost:4000
```

Webpack is watching the files...

Then visit `localhost:4000/materials`.

localhost:4000/materials

Listing Materials

Name	Amount
New Material	

We've got the “Listing Materials” header, an empty table with two columns (matching the two fields we put in the generator), and a “New material” button.

Let's click the “New material” button.

localhost:4000/materials/new

New Material

Name

Amount

SAVE

[Back](#)

The route we're at now is `/materials/new`. Here we've got a form with two fields – once again, corresponding to what we put in the generator.

Let's fill them out with “Hydrogen” for the name and 2000 as the amount.

New Material

Name
Hydrogen

Amount
2000

SAVE

[Back](#)

The name is self-explanatory, and in a move that will anger scientists the world over, we're not attaching any units to the amount. Sorry. Think of it as two thousand moles of Hydrogen, if that helps you cope.

There are 6.022×10^{23} atoms in a mole. One mole of Hydrogen is 1.008 grams. One mole of Helium is 4.003 grams, and one mole of Carbon is 12.011 grams. You don't need to remember any of that, since we're not actually attaching units in the app, but this fancy Chemical Physics degree needs to pay its way somehow.

Click the "Submit" button and you'll find yourself at the route `/materials/1`. It gives you a nice "Material created successfully" message, then shows the material you just created.

Material created successfully.

Show Material

- Name: Hydrogen
- Amount: 2000

[Edit](#) [Back](#)

We have two buttons at the bottom: `edit` and `back`.

For now, let's hit the `back` button, which will take us to the materials index page at `/materials`.

Listing Materials

Name	Amount
Hydrogen	2000

[New Material](#)

Now the table isn't empty- it's filled with the item you just created.

Let's create more.

Name	Amount	
Hydrogen	2000	Show Edit Delete
Carbon	500	Show Edit Delete
Helium	200	Show Edit Delete
Oxygen	800	Show Edit Delete

[New Material](#)

Now that we've got our table filled out, let's check out those buttons to the right.

Exploring the default actions

`delete` does exactly what you'd expect- it deletes a material.

`show` takes you back `/materials/1`, which is sometimes colloquially called the "show page", because it shows us one item.

○ Name: Hydrogen
○ Amount: 2000

[Edit](#) [Back](#)

Show Material

- **Name:** Hydrogen
- **Amount:** 2000

[Edit](#) [Back](#)

If you visit the `show` page of another material you'll get a URL with a different number. That number is the `id` of the material in the database- a unique identifier and therefore a great way to determine which resource we're talking about (unlike the name, which can change).

The `id` is automatically assigned. In this case it's "1" because this is the first material we created and our current `id`-assignment scheme is sequential starting from 1.

The contents of the `show` page is the displayed data for the resource. Currently this is exactly what's in the table on our index page, but often this includes more data than the table or a more user-friendly (but space-consuming) display of said data.

Below that we have two links we can click: `Edit` and `Back`. `Back` will take us back to our index page, so let's click the more interesting `Edit` link.

The screenshot shows a web browser window with the URL `localhost:4000/materials/1/edit`. The page title is "Edit Material". There are two input fields: "Name" containing "Hydrogen" and "Amount" containing "2000". Below the inputs is a blue "SAVE" button. At the bottom left is a "Back" link.

The url here is `materials/1/edit`. The `1` part is the id, determined the same way as in the `show` page.

You'll notice that the contents of this page are almost exactly like our `new` form, except that the data fields come pre-filled (with the current data) and it says "Edit" up top instead of "New". We'll see why that is later when we look at how these pages are constructed.

Go ahead and change the update the `amount` to 3000 and click `Submit`.

The screenshot shows a web browser window with the URL `localhost:4000/materials/1`. A blue banner at the top says "Material updated successfully.". Below it, the page title is "Show Material". It lists the material details: "Name: Hydrogen" and "Amount: 3000". At the bottom are "Edit" and "Back" links.

It automatically redirects us to the `show` page, with the `amount` changed and a nice "Material updated successfully" message.

Click the `Back` button to take us back to the `index` page. We'll see our table as before, but with the updated amount of Hydrogen.

Conclusions

That's how we generate and interact with a basic resource. We'll be going over that piece by piece over the next several chapter, showing how each part works and how we can customize it. We'll start by looking at what goes into the `index` route, tracing what happens starting from `web/router.ex`.

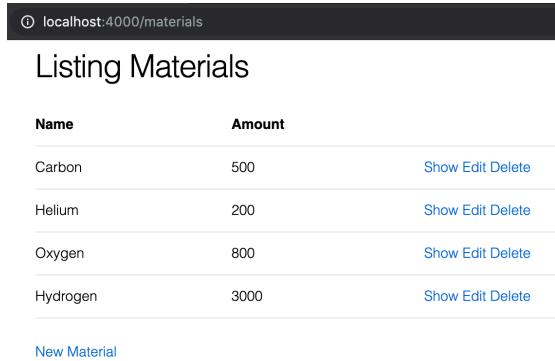
Exercises

1. Generate the materials resource as shown in the chapter. Use the same fields.
2. Input several materials into the database.
3. Edit one of the materials.
4. Delete one of the materials.

The Material Index Page

In the last chapter we auto-generated the Material resource, added a couple types of materials, then checked out the various pages that were auto-generated for us.

This chapter we're going to focus on only one of those pages – the index page at `/materials` – and dig in to the code that backs it. Then we'll use our knowledge to customize the page and make it our own.



Name	Amount	
Carbon	500	Show Edit Delete
Helium	200	Show Edit Delete
Oxygen	800	Show Edit Delete
Hydrogen	3000	Show Edit Delete

New Material

We're going to start at the route and work our way down to the template.

RESTful Routes

To make our generate resource work, we had to add one line to the `web/router.ex` file:

```
scope "/", StarTrackerWeb do
  pipe_through :browser # Use the default browser stack

  resources "/materials", MaterialController
  ...
end
```

This one line added several routes for us, and is the equivalent of the following seven lines:

```
scope "/", StarTrackerWeb do
  pipe_through :browser # Use the default browser stack

  get "/materials", MaterialController, :index
  get "/materials/:id", MaterialController, :show
  get "/materials/:id/edit", MaterialController, :edit
  get "/materials/new", MaterialController, :new
```

```

post "/materials", MaterialController, :create
patch "/materials/:id", MaterialController, :update
delete "/materials/:id", MaterialController, :delete

#...
end

```

Those seven routes are what make up a “RESTful” resource. The four `get` routes are the various pages that we visited while playing with our app in the last chapter, and the three other routes (create, update, and delete) are the three actions we could take to change the records.

We’ll cover the other six routes later; for now what we care about is that the line `resources "/materials", MaterialController` ends up giving us (among other things) `get "/materials", MaterialController, :index`, which is where we’ll find our Index page.

Index Controller Action

We’ve mapped out the url `/materials` to the `index` action on `MaterialController`. Let’s look at that action.

```

defmodule StarTrackerWeb.MaterialController do
  use StarTrackerWeb, :controller

  alias StarTracker.Trade
  alias StarTracker.Trade.Material

  def index(conn, _params) do
    materials = Trade.list_materials()
    render(conn, "index.html", materials: materials)
  end

  #...
end

```

A quick review of controllers and all the code that comes before the index.

First, we define our module: `StarTrackerWeb.MaterialController`. That name is connected with the filename, and with how it connects with views and templates, so don’t mess with it. Next we load in a bunch of controller-related code with `use StarTrackerWeb, :controller`. Our final piece of setup is our aliases. `alias StarTracker.Trade` means we can use `Trade` in the rest of the file without including `StarTracker` in front of it. In the same way, `alias StarTracker.Trade.Material` means we can use `Material` without including `StarTracker.Trade` in front of it.

Now we can get to the `index` action itself.

At the bottom we have our familiar `render` function, which takes the connection (`conn`), the name of the template ("index.html"), and an optional Map (`materials: materials`). That part all works like before. The new part is how we get our data.

Instead of hard-coding a list like we did in Chapter 2.6, we're bringing in data from the database. We call `Trade.list_materials()`, which we've defined in `Trade` context.

The Trade context

In `lib/star_tracker/trade.ex` we define the `Trade` context.

```
defmodule StarTracker.Trade do
  @moduledoc """
  The Trade context.
  """

  import Ecto.Query, warn: false
  alias StarTracker.Repo

  alias StarTracker.Trade.Material

  def list_materials do
    Repo.all(Material)
  end

  # five other methods, abbreviated
  def get_material!(id) #...
  def create_material(attrs \\ %{}) #...
  def update_material(%Material{} = material, attrs) #...
  def delete_material(%Material{} = material) #...
  def change_material(%Material{} = material) #...
end
```

This is where we store our interactions with the database, and while exploring the generated code we'll go over each of the six generated functions in this file.

For now, we'll just look at the `list_materials` function.

It's one line: `Repo.all(Material)`. We'll get into the database and the various ways you can interact with it later, but for this particular case the naming is really descriptive- we're going to the `Repo` (database) and getting all of the Materials.

** Captain's Log: Why the Trade context?

At first, the Trade context seemed unnecessary. Many of the items are just one line, and only used once, so why not put that code directly in the controller?

However, when my database interactions grew in complexity, I saw the value in having separate functions.

When I started doing stuff beyond the basic CRUD operations, I saw how the reusability was useful.

Finally, when my app became bigger and my crew became more involved, I saw how having all the database interactions in one place let me know where to look for code that effected persistence.

The Material Model

Let's look at what each Material consists of. We've seen this in action, of course, when we were creating and viewing our data, but now we're going to look at how the Material model is defined.

```
defmodule StarTracker.Trade.Material do
  use Ecto.Schema
  import Ecto.Changeset

  schema "materials" do
    field :amount, :integer
    field :name, :string

    timestamps()
  end

  @doc false
  def changeset(struct, params \\ %{}) do
    struct
    |> cast(params, [:name, :amount])
    |> validate_required([:name, :amount])
  end
end
```

We first have the conventional module naming (don't change it!), and then the `use` statement that pulls in a lot of model-related functions and macros. Some of those macros are `schema`, `field`, and `timestamps`.

After that we import `Ecto.Changeset`, which gives us `cast` and `validate_required`, among others.

The `schema` block is where we can view the structure of our model. Within the block you can define various fields.

The first three fields we have here are defined by giving an atom for the name of the field and an atom for the type of the field (`:name` and `:string`, or `:amount` and `:integer`). These were created by the options we gave the generator.

The next thing we see is `timestamps()`. This creates `createdAt` and `updatedAt` fields for you, which are really useful and are thus included on every generated model unless you remove the `timestamps()` line. These two datetime fields are not set by the user, but are set by the database itself (based on the current time) when a record is created or updated.

There are other options and ways of defining fields, which we'll go over later when we run into them.

For now we're going to ignore the `changeset` function. We're also going to ignore the `migrations` file. We'll go over both of those in future chapters. For now, we just wanted some context for the `index` action.

The Index Template

Finally, we get to the template file itself, in `/lib/star_tracker_web/templates/material/index.html.eex`. It's long, but you should already recognize quite a bit of what's going on.

```
<h1>Listing Materials</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Amount</th>

      <th></th>
    </tr>
  </thead>
  <tbody>
    <%= for material <- @materials do %>
    <tr>
      <td><%= material.name %></td>
      <td><%= material.amount %></td>

      <td>
        <%= link "Show", to: Routes.material_path(@conn, :show, material) %>
        <%= link "Edit", to: Routes.material_path(@conn, :edit, material) %>
        <%= link "Delete", to: Routes.material_path(@conn, :delete, material), method: :de
      </td>
    </tr>
    <% end %>
  </tbody>
```

```
</table>

<span><%= link "New Material", to: Routes.material_path(@conn, :new) %></span>
```

We start off with our `h2` header, giving the user an idea of what the page is about.

The rest of the page is a big html table.

A table consists of a table header (`thead`) and a table body (`tbody`). The table body is made up of rows and columns of data. The table header is where we put the titles of each column.

```
<thead>
  <tr>
    <th>Name</th>
    <th>Amount</th>

    <th></th>
  </tr>
</thead>
```

Within the `thead` element we have one table row (`tr`), and in that row we have three `th` elements- the first two holding titles of the columns and the last one being a blank placeholder.

Then we have the body of the table:

```
<tbody>
  <%= for material <- @materials do %>
    <tr>
      <td><%= material.name %></td>
      <td><%= material.amount %></td>

      <td>
        <!-- links go here -->
      </td>
    </tr>
  <% end %>
</tbody>
```

We've removed some of the links so we can focus on the structure, and better see its similarities with the table header.

Right within our `tbody` tag we see a `for` block which goes over every item in `@materials`. This means that whatever is within the `for` block will be repeated for every material.

So for every material we create a table row (`tr`), and within the table row

we create three table data (`td`) elements. The first two display data from the `material` record that's in this instance of the `for` block. The last one contains several buttons. Let's look at those.

```
<td>
  <%= link "Show", to: Routes.material_path(@conn, :show, material) %>
  <%= link "Edit", to: Routes.material_path(@conn, :edit, material) %>
  <%= link "Delete", to: Routes.material_path(@conn, :delete, material), method: :delete, data: { confirm: "Are you sure?" } %>
</td>
```

These use the same link helper we went over in chapter 2.6, but here they use more options and are considerably more complex.

The Expanded link Helper

Let's analyze our first link helper:

```
<%= link "Show", to: Routes.material_path(@conn, :show, material) %>
```

We start off with the name of the helper (`link`), followed by the text that will show on the page ("Show"). Then things start diverging from how we used the link helper before.

In the `to` option, we're using another helper in order to construct the URL that the link will be linking to.

```
Routes.material_path(@conn, :show, material)
```

I call this the `x_path` helper to signify that it's not just materials that it works with. If we had an `Officer` resource, then we would have an `officer_path` helper available.

To see the helpers available, and which routes they work with, run `mix phx.routes` in your command line.

\$ mix phx.routes			
material_path	GET	/materials	StarTrackerWeb.MaterialController
material_path	GET	/materials/:id/edit	StarTrackerWeb.MaterialController
material_path	GET	/materials/new	StarTrackerWeb.MaterialController
material_path	GET	/materials/:id	StarTrackerWeb.MaterialController
material_path	POST	/materials	StarTrackerWeb.MaterialController
material_path	PATCH	/materials/:id	StarTrackerWeb.MaterialController
	PUT	/materials/:id	StarTrackerWeb.MaterialController
material_path	DELETE	/materials/:id	StarTrackerWeb.MaterialController
	page_path	GET /	StarTrackerWeb.PageController
	page_path	GET /info	StarTrackerWeb.PageController
	page_path	GET /info/:name	StarTrackerWeb.PageController
	page_path	GET /info/:name/:position	StarTrackerWeb.PageController
websocket	WS	/socket/websocket	StarTrackerWeb.UserSocket

Right now we have `material_path` and `page_path` (and `websocket`), but this list will grow.

So the `x` in `x_path` is by default the name of the controller. The first argument is a connection (`@conn`), the second argument is an atom giving the name of the action on the controller, and then the rest of the arguments are parameters to the route. When the parameter is an id, we can give either the id itself or an object which has an `id` field.

So going back to our `material_path` for `show`:

```
Routes.material_path(@conn, :show, material)
```

We can see that the parameter we give it is the `material` object, from which we'll extract the id.

The `edit` link helper is very similar to the `show` link helper and has no new concepts:

```
<%= link "Edit", to: Routes.material_path(@conn, :edit, material) %>
```

The `delete` link helper does introduce several new things:

```
<%= link "Delete", to: material_path(@conn, :delete, material), method: :delete, data: [confir
```

The first new thing is `method: :delete`. By default, the method is `:get`, and that's what's been in all the links we've gone over so far. `get` is what we use when we want to view a page or get data. However, to make changes to the data we'll need to use `:post`, `:patch`, `:put`, or `:delete`. Our router will recognize the method used and direct us to the correct action.

Then we have `data: [confirm: "Are you sure?"]`. This makes it so that when we click on the button, it will pop up a confirmation box with the text "Are you sure?". If you hit "OK" it will continue on. If you hit "Cancel" it will stop and not perform the deletion. Try it for yourself in the browser!

Data confirmation can be used on a link with any method, but it's most commonly used for links with the `:delete` method.

Finally, we have one last link at the bottom of the page, for `new`.

```
<%= link "New material", to: Routes.material_path(@conn, :new) %>
```

The interesting thing here is that there are no parameters passed, since the `new` action doesn't take any parameters.

Conclusion

This chapter explored how the materials `index` route worked, from top to bottom.

We started with how `web/router.ex` used the `resources` macro to generate the 7 RESTful routes for `materials`, then moved on to deconstructing our controller action, followed by a quick detour into our `Material` model, and finally getting to the index template, which was mostly review except for the `table` elements and new `link` helper options.

In the next chapter we're going to do the same for the `show` route. It should be a shorter exercise, since it shares many things in common with the `index` route, but we'll still learn several new things.

Exercises

1. Add a column to the table: `updatedAt`. Make sure to add it to the header as well.
2. Add `createdAt` column to the table.
3. Convert `Routes.material_path(@conn, :edit, material)` to the raw string form we used in chapter 2.6. This is not because the raw string is better (`x_path` pattern is preferred), but so you can get practice figuring out what the `x_path` helper translates to. Make sure it works for all the materials, not just one.
4. Take the `to` field from the links on the info page and use `page_path` to define them.

The Material Show Page

In the last chapter we went over the Materials Index page. We started from how the route matches a URL to a controller, then showed how the controller gets data from the database to the template (via a method on the context), took a detour to explore the Material model, then finally went over the Embedded Elixir template.

This chapter will do the same thing for the Material Show page. It will be shorter because the show page shares many things in common with the index page. However, aside from the review, there are still some interesting new techniques to be learned that we'll be using throughout our Phoenix app.

ⓘ localhost:4000/materials/1

Show Material

- **Name:** Hydrogen
- **Amount:** 2000

[Edit Back](#)

The Show Route

We have one line in our `web/router.ex` file to represent all of the Material-related routes:

```
resources "/materials", MaterialController
```

This route macro could expand out to seven different routes, including the show route:

```
get "/materials/:id", MaterialController, :show
```

We're matching the URL to the `show` action on the `MaterialController`, and giving it one param: `:id`.

Pattern Matching on Map

The `show` action is as follows:

```
def show(conn, %{"id" => id}) do
  material = Trade.get_material!(id)
  render(conn, "show.html", material: material)
end
```

The first thing you may notice is how we've defined our params argument; instead of simply putting them all in `params` and pulling them out later, we're pattern-matching using `%{"id" => id}`.

There are two aspects to this. One is simply pulling information out of the `params` Map, and could be replicated with the following code:

```
def show(conn, params) do
  id = params["id"];
  material = Trade.get_material!(id)
  render(conn, "show.html", material: material)
end
```

The other aspect to pattern-matching on a Map is that it will refuse to match unless the data is provided as expected. Let's go back to our `LearningElixir` module and create a new `inspire` function:

```
defmodule LearningElixir do
  def my_map do
    %{
      "name" => "Enterprise",
      "type" => "CodeShip",
      "mission" => "To explore strange new techniques, to seek out new programming languages"
    }
  end

  def inspire(%{"name" => name, "type" => type, "mission" => mission}) do
    "These are the voyages of the #{type} #{name}. Its five-year mission: #{mission}"
  end
end
```

Then we can run the following in the command line:

```
iex()> LearningElixir.my_map |> LearningElixir.inspire
"These are the voyages of the CodeShip Enterprise. Its five-year mission: To explore strang...
```

It automatically mapped the values in `my_map` to the arguments in `inspire`. But what happens if we feed it a map that doesn't have all the required key-value pairs?

```
iex() %{"name" => "Jim", "type" => "CodeShip"} |> LearningElixir.inspire
** (FunctionClauseError) no function clause matching in LearningElixir.inspire/1
```

The following arguments were given to `LearningElixir.inspire/1`:

```
# 1
%{"name" => "Jim", "type" => "CodeShip"}
```

```
iex:10: LearningElixir.inspire/1
```

Pattern-matching on the keys protects the function from running unless the map that's been given to it has all the required information. This can catch a large class of possible errors, and prevents you from having to do null checks on those keys (like you would have to do in Ruby or Javascript).

The problem here: Jim has no mission. Hard to inspire people without a mission!

The careful observer will note that we used Strings to match instead of atoms. If you try to match a URL parameter with atoms, it will give the “no function clause matching” error.

Repo.get! and exclamation point functions

Back to our show action:

```
def show(conn, %{"id" => id}) do
  material = Trade.get_material!(id)
  render(conn, "show.html", material: material)
end
```

The next line is about how we get the selected Material. It calls out to the `Trade` context, so let's look there to see what it does.

```
def get_material!(id), do: Repo.get!(Material, id)
```

First, this is a shortcut for short methods. It's the same as the following:

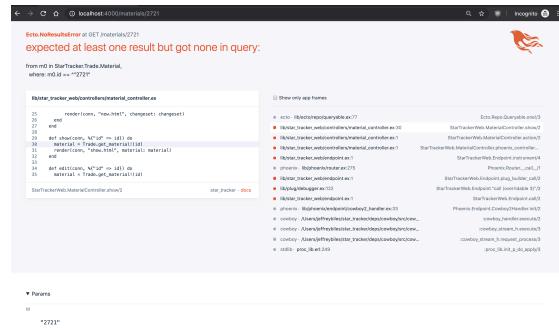
```
def get_material!(id) do
  Repo.get!(Material, id)
end
```

If you have a lot of short lines, then the one-line version can be nice. Now back to the database interaction!

`Repo.get!` takes two required arguments: the record type (`Material`) and the `id` of the record we're trying to get. Remember, `id` is automatically assigned when we create a record and is unique per type of record, so the two pieces of information we're giving `Repo.get!` is always enough information to uniquely identify a record.

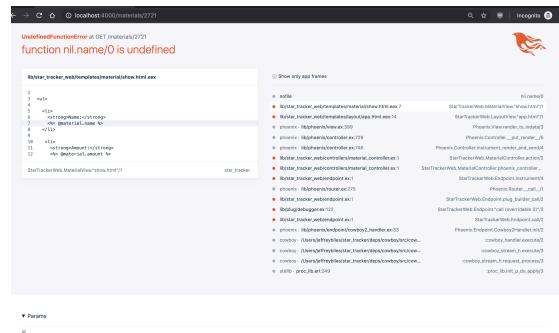
If you look at the list of functions on `Repo`, you may notice something funny... in addition to `Repo.get!` there's also a `Repo.get` function. It has the same name, but without the exclamation point. This is a shorthand adopted from Ruby; an exclamation point tells the user that the function will throw an error if doesn't work instead of failing silently.

Let's test this out by putting in an `id` that we don't have yet:



What a nice error! It points to the exact line with our failing `Trade.get_material!` call and gives us the information that we fed into the function. Plenty of information to reconstruct what went wrong.

Now let's try removing the exclamation point, and see what happens if we feed in a bad ID with `Repo.get`:



What a terrible error. “function nil.name/0 is undefined”. Because our app is still small, and because I have experience with errors like this, I can infer the problem is that `@material` is nil, and trace that back to where `@material` is defined, with `Repo.get`. But that’s a lot of unnecessary work, and if the app is big or you’re a beginner, then it may be overwhelming. The default – `Repo.get!` – is much better.

Anyways, back to our controller. Our last line is very familiar: `render(conn, "show.html", material: material)`. It renders the show template and feeds it the material we grabbed.

The Show template

Here’s the `show` template:

```
<h1>Show Material</h1>
```

```

<ul>
  <li>
    <strong>Name:</strong>
    <%= @material.name %>
  </li>

  <li>
    <strong>Amount:</strong>
    <%= @material.amount %>
  </li>
</ul>

<span><%= link "Edit", to: Routes.material_path(@conn, :edit, @material) %></span>
<span><%= link "Back", to: Routes.material_path(@conn, :index) %></span>

```

Most of this should be review: the header (`h2`), the list (`ul` and `li`), displaying data (`<%= @material.name %>`), and creating links (`<%= link "Edit", to: Routes.material_path(@conn, :edit, @material) %>`).

The only new thing is the `` tag, and it's pretty simple: it bolds the text inside. It is preferred to using the bold (``) tag directly because you can always redefine what the `` tag means; bolding is only a default.

Conclusion

In addition to reviewing the core Phoenix pattern, we saw two important new techniques- exclamation point functions and pattern matching on Map keys. You'll see both of these a lot throughout your Phoenix app and grow to appreciate their usefulness.

Exercises

1. Add the `id` property to the top of the list, above the name.
2. Add the `updatedAt` and `createdAt` properties to the bottom of the list, below the amount.
3. Make a version of the `inspire` function that doesn't require a mission. Not very inspiring, is it? Now make one that requires a mission but doesn't require a spaceship name or type. Test each of them in the command line, making sure that the version with all three keys is triggered when all three keys are available, but the other two versions are triggered when keys are missing.

These are the results you should get:

```
iex()> %{"mission" => "Write awesome code"} |> LearningElixir.inspire  
"Your mission: Write awesome code"  
iex()> %{"name" => "Jeffrey", "type" => "Human"} |> LearningElixir.inspire  
"These are the voyages of the Human Jeffrey."  
iex()> %{"name" => "Jeffrey", "type" => "Human", "mission" => "Make learning programming mor  
"These are the voyages of the Human Jeffrey. Its five-year mission: Make learning programm
```

Hint: the order that you have the function versions in matters.

Deleting a Material

In the last two chapters, we covered the Index and the Show pages. They were both primarily about *displaying* information.

For the rest of this section, we're going to be working with routes and actions where we change information. We may still display information while doing that, but our primary goal is to change things.

To ease our way into these concepts, we're going to start with clicking a link to delete a material from the database.

The Link

The button itself is in the Index page, in the set of three buttons in the final `td` of each row.

```
<%= link "Delete", to: Routes.material_path(@conn, :delete, material), method: :delete, data:
```

We already went over how the options in this particular link works in chapter 3.2, so instead of picking apart the link helper again we'll start investigating where clicking it takes us.

In our router, we've defined `resources "/materials"`, `MaterialController`, which is defining the 7 separate routes that create a RESTful resource. One of those is `delete "/materials/:id"`, `MaterialController`, `:delete`, which is where the link above will take us.

Then that will send us to the `delete` method on the `MaterialController`.

```
defmodule StarTrackerWeb.MaterialController do
  use StarTrackerWeb, :controller

  alias StarTracker.Trade
  # ...
  def delete(conn, %{"id" => id}) do
    material = Trade.get_material!(id)
    {:ok, _material} = Trade.delete_material(material)

    conn
    |> put_flash(:info, "Material deleted successfully.")
    |> redirect(to: Routes.material_path(conn, :index))
  end
end
```

Here we're doing the following steps:

0. Matching the passed-in ID param.
1. Getting the material from the database.

2. Deleting the material from the database.
3. Setting a flash message.
4. Redirecting to the Index.

Getting the `id` is basic pattern-matching of the params Map.

Then the material uses the same function on the `Trade` context as we used to get the material for the show page.

Then after that we start getting into the new stuff.

Deleting from the Database

The next line calls the `delete_material` function on the `Trade` context.

```
{:ok, _material} = Trade.delete_material(material)
```

Let's take a look at that function.

```
def delete_material(%Material{} = material) do
  Repo.delete(material)
end
```

The first thing to notice is that the argument is pattern-matched so it has to be a `Material`. If you try to feed it a string, an integer, or a random Map, then it will throw one of those “no function clause matching” errors.

Next, we call the function `Repo.delete`. It does exactly what it says, and returns a tuple: `{:ok, material}`.

Yes, it returns the material that we just deleted. We destroyed the database record, but that doesn't mean we can't still pass around a Map containing the material's information. However, we won't be using it right now, as indicated by the fact that our actual pattern-matched tuple is `{:ok, _material}`, with an underscore in front of the variable name.

Aside from returning the deleted material (which we just discarded), pattern-matching the tuple also allows us to catch errors better. If there's an error, it will throw at that line when the `:ok` atom isn't matched.

Flash Messages

The next line is

```
conn
|> put_flash(:info, "Material deleted successfully.")
```

Remember, this is the same as:

```
put_flash(conn, :info, "Material deleted successfully.")
```

`put_flash` takes three arguments: the connection, the type of flash message (`:info` and `:error` available by default), and then the message itself.

It returns the `conn` we gave as the first argument, but with an `:info` flash attached.

That will then be used by the following line in `app.html.eex`:

```
<p class="alert alert-info" role="alert"><%= get_flash(@conn, :info) %></p>
```

And will show up like this:

The screenshot shows a browser window with the URL `localhost:4000/materials`. A blue alert box at the top says "Material deleted successfully.". Below it is a table titled "Listing Materials" with three rows: Carbon (500), Helium (200), and Oxygen (800). Each row has "Show", "Edit", and "Delete" links. At the bottom left is a "New Material" link.

Name	Amount	
Carbon	500	Show Edit Delete
Helium	200	Show Edit Delete
Oxygen	800	Show Edit Delete

When you visit any other page (or reload the current page) the flash message will go away.

Redirection

Our final line is a redirect.

```
|> redirect(to: Routes.material_path(conn, :index))
```

It's taking the `conn` returned from the `put_flash` message as the first argument. The second argument is a hash, in this case containing the `to:` option. The `x_path` construct should be familiar from when we were using them in the `link` helper, and it works the exact same way here.

So we go to the `index` function on the `MaterialController` and run through that, getting the materials list and displaying the `index` template.

Why Redirect?

We're already on the Material Index route, so why redirect?

Because every action in a request/response cycle must end with a render or a redirect. If you don't, it will error.

What if you want to delete from multiple screens and have the redirect behavior be different?

Well, I can see two options.

One, which you'll want to choose most of the time, is to have your controllers be small enough that adding a second controller method isn't horrible. This already happens by default for most routes; note that `delete`, at 5 lines of code, is our longest controller method yet.

The other option is to have the route that you're redirecting to be fed in via a parameter. The upside is that you can use the same controller method and route over and over. The downside is that it's breaking the default pattern of where Phoenix wants you to put information, and can therefore confuse later developers.

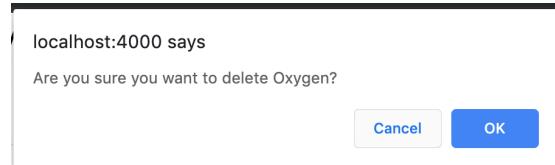
Conclusion

In this chapter we walked through the `delete` function on `MaterialController`. Along the way we learned about the `Repo.delete` function, how returning a tuple can help catch errors, how flash messages work, and how and why we redirect back to the index page.

We'll use much of that knowledge in the next chapter, where we'll walk through the `new` and `create` methods, and show how to use forms and Ecto changesets.

Exercises

1. What happens if you try to redirect to the show page of the material you just deleted? You're feeding in a `%Material` directly, so why does it error?
2. Add two more flash messages: a second `:info` and an `:error`. What happens? Why do you think that is?
3. In the confirmation message before deleting a material, include the name of the element.



Creating a Material

Of all the things that we can do with the generated routes, creating and editing the data are the most complex.

That's why I've saved them for last.

We'll go over creation in this chapter, which will be fairly long. In the next chapter we'll go over editing, which should be fairly short since it mostly goes over the same material. Then, in this section's final chapter, we'll pull together everything we've learned and add a new data field to our Materials.

The Two-Step Process

When creating a resource, we'll be going through two stages: first, we'll be visiting the `new` page, and then, after filling out the information, we'll be hitting the `create` function in order to add the new Material to the database.

Both of these are separate routes, which are defined by the `resources "/materials"`, `MaterialController` we put in our routes file.

If defined separately, they would look like this:

```
get "/materials/new", MaterialController, :new
post "/materials", MaterialController, :create
```

`new` is a `get`, so we'll be getting some data (in this case, an empty `%Material` object) then displaying a page. In that way, it'll work kind of like `show` and `index`.

`create` is a `post`, and it will change the database (adding an item), add a flash message, and then redirect. In that way, it'll work kind of like `delete`.

The `create` function does not have an associated page, but the `new` page will look like this:

The screenshot shows a web browser window with the address bar containing "localhost:4000/materials/new". The main content area is titled "New Material". It contains two input fields: one labeled "Name" and another labeled "Amount", both with empty text boxes. Below these fields is a blue rectangular button with the word "SAVE" in white capital letters. At the bottom left of the form area, there is a small blue link labeled "Back".

The `new` Controller Function

Let's take a look at the `new` function in `MaterialController`.

```
def new(conn, _params) do
  changeset = Trade.change_material(%Material{})
  render(conn, "new.html", changeset: changeset)
end
```

We're once again calling a function on the `Trade` context, but now instead of getting a `Material` we're getting a `changeset`. Let's investigate where that `changeset` comes from.

In the call `Trade.change_material(%Material{})`, we're feeding the `change_material` function the Schema `%Material{}`. We'll explain what exactly that is soon.

```
def change_material(%Material{} = material) do
  Material.changeset(material, %{})
end
```

The `%Material` Schema we fed to `change_material` pattern matches with what the function is expecting, so it's assigned to `material...` and fed into `Material.changeset`. The second argument is a blank Map.

Okay, so we're ultimately getting the `changeset` from `Material.changeset(%Material{}, %{})`. Keep that in mind, but for a second we're going to explore the `new` page's html and see what we're going to do with that `changeset`.

The new Template

`render(conn, "new.html", changeset: changeset)` feeds the `changeset` down to the `lib/star_tracker_web/templates/material/new.html.eex` file, so let's take a look at that.

```
<h1>New Material</h1>

<%= render "form.html", Map.put(assigns, :action, Routes.material_path(@conn, :create)) %>

<span><%= link "Back", to: Routes.material_path(@conn, :index) %></span>
```

The `h1` and `link` should both be familiar to you, but seeing `render` in this context may be new. It works exactly like `render` does in the controller, but it does so as a sub-section of the page. The first argument tells us that the html we'll be rendering will be found at `lib/star_tracker_web/templates/material/info.html.eex`, while the second argument is a Map that contains variables, or assignments.

The assignments Map was `%{changeset: changeset}` when being passed into `new.html.eex`, but with the `Map.put` it's now `%{changeset: changeset, action: Routes.material_path(@conn, :create)}` when being passed into the form.

Let's take a look at that form now:

```
<%= form_for @changeset, @action, fn f -> %>
  <%= if @changeset.action do %>
    <div class="alert alert-danger">
      <p>Oops, something went wrong! Please check the errors below.</p>
    </div>
  <% end %>

  <%= label f, :name %>
  <%= text_input f, :name %>
  <%= error_tag f, :name %>

  <%= label f, :amount %>
  <%= number_input f, :amount %>
  <%= error_tag f, :amount %>

  <div>
    <%= submit "Save" %>
  </div>
<% end %>
```

There's a helper, `form_for`, which takes both `@changeset` and `@action` as the first two arguments.

Now that we've tracked down both where we're using the changeset and where we're getting it from, let's explore what it actually *is*.

Ecto, Schemas, and Changesets

Changesets are one of the four main parts of Ecto.

Ecto is, according to the github repo, “a toolkit for data mapping and language integrated query”. In simpler (but slightly less accurate) terms, it’s a library for handling data and talking to the database.

The four main parts are:

- Ecto.Repo
- Ecto.Schema
- Ecto.Changeset
- Ecto.Query

We've already seen `Ecto.Repo` used before: `Repo.all` and `Repo.get!` are two of the methods we've used to get data from the database.

To understand the form, we'll also need to learn about `Ecto.Schema` and `Ecto.Changeset`.

To see how we're using them, let's take a look at our `Material` file. We'll find it in `lib/star_tracker/trade/material.ex`.

```
defmodule StarTracker.Trade.Material do
  use Ecto.Schema
  import Ecto.Changeset

  schema "materials" do
    field :amount, :integer
    field :name, :string

    timestamps()
  end

  @doc false
  def changeset(material, attrs) do
    material
    |> cast(attrs, [:name, :amount])
    |> validate_required([:name, :amount])
  end
end
```

Right off the bat, we've got `use Ecto.Schema` and `import Ecto.Changeset`.

Then we define our Schema:

```
schema "materials" do
  field :amount, :integer
  field :name, :string

  timestamps()
end
```

This uses a DSL (Domain Specific Language) to easily tell us what the data on the `%Material{}` Schema will look like. With the `name` and `amount` fields defined as above, we could create one like this: `%Material{name: 'Neon', amount: 200}`.

A Schema like `%Material{}` looks a lot like a Map, and that's because it *is* a type of Map, but with some extra features. What's the difference between them? A plain Map will take any key-value combo, while a Schema will *only* take the keys defined in the `schema` DSL.

So at the simplest level, a Schema is like a structured Map. Having the Map be structured can help keep all parts of your app on the same page, preventing many common errors... including the following typo-related example:

```
iex()> map = %{amount: 3}
%{amount: 3}
```

```
iex()> map = Map.put(map, :nme, "Sure hope this saves right")
%{amount: 3, nme: "Sure hope this saves right"
iex()> map.name
nil
```

So that's our Schema.

A Changeset is a tool for tracking changes to the data in Schemas like `%Material{}` before committing them to the database. Our first clue into how this works is where we define the `changeset` function, which will return a Changeset:

```
def changeset(material, attrs) do
  material
  |> cast(attrs, [:name, :amount])
  |> validate_required(:name, :amount)
end
```

In this code, when being called for the purposes of `new`, the `material` variable is equal to `%Material{}`, and the `attrs` variable is equal to `%{}`.

`material` is the current version of the Schema, and the Map in `attrs` contains new values that will be added to the Schema, replacing old values if there's a conflict.

In `cast`, we define the values that are *allowed* to change.

In `validate_required`, we define the values that *must* be present for the Changeset to be valid. You'll note that we do not yet have either of the required values present, so our Changeset is not currently valid.

Both of the above functions can take either a Schema or a Changeset (or a `{data, types}` tuple, but we won't go into that), and they both output a Changeset.

It's that Changeset that gets passed to the `changeset` variable, then put on the `assigns` Map (to be accessed as `@changeset`), and then used in our form.

Forms

Let's take a look at our form:

```
<%= form_for @changeset, @action, fn f -> %>
<%= if @changeset.action do %>
  <div class="alert alert-danger">
    <p>Oops, something went wrong! Please check the errors below.</p>
  </div>
<% end %>
```

```

<%= label f, :name %>
<%= text_input f, :name %>
<%= error_tag f, :name %>

<%= label f, :amount %>
<%= number_input f, :amount %>
<%= error_tag f, :amount %>

<div>
  <%= submit "Save" %>
</div>
<% end %>

```

Here's it again in highly simplified form:

```

<%= form_for @changeset, @action, fn f -> %>
  <!-- error message display -->
  <!-- name field -->
  <!-- amount field -->

  <div><%= submit "Save" %></div>
<% end %>

```

`form_for` is a helper that takes three arguments: `@changeset`, `@action`, and a function with one argument (`f`), which is short for “form”. The `submit` helper, when used withn a `form_for` helper, submits the form using the `@action` defined (in this case, we passed in the action when rendering the `form` partial.)

There are many ways you can choose to display a field, and we'll go over those variations later in this book, but for now let's look at the standard generated way.

```

<%= label f, :name %>
<%= text_input f, :name %>
<%= error_tag f, :name %>

```

There are three parts to this.

The first is the label. It's what displays above the input field. The label helper can take one to four arguments (or, said another way, there are at least four definitions of the `label` helper). We're using the one with two arguments: the form as defined by `form_for`, `f`, and the field in the `material` Schema that it applies to, `:name`. It will automatically use that field name to generate the label, but if you want something different you can also customize it via a third argument.

The second is the input field itself. The user will use this to type in the value for the field. Here it's `text_input`, but there are many input fields that match

different data types (such as `number_input`, which is used for `amount`). This takes two arguments plus an optional options hash. The first argument is once again the form as defined by `form_for`, `f`, and the second argument is once again the field in the `material` Schema that it applies to, `:name`.

The third is the error tag. This is what displays errors for this field. These errors will be calculated on the Changeset when we post to the `create` route, then displayed here. You already know what the first and second arguments are, because they're the same as before, but now it's even simpler because there are no optional fields.

If you understood the last three paragraphs, then you should be able to grok the three lines used for `:amount`.

```
<%= label f, :amount %>
<%= number_input f, :amount %>
<%= error_tag f, :amount %>
```

So we'll go ahead and move on to the last part of the form:

```
<%= if @changeset.action do %>
  <div class="alert alert-danger">
    <p>Oops, something went wrong! Please check the errors below.</p>
  </div>
<% end %>
```

When we submit the form and post to the `create` route, it will either succeed or there will be an error. If there's an error, we'll end up back here with a new Changeset, and this one will have an `action` on it, and we'll display the error message. So it'll have this error message on top, plus the specific error messages below individual fields.

So now we've finally finished covering the `new` route... but to actually create a new instance of `Material` and save it to the database, we need to hit the `create` route as well.

The `create` Controller Function

When rendering the `form` partial, we passed it `Routes.material_path(@conn, :create)` as the action. That means that when we hit the `submit` button, we'll go to the `create` function on the `MaterialController`.

Let's take a look at it.

```
def create(conn, %{ "material" => material_params }) do
  case Trade.create_material(material_params) do
    {:ok, material} ->
      conn
      |> put_flash(:info, "Material created successfully.")
```

```

|> redirect(to: Routes.material_path(conn, :show, material))

{:error, %Ecto.Changeset{} = changeset} ->
  render(conn, "new.html", changeset: changeset)
end
end

```

This is the most complex controller function we've seen, but there are no new concepts here yet (there are some in `Trade.create_material`, even though it's just three lines). Let's break it down.

First, we take two arguments: `conn`, the connection, and then the params, which we're pattern-matching for "`material`" and assigning that to `material_params`. These params are what we filled out in the field, and will currently hold `:amount` and `:name`.

Then we open up a case statement. The expression is the results from `Trade.create_material(material_params)`, and it's a tuple. There are two options for the tuple, as seen below: `{:ok, material}` and `{:error, %Ecto.Changeset{}}`. That is to say, it can return successfully with a newly created `material`, or it can return unsuccessfully with a new version of the Changeset.

If it returns successfully, we do something very similar to what we did in the `delete` function, with the differences being in the flash message used and the route that we send the user to (`:show` instead of `:index`).

If it returns unsuccessfully, then we re-render the `new` route, but with the new version of the Changeset. Now it will have the values we tried to apply (so the user doesn't have to fill out the form again) and the errors (so the user knows what needs to be fixed).

Now let's take a look at what's happening in `Trade.create_material` that gives us that tuple.

Creating the Material

Here's the code for `Trade.create_material`:

```

def create_material(attrs \\ %{}) do
  %Material{}
  |> Material.changeset(attrs)
  |> Repo.insert()
end

```

We start off with a blank Material Schema (`%Material{}`).

Then we pipe that into `Material.changeset`, and have the `attrs` as the second argument. Recall that this is what we did earlier in `Trade.change_material`,

which prepared a Changeset for the `new` function, but now we're feeding it actual attributes passed in from the form, rather than a blank Map (although a blank Map is the default if you don't pass anything in).

Finally, we pipe the Changeset we just created to `Repo.insert`. This will try to create a new record in the database, and this is where we get the success or failure tuples that we received in the `create` controller function.

Conclusion

We covered a lot in this chapter.

First, we saw how the `new` route is displayed. We generate a blank Material Changeset using `Trade.change_material`, then feed it to `lib/star_tracker_web/templates/material/new.html.eex` template. That template then renders the `form` partial, passing down the Changeset and an action (`Routes.material_path(@conn, :create)`). Then we used the `form_for` helper to specify how the form is displayed.

Then we saw what happened once the user hits the submit button and sends their form data to the `create` route. We take the form data and feed it to `Trade.create_material`. That method creates a Changeset from the data and tries to put it into the database via `Repo.insert`. If successful it returns `{:ok, material}`, and then we take the newly created material and redirect to its `show` page. If failed, it returns `{:error, %Changeset{} = changeset}`, and then we use the new Changeset to re-render the `new` page.

In the next chapter we're going to see the `edit` and `update` routes, which are very similar to the `new` and `create` routes, but with some minor changes. That will be a great review if the concepts of this chapter are still a little shaky.

But first, let's solidify what you've learned with some exercises.

Exercises

1. Changesets:
 - a. Change the Changeset so that the `amount` field isn't required, and will not error even if no amount is given.
 - b. Now change it so `amount` isn't allowed in the changeset, and will error if any number is given.
 - c. Make it so that there are always 1000 of any material that's created.
 - d. Challenge: allow the `amount` field to be filled out, but have 1000 as a default if it's not.
2. Replace the generic error message (the part sectioned off by `if @changeset.action`) with a flash error message in the error case of

create. After doing that, think about why Phoenix chose to do it the way they did, rather than use a flash message.

Editing a Material

We covered a lot in the previous chapter, but the knowledge gained isn't super flexible because we had to cover so much. In this chapter and the next we'll review these concepts and look at different options and how we can use them in new ways.

The first variation, covered in this chapter, is editing an existing Material instead of creating a new one.

The `edit` Controller function

Let's look at `new` and `edit` side by side.

```
def new(conn, _params) do
  changeset = Trade.change_material(%Material{})
  render(conn, "new.html", changeset: changeset)
end

def edit(conn, %{"id" => id}) do
  material = Trade.get_material!(id)
  changeset = Trade.change_material(material)
  render(conn, "edit.html", material: material, changeset: changeset)
end
```

They're similar in structure, although `edit` is a bit more complex.

First, instead of starting with a blank Schema (`%Material{}`), we're getting a previously-created material from the database. We're then feeding that into `Trade.change_material`, so our Changeset will contain the previously assigned values.

Second, when rendering, we render the `edit` template instead of the `new` template.

Finally, we send the `material` down alongside the `changeset`.

The `edit` Template

This is what gets rendered:

```
<h1>Edit Material</h1>

<%= render "form.html", Map.put(assigns, :action, Routes.material_path(@conn, :update, @mate
```

This is *very* similar to the `new` template. The only differences are that instead of saying “Create Material” we say “Edit Material”, and the action is `update` instead of `create`. The `update` action is the only place we use the `@material` we assigned.

The `form` that we render is the exact same as the one we rendered in `new`. This reuse is the reason we put it in a separate template file. The only difference is that the Changeset we give it already has values for the fields, so they’ll already be filled out.

Then, when the user hits the submit button, it will go to the `update` function in the `MaterialController`.

The `update` Controller function

Once again, we’ll compare what we did last chapter with the `new` function.

```
def create(conn, %{"material" => material_params}) do
  case Trade.create_material(material_params) do
    {:ok, material} ->
      conn
      |> put_flash(:info, "Material created successfully.")
      |> redirect(to: Routes.material_path(conn, :show, material))

    {:error, %Ecto.Changeset{} = changeset} ->
      render(conn, "new.html", changeset: changeset)
  end
end

def update(conn, %{"id" => id, "material" => material_params}) do
  material = Trade.get_material!(id)

  case Trade.update_material(material, material_params) do
    {:ok, material} ->
      conn
      |> put_flash(:info, "Material updated successfully.")
      |> redirect(to: Routes.material_path(conn, :show, material))

    {:error, %Ecto.Changeset{} = changeset} ->
      render(conn, "edit.html", material: material, changeset: changeset)
  end
end
```

As you can see, `create` and `update` are structured very similarly. They both destructure params, then they have a `case` statement where the expression is a call to the `Trade` context, and then they have success and failure conditions which match based on the same tuples. The `success` condition is, in fact, the

almost the exact same, except the message says “updated” instead of “created”. The `error` condition is similar, except it renders `edit` instead of `new`, and passes down a `material` in addition to a `changeset`.

However, despite the significant similarities, there are some important differences.

First, in `update`, we’re working with a pre-existing material, so we match the `"id"` parameter, then use that to get the material (`Trade.get_material!(id)`), and feed that material into `Trade.update_material(material, material_params)`.

The other major difference is hidden in `Trade.update_material`, so let’s look at that.

```
def create_material(attrs \\ %{}) do
  %Material{}
  |> Material.changeset(attrs)
  |> Repo.insert()
end

def update_material(%Material{} = material, attrs) do
  material
  |> Material.changeset(attrs)
  |> Repo.update()
end
```

As you can see, we’re working with the pre-existing material, instead of a blank `%Material{}` Schema. Then we pass it into the changeset with the new attributes, as before. Then, instead of inserting a new database record, we update the existing database record.

Conclusion

As promised, this was much shorter than the previous chapter. Instead of learning all new concepts, we only had to repurpose what we’d already learned.

The biggest difference is that we were working with an already-existing material, so from the start our Changeset had valid data.

Exercises

Let’s test some edge cases.

1. In `update`, feed `Trade.update_material` an empty `%Material{}` instead of the material from the database for the first argument. What errors occur? Why?

2. In `update`, feed `Trade.update_material` an empty `%{}` instead of the `material_params` from the POST call for the second argument. What errors occur? Why?
3. In `edit`, feed `Trade.change_material` an empty `%Material{}` instead of the material from the database. Leave everything else the same. What errors occur? Why?

Adding a Field

We've now completed our tour of the generated Material resource. We've gone through the index page, the show page, the new page, the edit page, and the create, update, and delete actions.

Along the way we've learned a lot:

- routes: resources, alternate methods (post, delete, etc.), redirects
- interacting with the database via `Repo` methods
- Changesets
- Schemas
- new pattern-matching techniques
- flash messages
- templates: partials, forms, tables

Seriously, that's a lot. Good job.

Now it's time to take a quick victory lap.

In this chapter we're going to be adding a `price` field to our Materials. We'll be learning one new concept (migrations) and then reviewing a lot of stuff we've learned in the previous chapters in this section.

We'll need to update our app in six different places:

1. The database
2. The schema
3. The changeset
4. The form
5. The show page
6. The index page

So let's get started!

Changing the database with migrations

First we need to add `price` to the `materials` database table. We'll do this via a migration.

Let's take a look at the migration we've already used. The filename will vary slightly, since part of it is determined by the time at which it was created, but it should be the only migration file in the `priv/repo/migrations` folder.

```
defmodule StarTracker.Repo.Migrations.CreateMaterials do
  use Ecto.Migration

  def change do
    create table(:materials) do
```

```

    add :name, :string
    add :amount, :integer

    timestamps()
end

end
end

```

We have several macros (`create`, `table`, `add`, and `timestamps`) that we get from `use Ecto.Migration`. There are others, including another that we'll be learning this section, but those are the four used in this generated migration.

All the action is happening within the `change` function.

We create a table called `:materials`, and then we add two fields to it: `:name` and `:amount`, which have the types of `:string` and `:integer`. The field name tells us the key to store the value under, and the type tells us what form the values will take.

We also add `timestamps()`, which adds `inserted_at` and `updated_at` fields with a type of `:naive_datetime`.

So that's what the generate migration does. Now it's time to create the migration that will add `price` to the database.

Let's go to the command line to generate it.

```
$ mix ecto.gen.migration add_price_to_materials
* creating priv/repo/migrations/20190509073235_add_price_to_materials.exs
```

Now let's find the file that was just generated. Your filename will differ from mine, but it should be in the same folder and end with `add_price_to_materials.exs`.

```

defmodule StarTracker.Repo.Migrations.AddPriceToMaterials do
  use Ecto.Migration

  def change do
    end
end

```

It gives us the boilerplate, setting up a blank `change` function. Let's fill it in.

```

defmodule StarTracker.Repo.Migrations.AddPriceToMaterials do
  use Ecto.Migration

  def change do
    alter table(:materials) do

```

```
    add :price, :integer
  end
end
end
```

Instead of creating a table with `create`, we're going to alter it with `alter`. This is another macro that `use Ecto.Migration` gives us.

The syntax for adding a field is the same as before, and just like `:amount`, `:price` is an integer.

Now let's go ahead and use this migration to change the database.

```
$ mix ecto.migrate
[info] == Running 20190509073235 StarTracker.Repo.Migrations.AddPriceToMaterials.change/0 ...
[info] alter table materials
[info] == Migrated 20190509073235 in 0.0s
```

Notice that we only ran this one migration, and not the previous one. That's because `mix ecto.migrate` will only run the migrations that haven't already been run.

If you try it again, you'll see that no changes are made, since everything's already been run.

```
$ mix ecto.migrate
[info] Already up
```

So now our database table has a new field! Let's change the rest of our Ecto setup to match.

Changing the Schema and Changeset

We define both the Schema and the Changeset for our materials in the `StarTracker.Trade.Material` module. Let's update those to include the `price`.

```
defmodule StarTracker.Trade.Material do
  use Ecto.Schema
  import Ecto.Changeset

  schema "materials" do
    field :amount, :integer
    field :name, :string
    field :price, :integer

    timestamps()
  end
```

```
@doc false
def changeset(material, attrs) do
  material
  |> cast(attrs, [:name, :amount, :price])
  |> validate_required([:name, :amount, :price])
end
end
```

We've added the line `field :price, :integer` to the schema, which will now allow that in all `%Material{}` Schemas.

We've also added `:price` to both the `cast` and `validate_required` portions of the `changeset` pipeline. This means that when the user fills out the price on the form it will get turned into data in the database, and that if they don't fill anything out for the price then there will be an error.

And, of course, that error will happen automatically if we don't add the price field to the form, so let's go do that.

Adding the price field to the form

We're going to copy the code for `:amount` and just change the atom used to `:price`. We'll also customize the label a bit by using the optional third argument, but that's about it.

```
<%= form_for @changeset, @action, fn f -> %>
  <%= if @changeset.action do %>
    <div class="alert alert-danger">
      <p>Oops, something went wrong! Please check the errors below.</p>
    </div>
  <% end %>

  <%= label f, :name %>
  <%= text_input f, :name %>
  <%= error_tag f, :name %>

  <%= label f, :amount %>
  <%= number_input f, :amount %>
  <%= error_tag f, :amount %>

  <%= label f, :price, "Price Per Unit" %>
  <%= number_input f, :price %>
  <%= error_tag f, :price %>

<div>
  <%= submit "Save" %>
```

```
</div>
<% end %>
```

Now we've got it displaying the price.

localhost:4000/materials/2/edit

Edit Material

Name

Amount

Price per unit

SAVE

[Back](#)

Price on the show page

To add the price to the show page, we'll extend the pattern we've used for the other attributes.

```
<h1>Show Material</h1>

<ul>
  <li>
    <strong>Name:</strong>
    <%= @material.name %>
  </li>
  <li>
    <strong>Amount:</strong>
    <%= @material.amount %>
  </li>
  <li>
    <strong>Price:</strong>
    <%= @material.price %>
  </li>
</ul>

<span><%= link "Edit", to: Routes.material_path(@conn, :edit, @material) %></span>
<span><%= link "Back", to: Routes.material_path(@conn, :index) %></span>
```

There's no new concepts here, we're just repeating the pattern with our new field.

① localhost:4000/materials/2

Show Material

- **Name:** Carbon
- **Amount:** 500
- **Price:** 9

[Edit](#) [Back](#)

Price on the Index page

To add the price to the Index page, we'll add to our loop a table data cell (`td`) containing the price of our material. We'll also add a table header cell (`th`) that contains the name of the column.

```
<h1>Listing Materials</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Amount</th>
      <th>Price</th>

      <th></th>
    </tr>
  </thead>
  <tbody>
    <%= for material <- @materials do %>
    <tr>
      <td><%= material.name %></td>
      <td><%= material.amount %></td>
      <td><%= material.price %></td>

      <td>
        <%= link "Show", to: Routes.material_path(@conn, :show, material) %>
        <%= link "Edit", to: Routes.material_path(@conn, :edit, material) %>
        <%= link "Delete", to: Routes.material_path(@conn, :delete, material), method: :de
    </td>
  </tbody>
```

```
</tr>
<% end %>
</tbody>
</table>

<span><%= link "New Material", to: Routes.material_path(@conn, :new) %></span>
```

Once again, there are no new concepts in this code. This is the logical extension of the way we've displayed data in the table before.

Name	Amount	Price
Helium	200	Show Edit Delete
Oxygen	800	Show Edit Delete
Carbon	500	9 Show Edit Delete

New Material

Conclusion

In this chapter we added the `:price` field to our materials.

We learned how to create and run a custom migration to add it to the database, then how to edit our Schema and Changeset so our code can interact with the database correctly. Although we'd touched on these before, this is the first time we made serious changes.

Finally we updated our templates: the edit form, the show page, and the index page. These simply extended the patterns we'd set forth in previous versions of the templates. However, there are sometimes reasons to make more extensive edits to the templates. In future chapters we'll see how complex data can make it necessary to change. In the next chapter, which is optional, we'll show how to change how our app looks through extensive changes in the template's display code.

Exercises

1. Add a `description` field to our app.
 - a. Create and run the migration that adds it.
 - b. Add it to the schema and changeset; changing it should be allowed but not required

- c. Add the field to the form. Hint: look up the `textarea` input field.
- d. Display the field in the show page.
- e. Display a checkmark (html code: `✔`) in the index page if a description is present;

Appendix: Use, Import, Alias

One of the most important tasks in any programming environment is bringing in code from elsewhere; it lets you reuse functionality and keep everything clean. Elixir and Phoenix have a series of elegant tools available to do that.

In this appendix we're going to focus on the `use`, `import`, and `alias` keywords and what they specifically bring to our Router and Controller files. There are, of course, functions beyond this (config files, mix, etc.).

`use StarTrackerWeb, :router`

In the tutorial we had the following Router file:

```
defmodule StarTrackerWeb.Router do
  use StarTrackerWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", StarTrackerWeb do
    pipe_through :browser

    get "/", PageController, :index
    get "/info", PageController, :info
  end

  # Other scopes may use custom stacks.
  # scope "/api", StarTrackerWeb do
  #   pipe_through :api
  # end
end
```

We explained briefly what each of the macros (`pipeline`, `plug`, `scope`, `pipe_through`, `get`) did, but not where they came from. They're not defined in the core Elixir libraries... instead, they're all made available by the line `use StarTrackerWeb, :router`.

The first link of the chain can be found in `lib/star_tracker_web.ex`.

```
defmodule StarTrackerWeb do
  # ...

  def router do
    quote do
      use Phoenix.Router
      import Plug.Conn
      import Phoenix.Controller
    end
  end

  # ...

  @doc """
  When used, dispatch to the appropriate controller/view/etc.
  """
  defmacro __using__(which) when is_atom(which) do
    apply(__MODULE__, which, [])
  end
end
```

The comments with `...` are simply code blocks that I've left out in order to make the explanation more concise.

The path through this code starts with `defmacro __using__`. That is what is called first by Elixir when someone calls `use ThisModuleName`. So it takes one argument, which in this case is `:router`. The only line in `defmacro __using__` then uses that argument to call a function within the module with the name of that argument- in this case, the `router` function. A convoluted way of doing things, but having this complication once means that every Phoenix app gets to enjoy a nicer syntax every time they use the `use MyAppWeb` interface.

quote

There are lots of cool complicated things you can do with `quote`, but in this file it sticks to doing one relatively simple thing- taking what's within the `quote do ... end` block and dumping it into the file that called the macro where `quote` is used.

So, in this case, the effect of putting `use StarTrackerWeb, :router` in your file is the same as putting in `use Phoenix.Router`, as well as `import Plug.Conn` and `import Phoenix.Controller`.

Why the added layer of indirection? Having this indirection allows us to put in "standard" things that will show up in every instance where the macro is used.

It's much easier to type in `use StarTrackerWeb, :router` than all three lines, and if you want to change those lines it's better to have them in one centralized place. This is more useful for Controllers and Views, since there's generally one Router per Phoenix app, but many Controllers and Views.

Previously On: Indirection

"Indirection" can mean various things in a programming context, but when it's used as in "layer of indirection" people generally mean that there are several function calls that don't seem strictly necessary but are there for organizational purposes.

There are cases where these organizational purposes are valid and worth the cost of adding a layer of indirection (Phoenix usually makes good choices in this regard), but there are also cases where it's meaningless complication that just makes your program harder to understand.

One of the marks of an experienced developer is that they understand the pros and cons of adding specific layers of indirection. For now, just go with Phoenix's defaults unless you have a very compelling reason not to.

So now that we see that this use of `use StarTrackerWeb, :router` gives us the same effect as `use Phoenix.Router, import Plug.Conn, and import Phoenix.Controller`. Let's see what that first command brings us.

use Phoenix.Router

So far we've gotten all our modules from `StarTracker`- our own web app. Now we're getting a module from `Phoenix`, so we'll have to dig into where the framework code is stored.

deps

The path for that is in the `deps` folder. Your text editor may have that folder greyed out, but it should still be available for browsing. If your text editor has it completely hidden, you can use the command line (`ls` and `cd` commands) or your OS's file browser to explore.

Within the `deps` are all your app's dependencies, brought in from across the web. Currently we're only interested in the `phoenix` folder. Specifically, the file we're looking for has a path of `deps/phoenix/lib/phoenix/router.ex`.

Phoenix.Router

Here's a simplified version of that file:

```
defmodule Phoenix.Router do
  @http_methods [:get, :post, :put, :patch, :delete, :options, :connect, :trace, :head]

  def __using__(_) do
    defmacro __before_compile__(env) do #
    defmacro match(verb, path, plug, plug_opts, options \\ []) do #
      for verb <- @http_methods do
        defmacro unquote(verb)(path, plug, plug_opts, options \\ []) do #
          add_route(:match, unquote(verb), path, plug, plug_opts, options)
        end
      end
    end

    defmacro pipeline(plug, do: block) do #
    defmacro plug(plug, opts \\ []) do #
    defmacro pipe_through(pipes) do #
    defmacro resources(path, controller, opts, do: nested_context) do #
    defmacro resources(path, controller, do: nested_context) do #
    defmacro resources(path, controller, opts) do #
    defmacro resources(path, controller) do #
    defmacro scope(options, do: context) do #
    defmacro scope(path, options, do: context) do#
    defmacro scope(path, alias, options, do: context) do #

      defmacro forward(path, plug, plug_opts \\ [], router_opts \\ []) do #
    end
  end
end
```

You can see that we're defining all of the macros which we used earlier, including several versions of `scope` (remember: pattern matching based on number of arguments), as well as a couple more we haven't seen yet.

That's enough of a dive into the source code- we won't go into detail on how each of them is defined... although you're welcome to explore as much as you like on your own.

import

When we used `use StarTrackerWeb, :router`, it also automatically gave us two import statements: `import Plug.Conn`, and `import Phoenix.Controller`.

Like `use`, `import` drops function definitions directly into the current module's namespace.

What sets them apart: `use` calls the `__using__` callback in order to inject code into the current module, and `import` doesn't. `import` can also have the `only` option, which takes a list of function names (and their arities) and then only puts those specific functions into the current module's namespace.

We'll be using `import` often in our own code, and we'll recap it the first time that happens.

use StarTracker.Web, :controller

In our controller we call `use StarTrackerWeb, :controller`. Like in the Router, this takes us to our `star_tracker_web.ex` file.

```
defmodule StarTrackerWeb do
  # ...
  def controller do
    quote do
      use Phoenix.Controller, namespace: StarTrackerWeb

      import Plug.Conn
      import StarTrackerWeb.Gettext
      alias StarTrackerWeb.Router.Helpers, as: Routes
    end
  end
  # ...
  def router do
    quote do
      use Phoenix.Router
      import Plug.Conn
      import Phoenix.Controller
    end
  end
  # ...

  @doc """
  When used, dispatch to the appropriate controller/view/etc.
  """
  defmacro __using__(which) when is_atom(which) do
    apply(__MODULE__, which, [])
  end
end
```

We're `use-ing` and `import-ing` some different files, and we're also using `alias`. Let's tackle the now-familiar `use` first.

use Phoenix.Controller

Here's a highly-compacted version of `Phoenix.Controller`, with 90% of the function names and 100% of the actual code redacted for simplicity's sake (the actual file is 1475 lines long, as of this writing).

```
defmodule Phoenix.Controller do
  #...
  def html(conn, data) do #...
  def redirect(conn, opts) when is_list(opts) do #...

  # ...

  def render(conn, template_or_assigns \\ [])
  def render(conn, template) when is_binary(template) or is_atom(template) do # ...
  def render(conn, assigns) do #...

  def render(conn, template, assigns) when is_atom(template) and (is_map(assigns) or is_list(assigns)) do # ...
  def render(conn, template, assigns) when is_binary(template) and (is_map(assigns) or is_list(assigns)) do # ...
  def render(conn, view, template) when is_atom(view) and (is_binary(template) or is_atom(template)) do # ...
  def render(conn, view, template, assigns) when is_atom(view) and (is_binary(template) or is_atom(template)) do # ...

  #...
end
```

Here we see 7 different definitions of `render` (the one Controller function we've used so far), each differentiated by the number of arguments (arity) as well as some `when` clauses (a particular definition of `render` will only be used if its `when` clause is true).

There are lots more definitions in this file, including `html` and `redirect`, but for now there's nothing else for us to learn here. Time to look at the other things brought in by `use StarTracker.Web, :controller`.

alias

The final line for `controller` is `alias StarTrackerWeb.Router.Helpers, as: Routes`. We haven't used this yet, but the explanation is easy *if you don't worry about what StarTrackerWeb.Router.Helpers actually does*.

Let's start with a slightly simpler case. If we had `alias StarTrackerWeb.Router.Helpers`, then whenever we want to access any function in that module, we could leave off the `StarTrackerWeb.Router` part, and just reference `Helpers.function_name`. This makes things more convenient.

Then we add the `as` option to the alias. `alias StarTrackerWeb.Router.Helpers, as: Routes`,

`as: Routes` allows us to call `Routes.function_name`. The default usage is equivalent to putting the last part of the aliased module in the `as` option (`alias StarTrackerWeb.Router.Helpers, as: Helpers`).

We'll be using `alias` often in our own code, and we'll recap it the first time that happens.

Conclusion

So that's how we get all the macros like `render`, `get`, and `scope`.

I hope this demystified some of the Elixir Magic for you. If not, then don't worry too much about it; although an understanding of the concepts in this appendix is helpful, it's by no means required, and you can make a great app while still thinking of `use` as just a mysterious line that makes your Router and Controller work.