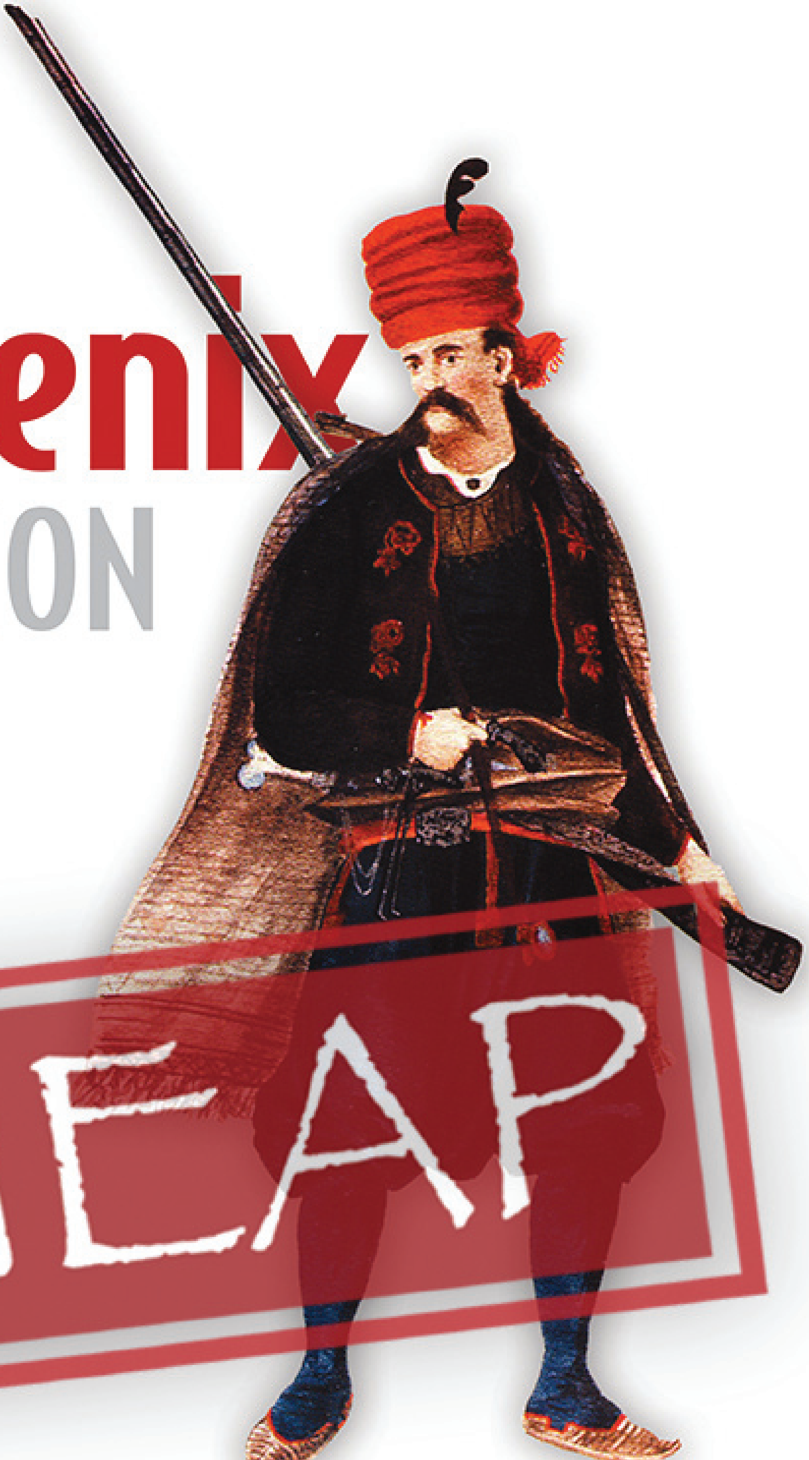
Phoenix IN ACTION

Geoffrey Lessel



MANNING

MEAP Edition
Manning Early Access Program

Phoenix in Action

Version 7

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Phoenix in Action*. Elixir and Phoenix have had an exciting past few years as the Elixir language and the Phoenix framework have both started to take off and become legitimate players in the web development world. There are now many companies that have staked their future business on the fact that Phoenix sites built on Elixir are the future of web development.

The incredible processing speed of Elixir combined with its built-in support for parallel and distributed programming (not to mention its fault-tolerance) really shine when paired with the Phoenix framework. With Phoenix, web development with Elixir is quick and enjoyable with speed of development and developer happiness major selling points.

While *Phoenix in Action* will not teach you everything there is to know about Elixir, it does have a chapter introducing you to (or refreshing you on) the basics of the Elixir language. As more complex subjects come up during the book, there will likely be a sidebar describing the new Elixir concept and definitely footnotes directing you to where you can find more or deeper information regarding those features if you so choose.

The flow of the chapters of *Phoenix in Action* will have you building a real-time auction website step-by-step, first starting with business logic completely separate from Phoenix and slowly adding more and more complex features and introducing Phoenix-specific functionality along the way.

While Elixir and Phoenix are certainly production-ready, they are still relative newcomers in the world of application development. Because of that, there are still discussions around best practices and the most appropriate way to structure applications and even where to introduce functionality. One of the best things about Phoenix is that it stays out of your way, keeping most of its opinions to itself. But the other side of that coin is that there can be many competing opinions. This book will have its own opinion regarding how to structure our application, but there are other ways to do it that are no less correct. But *Phoenix in Action* will encourage you to make Phoenix only a single boundary to your real underlying application logic.

I'm excited about Elixir and Phoenix and what it will mean for the future of application development, specifically for the web. I hope that you join me in this excitement and provide feedback and comments during the development of *Phoenix in Action*. Your involvement will be essential while I continue writing this book, so please don't be shy in your interactions with me.

Many thanks,
—Geoffrey Lessel

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

brief contents

PART 1: AN INTRODUCTION

1 *Ride the Phoenix*

2 *Intro to Elixir*

3 *A Little Phoenix Overview*

PART 2: THE BASICS

4 *Phoenix is Not Your Application—Just a Boundary*

5 *Elixir Application Structure*

6 *Bring in Phoenix*

7 *Being Persistent with a Database*

8 *Making Changes with Ecto.Changeset*

9 *Transforming Data In Your Browser*

10 *Plugs, Assigns, and Dealing With Session Data*

11 *Associating Records and Accepting Bids*

PART 3: GETTING ADVANCED

12 *Using Phoenix Channels For Real-Time Communication*

13 *Real-time updates with channels*

14 *Testing*

1 Ride the Phoenix

This chapter covers:

- What Phoenix is and the benefits it provides over its alternatives
- The power of the Elixir programming language
- A brief overview of the differences between object-oriented and functional programming
- Some of the potential drawbacks to using Phoenix and Elixir

In this chapter, you will be introduced to the Elixir programming language and the Phoenix web framework. We will be taking a look at the main advantages and disadvantages a language and framework like Elixir and Phoenix provides. By the end of the chapter, you'll have a good overview of the technologies, why they exist, and you'll be ready to jump into learning the details and creating your own web applications.

1.1 Along Comes Phoenix

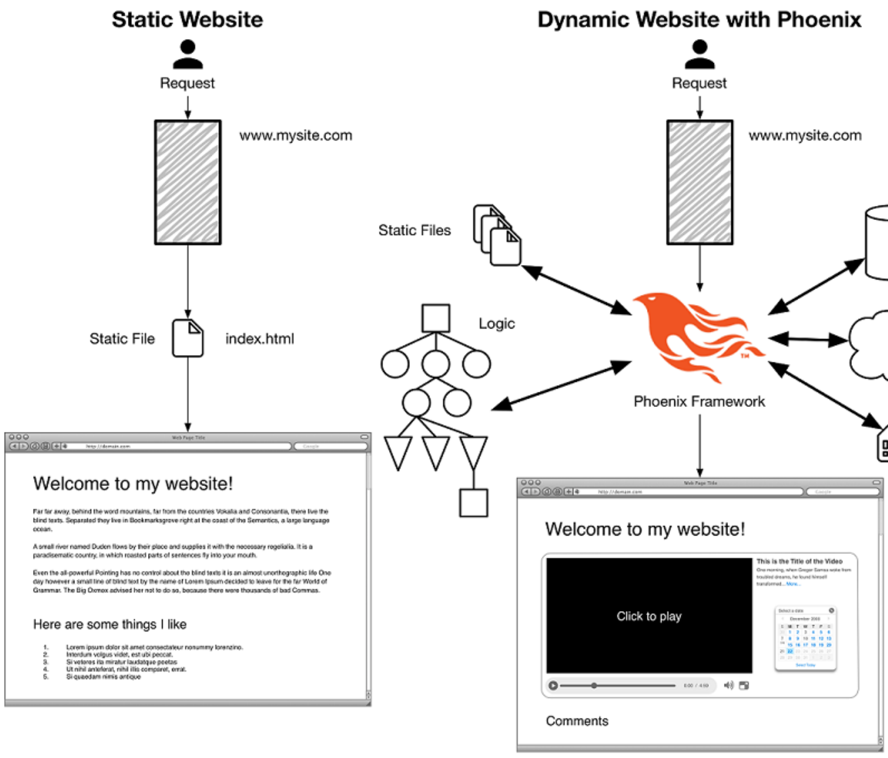
While people have been creating websites for decades now, the tools they use to create them have changed. In the early days, developers used simple text editors on their computers to type out all the HTML needed in order to hyperlink one page to another on their website. But once interactivity with the user was desired, things got complicated. How can you create an e-commerce site with plain HTML? How can you get information from a central database and display pages dynamically? What if you wanted to display personalized information for every user that visited?

All these questions and more have been answered differently as time marched on. We've gone from Perl scripts to PHP to Python, Java, C# and Ruby (not necessarily in that order and leaving out many other languages). These languages and others allowed the server to respond dynamically to information given to it via

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

a user's interaction with the web site. Figure 1 shows how much more complex a dynamic website may be over a static one.

Figure 1.1. Static websites vs. dynamic websites that frameworks help provide.



Arguably the biggest impact on the open-source web in the 2000's was the release and adoption of Ruby on Rails—a web framework written on the Ruby programming language. This allowed developers to quickly and somewhat easily get a dynamic web site up and running in very little time. The famous example from that time was creating a blogging engine in 15 minutes. It allowed developers great productivity and "Web 2.0" exploded.

From Ruby on Rails, many new web frameworks attempted to match or outdo Ruby on Rails in terms of features and developer happiness. There became a go-to web framework for each major programming language. In that vein Phoenix was born.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Other popular web frameworks

Language	Framework
Elixir	Phoenix
Ruby	Ruby on Rails, Sinatra
Python	Flask, Django
PHP	Laravel, Symfony
Java	Spring, Ninja
Javascript	Meteor, Express, Sails
C#	Nancy, ASP.NET Boilerplate
Perl	Mojo, Catalyst

1.1.1 What is Phoenix?

Phoenix is a web framework that attempts (and, in my opinion, succeeds in) to help the creation and maintenance of dynamic websites much simpler. Phoenix does not attempt to copy the big player in this space: Ruby on Rails. Phoenix is built on top of the Elixir programming language (which is a functional language) while Ruby is an Object Oriented programming language (we'll get into the details of the differences later in this chapter). While Phoenix has best practices for structuring and maintaining your application, it is not as dogmatic about it as Ruby on Rails and other frameworks are — giving you flexibility to write your application how you see fit.

Phoenix goes beyond many web frameworks as well. For example, Phoenix has the concept of Channels built-in. Channels allow for "soft-realtime" features to be easily added to applications. We'll cover the topic of Channels further in-depth in later chapters, but this is what allows you have chat applications, instant push updates to thousands of simultaneous users, and page updates for the user without a full-page refresh. While these kinds of features are difficult to add in other languages and frameworks, it borders on trivial for Phoenix applications. Whole books can be written about how to get these features into web applications using other web frameworks (or rolling your own), but we'll be adding them to our application in just a few chapters.

1.2 Elixir And Phoenix Versus the Alternatives

This chapter is not intended to be a bullet-point list of things that Phoenix and Elixir do well over other alternatives, but in order to introduce you to the power of Phoenix and Elixir, here are a few of the exciting things that Phoenix offers and that we will be covering in this book.

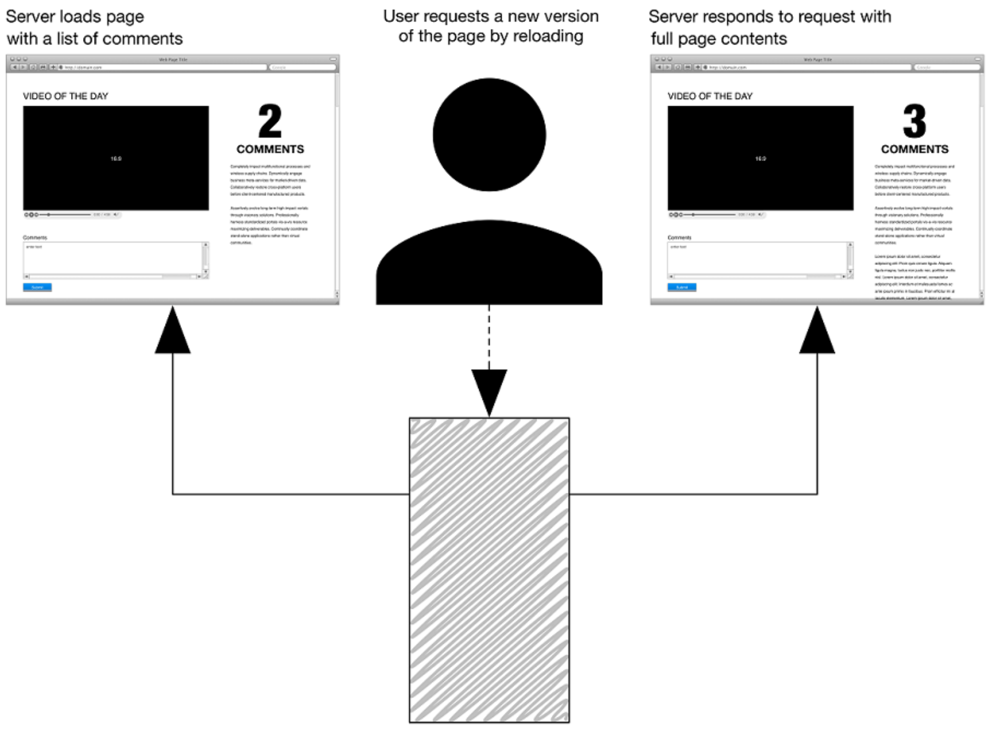
1.2.1 Real-time Communication

What can you do with real-time communication? Well, any time you want to push information to many users simultaneously, you'll need some solution for that. That

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

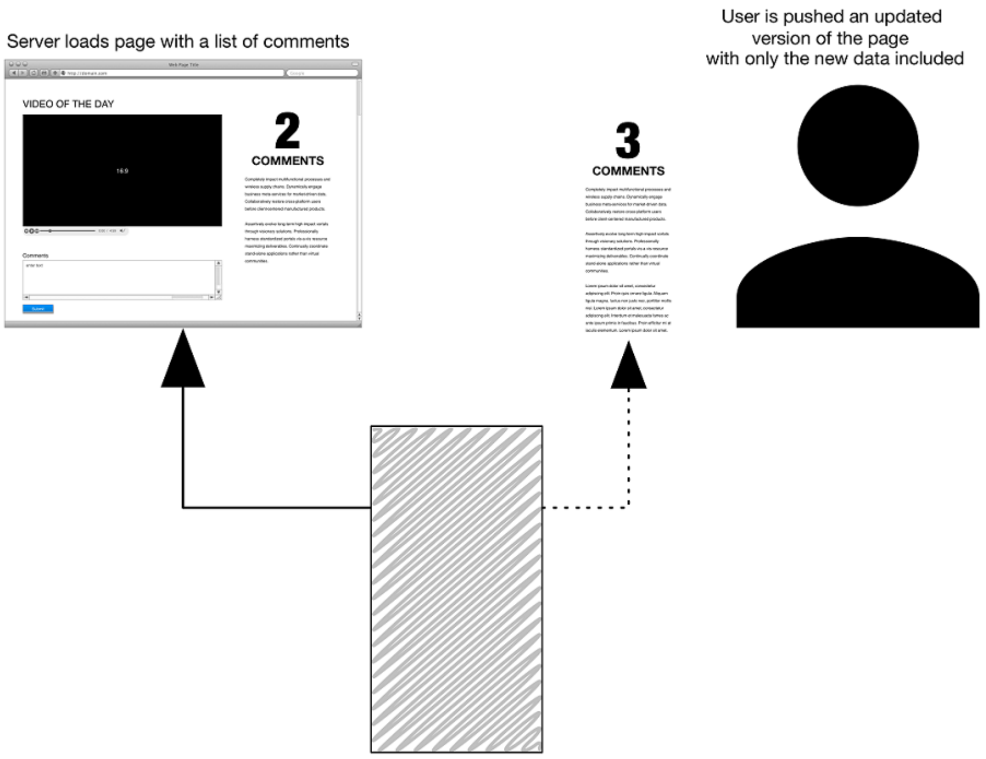
could be with a chat application where users can send and receive chat messages to thousands of other users and you need to keep each client up-to-date when another client's user submits a new message. It could be something more complex like an auction site that wants to provide users visiting the auction item's page up-to-the second information regarding the state of the bids. Maybe it's a workplace collaboration site where users are sharing files and even working on the same file simultaneously. Or perhaps there is a real-time multiplayer game server that you want to build and you need to ensure that all players have the same information at the same time. There are many different situations in which real-time channels are necessary, and many more in which it would be beneficial. Figure 7 illustrates a simple situation in which bandwidth can be saved and the user experience improved with pushing data to the user instead of requiring the user to pull the data from the server.

Figure 1.2. Traditional "pull" refreshed require the user to initiate the request and the server to return the entire web page again.


Server loads page with a list of comments
User requests a new version of the page by reloading
Server responds to request with full page contents
VIDEO OF THE DAY
2
COMMENTS
VIDEO OF THE DAY
3
COMMENTS

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Figure 1.3. A "Push" requests originates from the server and "pushes" new information to the user which is typically only that which changed, greatly reducing payload and speeding up page rendering.


Server loads page with a list of comments
VIDEO OF THE DAY
2
COMMENTS
User is pushed an updated version of the page with only the new data included
3
COMMENTS

Elixir can spawn and efficiently handle hundreds of thousands of processes without blinking an eye (we go into a bit more detail in the section about Elixir below). You’ve also read briefly about Phoenix’s concept of channels which allow real-time communication. Phoenix actually spawns each channel into its own process so that no one process can damage or take down any of the others — they are beautifully isolated.

Using these processes and a channels, Phoenix can handle a couple *hundred thousand* connections on a single server! Granted, kudos to you if your web application ever has hundreds of thousands of simultaneous users wanting to communicate in real time—but that gives you an idea of the power of Phoenix and Channels running within processes.

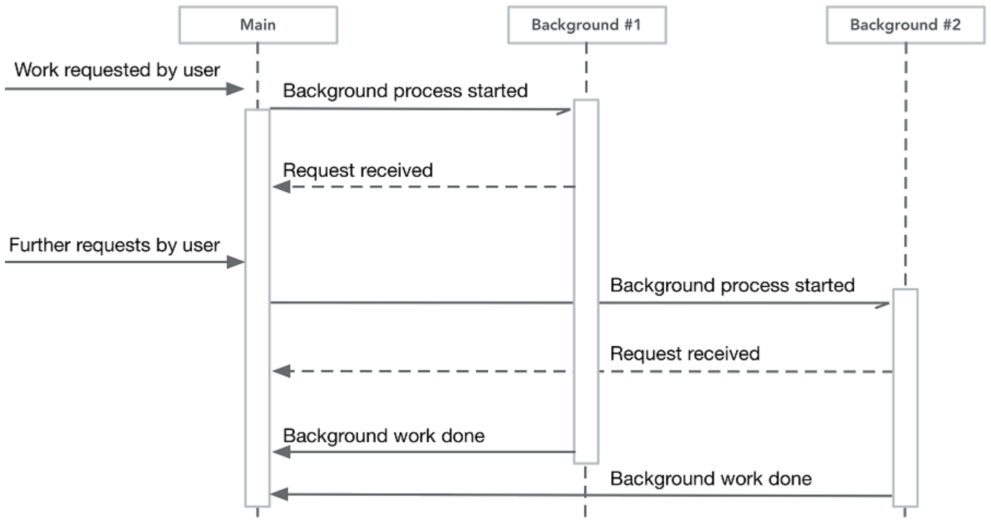
We’ll get more into Channels (and build some into our application) in later chapters.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

1.2.2 Background Computation

There will be times during your time in development that you'll want to do a long-running computation or process but don't want to keep the user from interacting with your software whether that's on the web or locally with some user interface. For example, a user of a digital product e-commerce site may want to purchase a product that is customized and then can be downloaded. Once the user finalizes the purchase, you don't want to require the user to wait on that page until the personalization process is complete. Instead, you'd like for them to continue to browse your store or set up their profile. This is a simple scenario in where background computation can play a role.

Figure 1.4. Background processes can help do complex work without holding up the main responses to the user.



If your application needs to do any more than a small amount of computation in the background (normally asynchronously), then it is a perfect fit for something built on top of Elixir like Phoenix. It is trivially easy to kick off background, asynchronous processes with Elixir.

This fits in perfectly with the real-time communication features of Phoenix. Once the long-running process is complete, the server could send a message through the Channel to the user's client that the process is complete and either display the results directly on the client or provide a link or further instructions to retrieve the results at a later time.

1.2.3 Low Hardware Requirements / Low-cost Scaling

If you look at any hosting provider's offerings, you'll notice that most of the time,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

adding more CPU power/cores costs less than adding more RAM. This fits perfectly with the way Elixir works. Elixir automatically utilizes all the different CPU cores available to it and takes up a relatively very small amount of RAM in order to run. Compare this to a web framework and language like Ruby on Rails which is RAM hungry.

Other web frameworks built on differing programming languages CAN scale, but if you are looking for a low-cost way to scale quickly, then Elixir and Phoenix would be a good choice. Pinterest explains in a blog post[1] that they moved one of their systems from Java to Elixir.

So, we like Elixir and have seen some pretty big wins with it. The system that manages rate limits for both the Pinterest API and Ads API is built in Elixir. Its 50 percent response time is around 500 microseconds with a 90 percent response time of 800 microseconds. Yes, microseconds.

We've also seen an improvement in code clarity. We're converting our notifications system from Java to Elixir. The Java version used an Actor system and weighed in at around 10,000 lines of code. The new Elixir system has shrunk this to around 1000 lines. The Elixir based system is also faster and more consistent than the Java one and runs on half the number of servers.

1.2.4 It's Not All Roses

While the Elixir language and the Phoenix framework are awesome and are usually what I reach for when starting new projects, it is not always the de-facto best choice for the job. There are some areas in which Elixir doesn't do as good a job as a different alternative.

- Time to Productivity: If you are already productive in a different web framework using a different programming language, it may be hard to justify the cost of getting up to speed in something new like Elixir. If you are just getting started in development, one consideration is how long it will take you to become productive in your new chosen language. Some people I've spoken to believe it is harder to learn functional programming than Object-oriented programming while others swear it's the other way around. Regardless, I believe that Elixir and Phoenix offer enough reasons for you to take the plunge and learn the language regardless of your past experience levels.
- Numbers: Elixir isn't a number-crunching powerhouse. It will never match the pure computational speed of something lower level like C or C++. I'd even go so far to say that if you are looking to primarily crunch large numbers, even Python is a better choice. Elixir can do it and do it well, but it won't win in a head-to-head competition in this area.
- Community: While the community behind Elixir, Phoenix, and Erlang is very helpful and enthusiastic, it is not as large as a community for something like Ruby,

[1] medium.com/@Pinterest_Engineering/introducing-new-open-source-tools-for-the-elixir-community-2f7bb0bb7d8c

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Python, Java, or even PHP. Those languages have a deeper and longer history in the web application world and it might be easier to find help with your project in one of those worlds. However, more and more meetups, conferences, blogs, jobs, etc. are arriving all the time for Elixir and I believe the future will only hold great things for the Elixir community.

- Packages: Going along with the above comment regarding the size and age of the community surrounding Elixir versus other languages, the amount of open-source packages available to use in your project is smaller. As of the time of this writing, there are currently just under 4,500 packages available on hex.pm, the package manager source for Elixir. Contrast that to rubygems which has over 8,500 packages and PyPI (the Python Package Index) which has over 109,000. While it may be harder to find something that does exactly what you used in previously programming languages, more packages are being built all the time. It also means there is space for *YOU* to create a new helpful package and help grow the community.
- Deploying: In the original draft of the table of contents of this book, a chapter regarding deploying your application was discussed. However, this is an area that is still, in all honesty, in need of a single, "best" solution. Deploying an Elixir application (including ones that utilize the Phoenix web framework) is pretty tricky and involves multiple steps. Beyond that, those steps are still not quite clearly defined nor have mature tools to help in the process. We'll provide resources later on in the book that will give you some guidance on how to learn about deploying your application, but it isn't exactly easy at the moment.

Phoenix provides a lot of things that help you add normally-complex features to your web applications but it will not be the foundation of your application. While it may be strange to read that in a book specifically about Phoenix, the truth is that Phoenix derives its powers from the amazing Elixir programming language.

1.3 The Power of Elixir

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

-- http://elixir-lang.org

Erlang and WhatsApp

There is a great story around Erlang and the popular messaging app WhatsApp which was acquired by Facebook for $19 billion in 2014. When it was acquired, it had 450 million users sending over 70 million Erlang messages per second. If WhatsApp had been built on a programming language other than Erlang, it likely would have had a huge engineering staff. However, since the application was written in Erlang and Erlang provides so many benefits for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

real-time messaging applications, the engineering team at the time of acquisition was made up of only 35 people. Your application may not ever have 450 million users sending 70 million messages per second (well over that these days), why not utilize the technology that would allow you to easily do so if required?

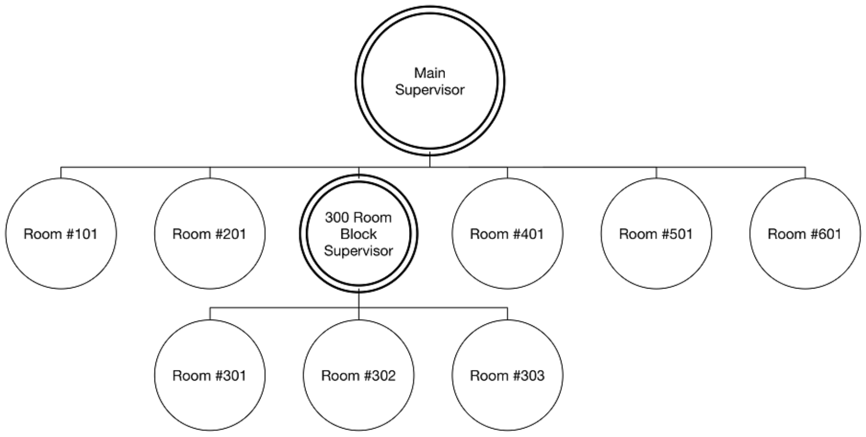
1.3.1 Scalability

One of the amazing things about Elixir is its scalability. Elixir has the idea of processes running on your machine. These aren't processes like you may have thought of them in the past — they aren't threads. Elixir processes are extremely light-weight (the default initial size is 233 words or 0.5kb) and run independent of all the other running processes. We will be building a Phoenix application through the course of this book and it will not be surprising to find that thousands of concurrent processes will be running on your machine in order to allow our application to run. These processes are FAST, independent of each other, have unshared memory, and can easily send and receive messages to other processes on distributed nodes (different machines potentially in different parts of the world).

1.3.2 Supervision trees

Elixir applications (including Phoenix apps and our own application) have Supervisors and Workers. The Supervisors monitor the Workers and ensure that if they go down for one reason or another, they are started right back up. This provides an amazing backbone of fault-tolerance without much work on our part. We can configure how we'd like our Supervisors to handle their Workers if something *does* happen to them including shutting down and restarting any sibling workers. Beyond that, Supervisors can supervise other Supervisors! The supervision tree can get pretty large and complex but this complexity does not necessarily mean our application is harder to comprehend.

Figure 1.5. Supervisors and workers

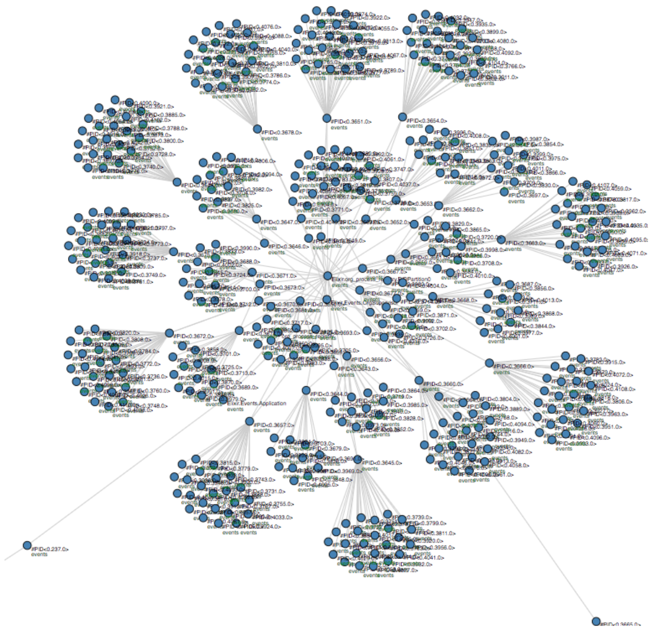


©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Figures 2, 3, and 4 are true visualizations of a real-life application developed by a friend of mine. This application is constantly running in the background to find scheduling conflicts for an organization's physical assets like rooms, buildings, projectors, chairs, and the like. Each supervisor handles a group of rooms. When a request comes in, each supervisor gets a request for it to ask each of the rooms it supervises whether there are any scheduling conflicts. If any of the children processes has any trouble and crashes for some reason, the supervisor notices and starts it right back up.

These figures can give you an idea of how many processes can be spawned during the running of a moderately complex application.

Figure 1.6. Each circle in this diagram is a real Elixir Process. Supervisors can supervise other Supervisors, which in turn supervise Workers.



©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Figure 1.7. The Process tree can continue to grow as your application runs.

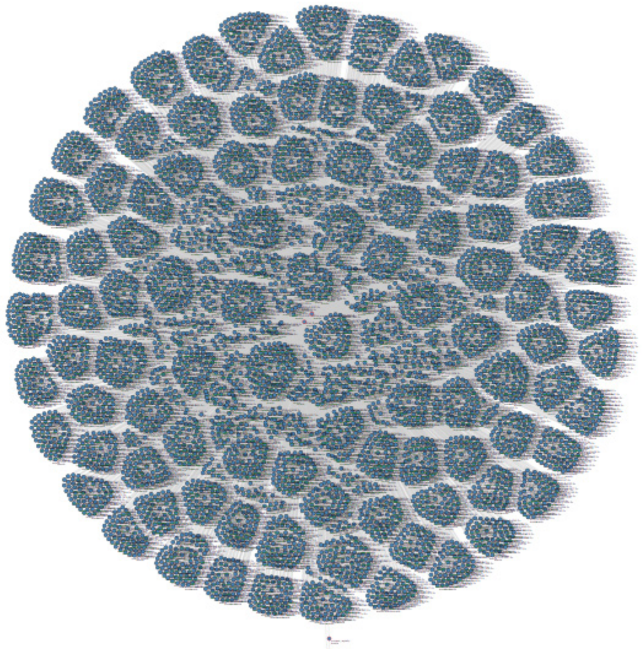
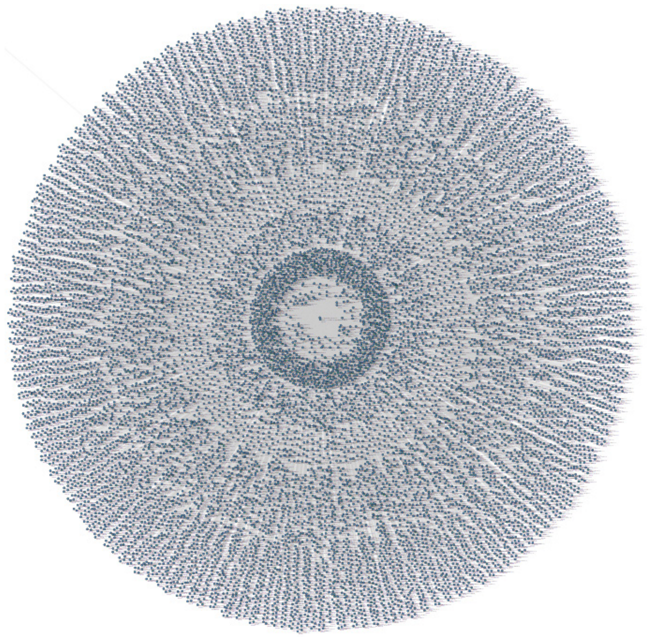


Figure 1.8. Soon your application can spawn thousands of Workers, each isolated from the other.



©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

The benefits of these Processes and Supervisors are many, but one in particular is that it makes created distributed systems much easier than previously possible with other languages. You can deploy your code to multiple servers in multiple areas of the world (or in the same room) and have them communicate with each other. Need more power? You can bring up another server node and your previous nodes can communicate with the new node with minimal configuration, passing off some of the required work as it sees fit.

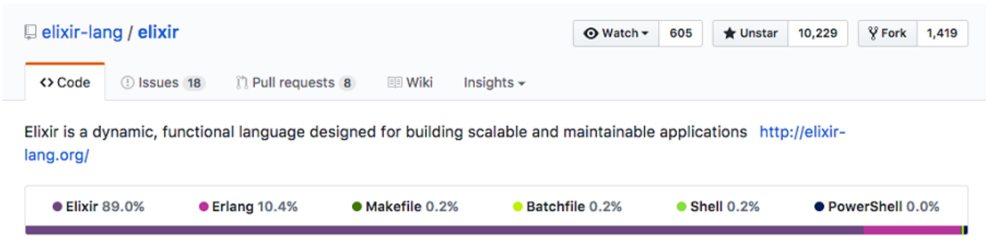
1.3.3 Erlang VM

These particular features are available to us because the backbone of Elixir is Erlang. Erlang has been around for decades now, silently and dependently providing services to many of the things you use on a daily basis. Erlang was created with the telecommunications industry in mind. And what does a telecommunication service require? High fault tolerance, distributability, live-updating of software — things that a modern-day web application should also be able to rely on. Elixir runs on top of the Erlang VM which is rock-solid and has been for decades. You can even run Erlang functions and use Erlang modules directly in your Elixir applications! While this book will not go into the details of Erlang and its VM, there are plenty of resources available that cover Erlang.

1.3.4 Macro and metaprogramming support

While Elixir runs on the Erlang VM, it is itself written in Elixir. How is that possible? It is another one of the great things about the Elixir language — macros. Elixir is extensible via built-in support for macros which allow anyone to define their own language features. You can build your own little language or DSL within Elixir itself. As Figure 5 illustrates, 89% of the Elixir codebase is itself Elixir—the rest is made up of Erlang and various shell files.

Figure 1.9. Breakdown of language types in Elixir's source code (as of June 2017)



Elixir is a dynamic, functional language designed for building scalable and maintainable applications http://elixir-lang.org/

● Elixir 89.0% ● Erlang 10.4% ● Makefile 0.2% ● Batchfile 0.2% ● Shell 0.2% ● PowerShell 0.0%

1.3.5 OTP

Finally, one feature of Elixir that cannot go unwritten is its ability to utilize OTP. OTP stands for "Open Telecom Platform" but that name isn't very relevant anymore so you'll almost always just see OTP. These are a set of functions, modules, and standards that make using things such as Supervisors and Workers,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

and the fault-tolerance that goes along with that possible. There are concepts such as GenServer, GenStage, concurrency, distributed computation, load balancing, and worker pools to just name a few. It's been light-heartedly said that any complex problem you are attempting to solve has likely already been solved in OTP.

> *If half of Erlang's greatness comes from its concurrency and distribution and the other half comes from its error handling capabilities, then the OTP framework is the third half of it.*
>
> -- http://learnyousomeerlang.com/what-is-otp

So with all that greatness backing Phoenix, what if you don't know Elixir? That's OK. Chapter 2 will take you through the basics of the Elixir language and, if you want to go even deeper, recommend some resources you can check out.

1.4 Functional vs OO Programming

To this point in history, I think it is fair to say that the majority of web applications are built on object-oriented programming languages. Ruby, Python, Java, and C# are all object-oriented and are widely used in web application development. In recent releases, even historically procedural languages like PHP and Perl have been gaining object-oriented features. And while the number of developers productively using these languages remains high, there seems to be a recent shift away from the notion that object-oriented programming is the best way forward and towards more functional programming languages (such as Elixir, Haskell, Lisp, Clojure). Even though most object-oriented programming languages are not prevented from writing applications in a functional way, having a language that is strictly functional from the beginning has its advantages.

1.4.1 Overview of Functional Programming

Functional programming is all about data transformation. Mutable (changeable) data and changing state are avoided (and in some cases impossible). The reality is a function which is passed the same parameters multiple times will return the same answer every time. In purely functional code, nothing outside of that function will be considered when the return value is computed. One of the results of this is that return values of functions need to be captured in order to be used again — there is no object holding onto its state.

Let's suppose that we want to do some calculations regarding a race car and keep track of things like what kind of tires it has, it's speed, acceleration, etc. We could group that information into an Elixir module that will not only defined the structure of the data we desire, but also be a central place to hold all the functions regarding a race car.

Let's take a look in Listing 1.4 at what a race car module might look like in Elixir.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Don’t worry too much about the syntax — we’ll cover that later. This may not be the best example of using the features of the language to their fullest potential, but it will give you an idea of the code syntax and structure.

Listing 1.1. An example RaceCar module in Elixir

```
defmodule RaceCar do
  defstruct [:tires, :power, :acceleration, :speed]

  def accelerate(%RaceCar{speed: speed, amount: amount} = racecar) do
    Map.put(racecar, :speed, speed + amount)
  end
end

ferrari_tires = [
  %Tire{location: :front_right, kind: :racing},
  %Tire{location: :front_left, kind: :racing},
  %Tire{location: :back_right, kind: :racing},
  %Tire{location: :back_left, kind: :racing}
]
ferrari = %RaceCar{tires: ferrari_tires,
                   power: %Engine{model: "FR223"},
                   acceleration: 60,
                   speed: 0}
ferrari.speed
# => 0
RaceCar.accelerate(ferrari)
# => 60
ferrari.speed
# => 0
new_ferrari = RaceCar.accelerate(ferrari)
new_ferrari.speed
# => 60
ferrari.speed
# => 0
```

Look carefully — our `ferrari` variable does not get changed when we pass it to the `RaceCar.accelerate/1`[2] function. We can run that line 1,000 times and we’d get the same return value every time: an updated structure of the `ferrari` with a new speed. But remember, our *original* `ferrari` doesn’t change in memory. We have to capture that return value in order to use it later. What this kind of programming provides is an elimination of side effects. We can run our function at any time and be confident that it will always return the same value for the same input — regardless of time of day, "global state", what order functions were called in, etc.

[2] This syntax means the `accelerate` function inside the `RaceCar` module that has argument arity (the number of function arguments) of 1

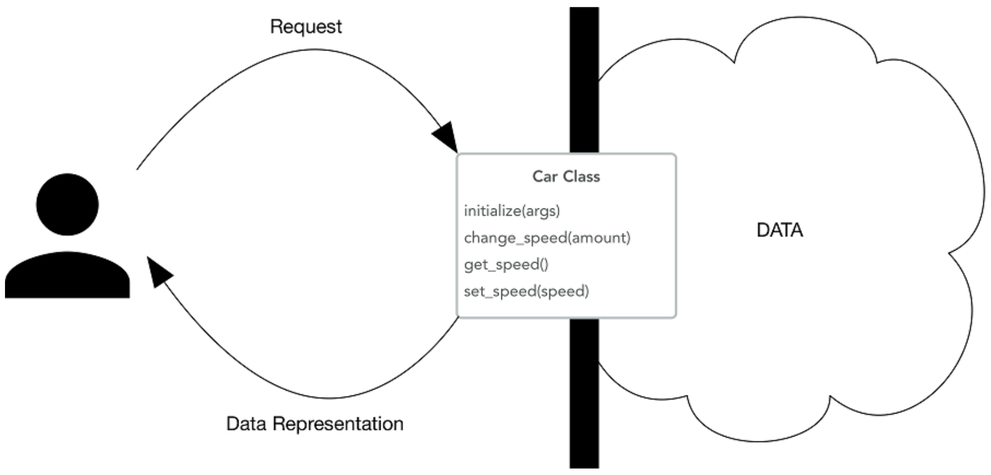
©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

-- https://en.wikipedia.org/wiki/Functional_programming#Coding_styles

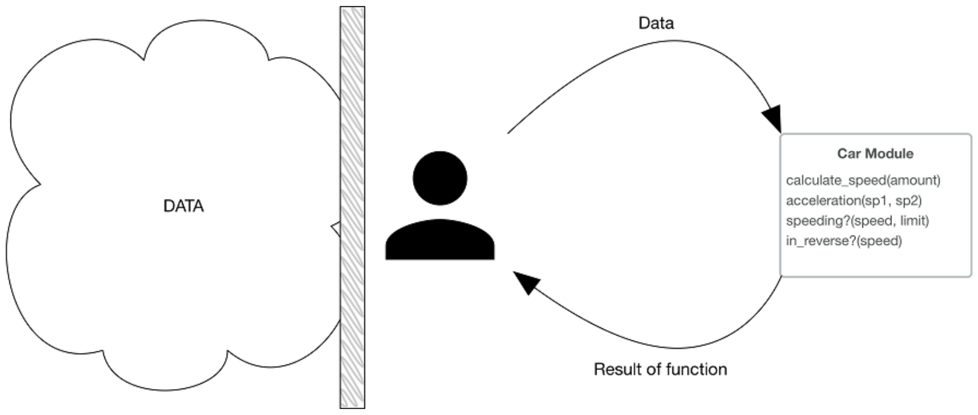
In simple terms, object-oriented programming involves objects holding onto their state (data) and providing methods to the outside world that allow them to access, modify, or even delete that data. In contrast, Elixir modules don't store state (data) on their own—they simply provide functions that operate on the data passed to them and return that data back to the caller. Figure 6 gives a simple visualization of how Objects handle data vs how functional modules handle data.

Figure 1.10. Object-oriented programming generally involves an object that keeps track of its own data (state). Methods are used to manipulate that internal state and retrieve it for other uses.



©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

Figure 1.11. Functional programming generally involves having a module that contains functions that act together for a purpose. The module holds no data (state) but only returns the result of the function based on the data the user gives it.



In the world of object oriented programming, an object's internal data or state is open for many different things to mutate it during an operation. Because of the often dynamic nature of object-oriented languages, a third-party package could hook into your codebase and modify your data without you even knowing. In contrast, when working with Phoenix, the data flows in one single line from the user making the request to the requested page being displayed. And for each step the data takes, you specify how that data is being passed along. The side effects are therefore minimal (and most of the time eliminated entirely).

1.5 Summary

In this chapter you learned:

- Dynamic websites can be a tricky problem to solve. Web frameworks cam along to help make the solution easy and quick to implement. Phoenix delivers on this promise and can get your dynamic (or static) website online with minimal fuss.
- Much of the power of Phoenix comes from the Elixir langauge. Elixir runs on the Erlang virtual machine (VM) so you could say that much of the power of Elixir comes from the Erlang VM.
- There are strong benefits to writing a web application with Phoenix over other alternatives including fault-tolerance, speed, concurrency, real-time communication, distribution, and potential hardware cost savings.
- A few key differences between object-oriented programming in languages such as Ruby vs functional programming in a language like Elixir. While OO languages try to model the domain in real-world object terms that abstract away the code interface from the data that object holds, functional programs treat the data as king. All the code of a program is built around explicitly manipulating the data in an immutable way by passing the result of one function to the input of another.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action

- Some of the areas in which Elixir/Phoenix may not be the best choice for your particular application — like requirements for pure processing speed, number crunching, huge communities, lots of readily-available packages, considering the time it will take to become productive, and the current difficulty of deployment.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/phoenix-in-action