Extracted from:

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

The Pragmatic Bookshelf

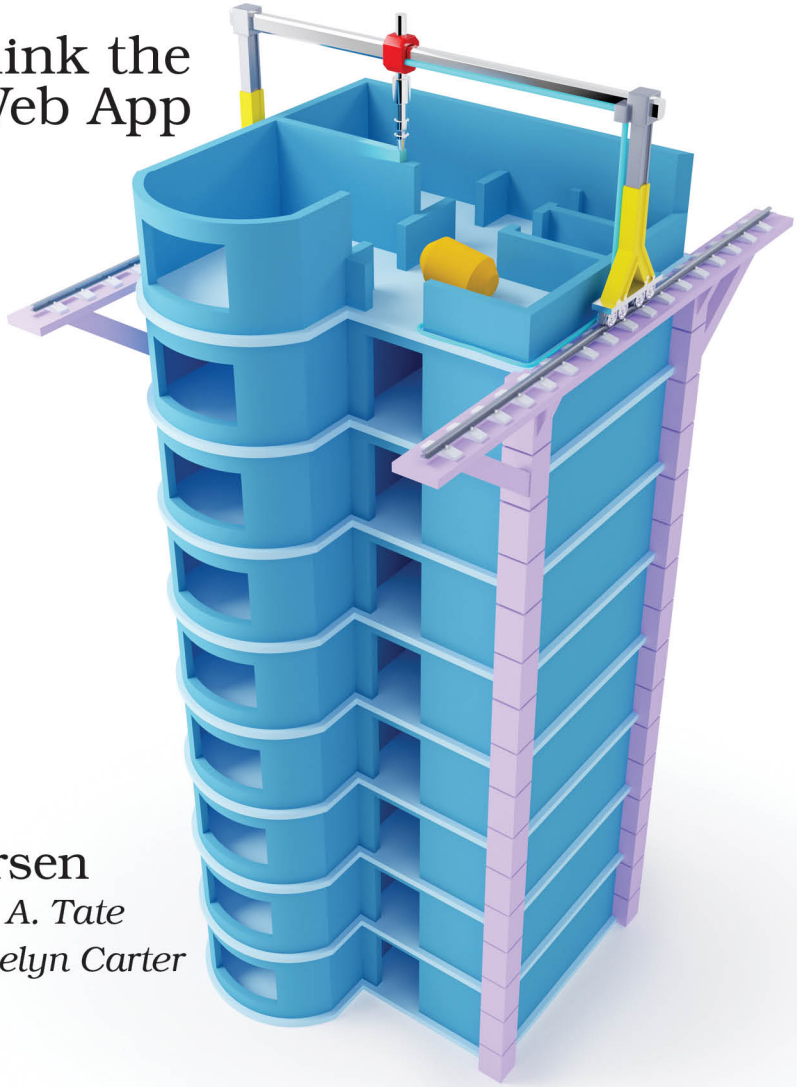
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App



Lance Halvorsen
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

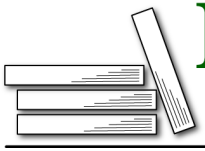
Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

Lance Halvorsen

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-243-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—March 22, 2017

CHAPTER 1

Mapping Our Route

Welcome! We're about to go exploring, and it's going to be a blast. We're going to do what many of us say we love most—play with new languages, experiment with new techniques, and expand our understanding of writing software for the Web. Whenever you go exploring, it's important to have a map, a good idea of where you're headed, and a plan for how you'll get there. That's what this chapter is all about.

Many early client server systems were stateful. Servers kept working state in memory. They passed messages back and forth with their clients over persistent connections. Think of a banking system with a central mainframe and a dedicated terminal for each teller. This worked because the number of clients was small. Having fewer clients limited the system resources necessary to maintain those concurrent connections.

Then Tim Berners-Lee invented a new client server system called the World Wide Web.

The Web is an incredibly successful software platform. It's available almost everywhere on Earth, on virtually any device. As the Web has grown and spread, so has HTTP. HTTP is a stateless protocol, so we think of Web applications as stateless as well. This is an illusion. State is necessary for applications to do anything interesting, but instead of keeping it in memory on the server, we push it off into a database where it awaits the next request.

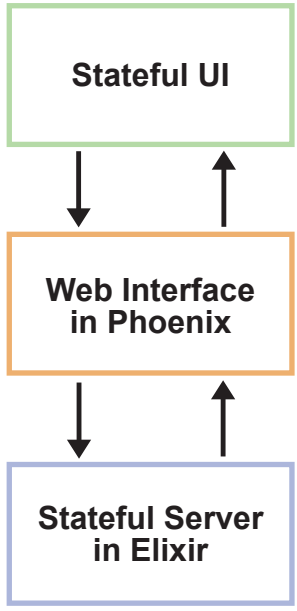
Offloading state to a database provides some real advantages. HTTP based applications need only maintain temporary connections with clients until they send a response, so they require far fewer resources to serve the same number of requests. Most languages can't muster the concurrency necessary to maintain enough persistent connections to be meaningful for a modern Web application.

Going “stateless” has let us scale.

But statelessness comes at a cost. It introduces significant latency as applications need to make one or more trips to the database for the data to prepare a response. It makes the database a scaling bottleneck, and it habituates us to model data for databases rather than for application code.

Elixir offers more than enough concurrency to power stateful servers. Phoenix Channels provide the conduit. A single Phoenix application can maintain persistent Channel connections to hundreds of thousands or even millions of clients simultaneously. Those clients can all broadcast messages to each other, coordinated though the server. While processing those messages, the application remains snappy and responsive. Elixir and Phoenix provide a legitimate alternative to stateless servers capable of handling modern Web traffic.

We’re about to explore this new opportunity with a stateful application written in Elixir and a persistent Phoenix Channel ready to connect it to any frontend application.



We’ll do this by building a game called Islands. It may not be a top download on your favorite gaming platform, but it will be fun to play. Most importantly, you’ll learn a lot by building it. Game developers have always pushed the web to the extreme. They’ve had to approach problems in novel ways to meet their

performance needs. We'll be re-thinking our approach as well, and what we'll learn will help us solve everyday business problems in radically improved ways.

We're going to tackle Islands in distinct parts. We'll start with a stateful game engine written in Elixir, and then we'll layer on a web interface with Phoenix. We'll stop just short of building out a full frontend application—there won't be any new territory for us to cover—but we'll provide examples for you to run and follow along with.

We'll build Islands in a way you might not be used to, so let's get an idea of what lies ahead.

Lay The Foundation With Elixir

In Part One we'll begin by building a stateful game server in pure Elixir. We won't use a database to store the game state, and we'll define our domain elements with native Elixir data structures instead of ORM models. We'll maintain state with Elixir Agents, and we'll wrap those agents up in a GenServer to provide a consistent interface to the game.

We'll bring in a finite state machine to manage state transitions—like switching from one player's turn to the other, and moving from a game in progress to one player winning. We'll also build a supervision tree to take advantage of Elixir's incredible fault tolerance.

Building the game engine solely in Elixir solves a long standing problem in Web development, the tendency for framework code to completely entangle application logic so the two can't be easily separated. Without that separation, it's hard to reuse application logic in other contexts. As we build Islands, we won't even begin to work with the Phoenix framework until our game logic is complete.

By the time we're done with Part One, we'll have a fast, fault tolerant game engine that can spin up a new GenServer for a game almost instantly. We'll be able to reuse it with any interface we want—the Web, a native mobile app, plain text, or whatever else we can think of. If we look at it the right way, the GenServer for each game is really a microservice, or a nanoservice, living right inside the virtual machine.[1]

1. http://blog.plataformatec.com.br/2015/06/elixir-in-times-of-microservices/

Add a Web Interface With Phoenix

In Part Two, we'll generate a new Phoenix application without Ecto, the database layer that ships with Phoenix. We'll bring in our new Islands engine as a dependency and make it part of our new Phoenix application's supervision tree. We'll create a lobby for the game with the standard Phoenix MVC parts—the router, a controller, a view, and some templates.

Then we'll move on to the really exciting part, replacing HTTP's temporary client-server connections with persistent ones via Phoenix channels. Channels provide a conduit for lightning-fast message passing between front end applications, and in our case, a stateful back end server. We'll make good use of Channel naming conventions to allow two players to connect to their own private GenServer running Islands. And we'll be able to run thousands of games simultaneously on a single server. Many languages would struggle to keep persistent connections open for all the players of all current games, but Elixir's incredible concurrency model will make it easy.

As we finish up, we'll have a Web interface to our Islands engine. The main component will be a Phoenix Channel able to connect two players directly to an individual Islands game. We'll customize the JavaScript files that Phoenix provides to get it primed and ready for your favorite frontend framework. When we're done, it'll have much less code and far fewer moving parts than a conventional Web application.

The Game of Islands

Let's talk a little bit about Islands. It's a game for two players, and each player has a board which consists of a grid of one hundred coordinates. The grid is labeled with the letters A through J going down the left side of the board and the numbers 1 through 10 across the top. We name individual coordinates with a letter-number combination—A1, J5, D10, and so on.

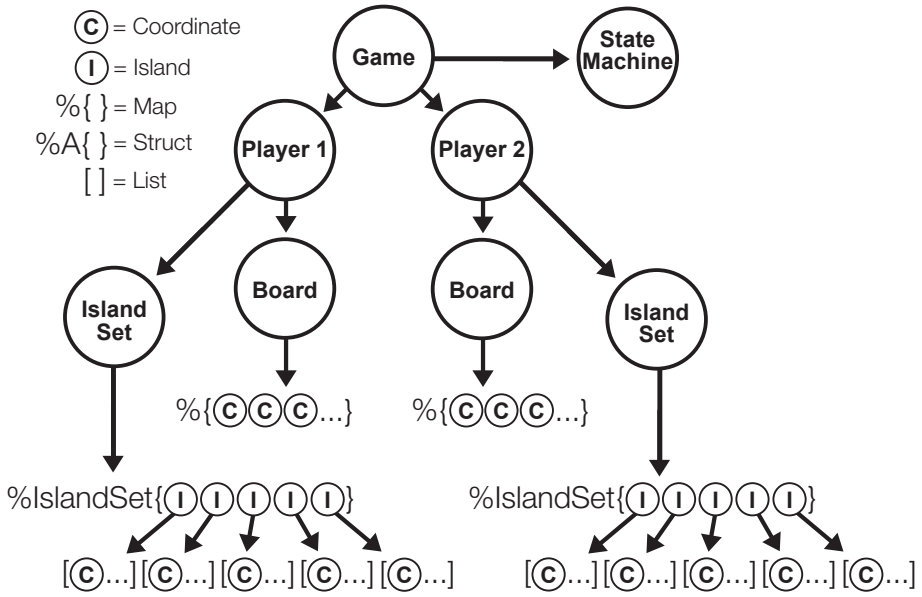
The players cannot see each other's boards.

The players have matching sets of islands of various shapes and sizes which they place on their own boards. The players can move the islands around as much as they like until they say that they are set. After that, the islands must stay where they are for the rest of the game.

Once both players have set their islands, they take turns guessing coordinates on their opponent's board, trying to find the islands. For every correct guess, we plant a palm tree on the island at that coordinate. When all the coordinates for an island have palm trees, the island is forested.

The first player to forest all their opponent's islands is the winner.

That gives us a rich, interesting data structure model across multiple processes. When we're done, we'll have something that looks like this.



Before we get to work, let's make sure we have all of our dependencies installed. For the first part of the book, all we'll need are Elixir and Erlang. For the second part, we'll need to install the Phoenix archive, Node.js, and npm. Have a look at the *the (as yet) unwritten appendix.installation_instructions* for help getting them installed.

We've got a plan! Time to start building.