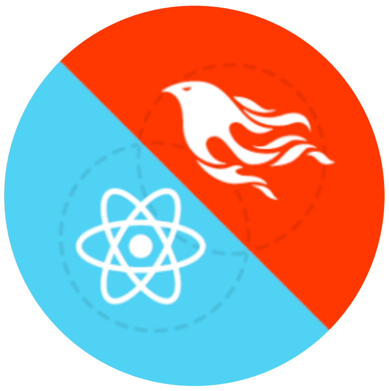
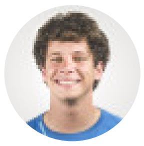
Phoenix and React: A Killer Combo





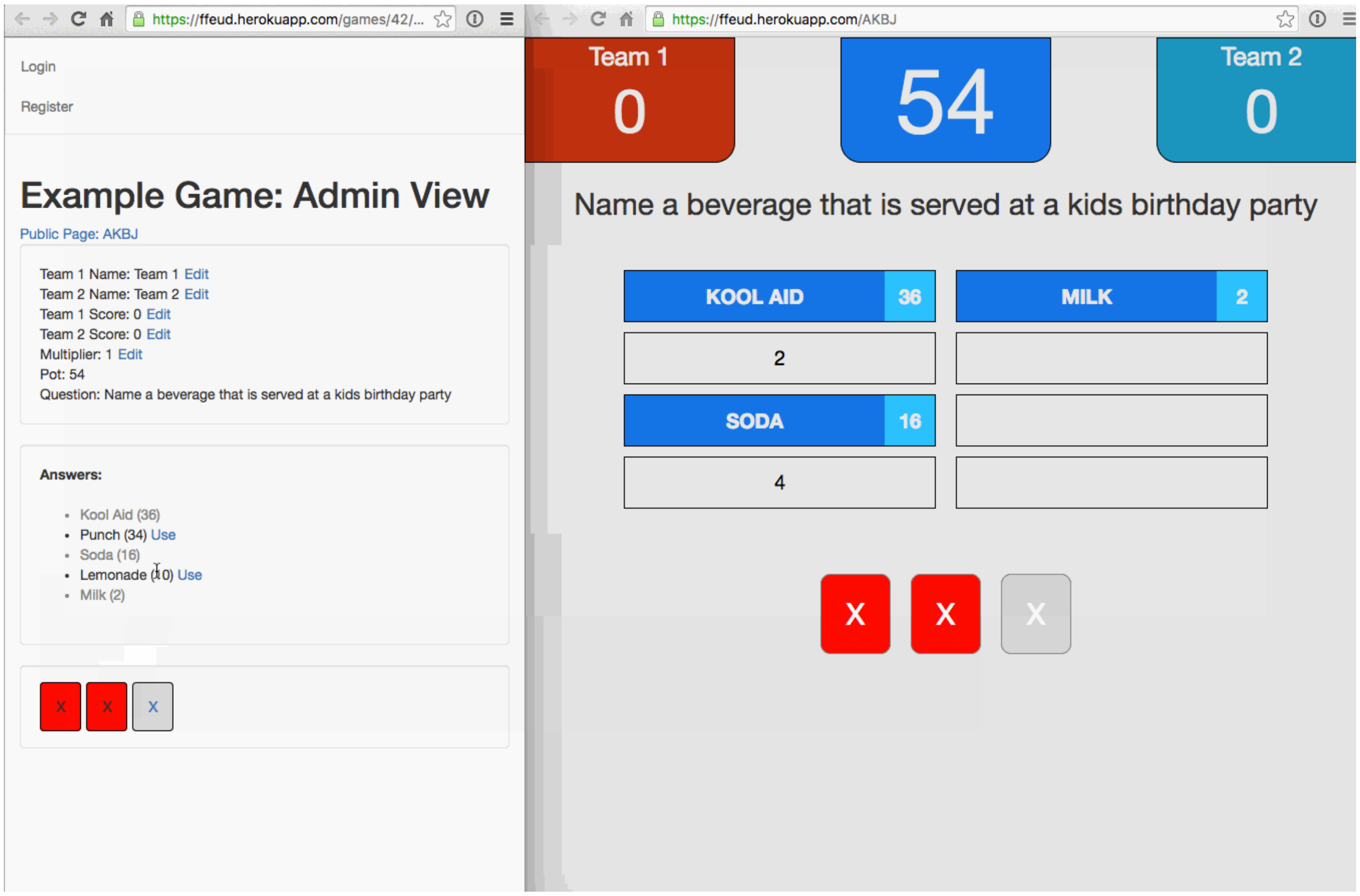
Eli Fatsi, Developer

Posted in **#CODE** , **#FRONT-END ENGINEERING** , **#BACK-END ENGINEERING**
on October 14, 2016

Given the right project, there's nothing sweeter than using Phoenix and React together. For great justice.

We recently built our own clone of Family Feud, that popular game show you used to watch when you were skipping school because you were "sick".

You can check out the app yourself! Play the preview game by pulling up the following links: Admin View & Public View. It looks like this:


https://ffeud.herokuapp.com/games/42/...
Login
Register
https://ffeud.herokuapp.com/AKBJ
Team 1
0
54
Team 2
0
KOOL AID 36
MILK 2
2
SODA 16
4
X
X
X
X
X
X

Example Game: Admin View

Public Page: AKBJ

Team 1 Name: Team 1 Edit
Team 2 Name: Team 2 Edit
Team 1 Score: 0 Edit
Team 2 Score: 0 Edit
Multiplier: 1 Edit
Pot: 54
Question: Name a beverage that is served at a kids birthday party

Answers:

- Kool Aid (36)
- Punch (34) Use
- Soda (16)
- Lemonade (10) Use
- Milk (2)

Name a beverage that is served at a kids birthday party

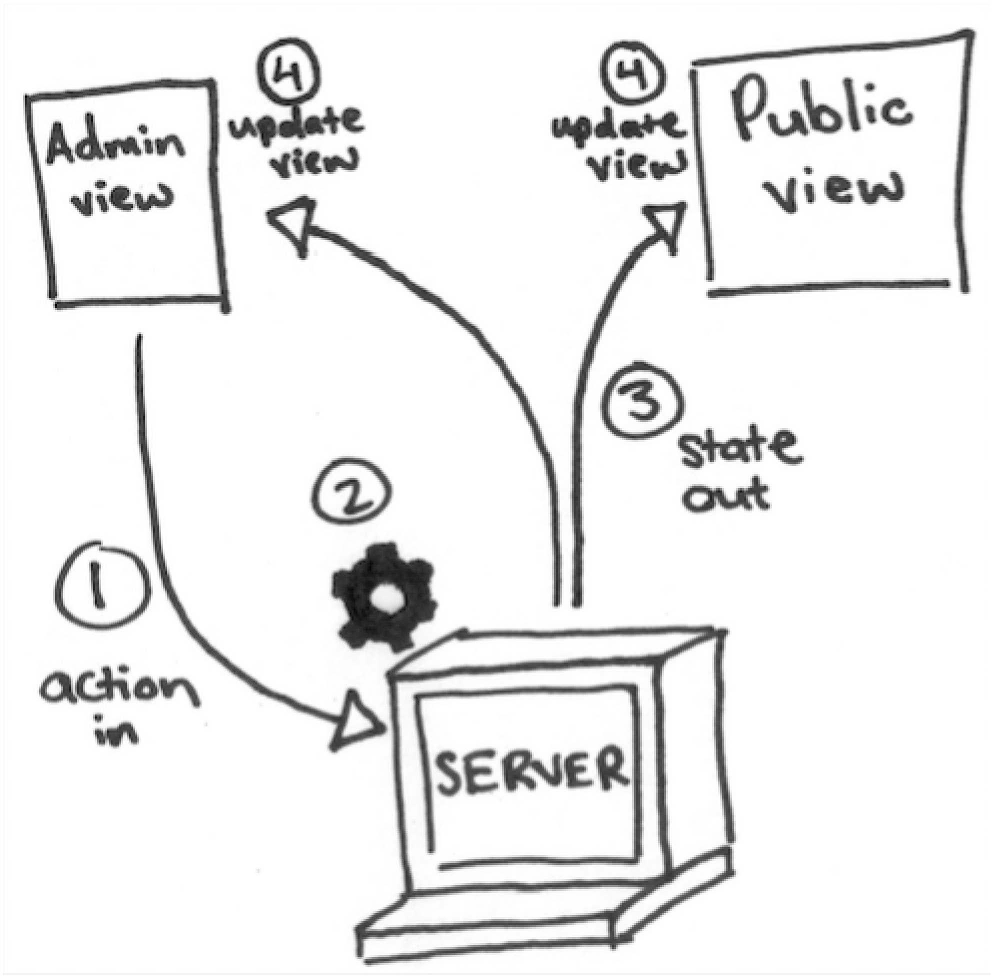
We reached for Phoenix to make use of it's easy-to-use websockets, and React for it's state-driven rendering capabilities. The result was glorious and I'd like to tell you about it.

If you'd like to peek at the code, it's all available here - https://github.com/vigetlabs/ffeud.

The Basic Setup

I'll be talking about the side of the app that handles the actual playing of a Family Feud game. There's the more boring other side that handles user login, game creation, and other managerial tasks built with Phoenix that I won't go into.

Here's an image of the real game screen if you don't quite recall what it looks like. There is a set of answers, that have points, which add up, and go to one of two teams. Unless you strike out first, in which case the other team gets a shot at stealing the round's points! So, lots of rules to implement, and what does all of that functionality look like in diagram form?



Pretty straightforward right? We built two different pages to handle game play - a **public page** that shows what you typically see in a Family Feud view, and an **admin page** that lets the game master control the flow of the game. The flow of changes follows this strategy:

- The admin client is authorized to **send down "actions"** to the server
- The server **handles the inbound action**, updating the database as necessary
- The server then **broadcasts the new state** to all connected clients (admin and public)
- The clients receive their new state and **update their view accordingly**

The Nerdy Details

In my opinion, the way that Phoenix and React can be used together to **manage and display your application's state** is what makes the combination so great. If you think about it, any application can be thought of as responsible for handling two tasks:

1. Determining what is the current state
2. Displaying that state in a meaningful way

Interestingly enough, this is a distinction I've also focused on (and blogged about) when designing firmware to run on microcontrollers.

I've found that when dealing with rich display logic, the more you can keep these two tasks isolated, the better. Fun fact - Phoenix and React together make this separation of concerns trivial. BUT I DIGRESS!! Back to what you probably came here to read about.

How Phoenix Helped - Websockets

As the diagram above detailed, an "action" from the admin page was sent to the server, some stuff happened, and then the state was immediately broadcasted out to all clients. **Communication between the client and the server is fast, flexible, and easy**, and we have websockets to thank for that.

Another major benefit is that **all of the state manipulation is handled on the server**. The server, in this case, being a highly concurrent application with direct access to application's database. Hard to think of a better place for handling state changes.

Here's a quick look at the wiring required to get this communication in place:

On the javascript side, we initialize the connection, load the current state on successful connection, and listen for inbound messages. Here is a simplified version:

```
import {Socket} from "phoenix"

let socket  = new Socket("/socket")
let channel = socket.channel("game:" + gameId)

channel.on("state", payload => {
  console.log(payload)
})

channel.join().receive("ok", resp => {
  channel.push("load_state")
})
```

As you can see, we're initializing a websocket connection on a specific topic (based on the game ID), and console logging out the payload of a `state` message whenever it's received over the connection.

On the server, we laid the groundwork for accepting new connections, as well as handling inbound actions:

```
def join("game:" <> game_id, _params, socket) do
  socket = assign(socket, :game, Repo.get(Game, game_id))
  {:ok, socket}
end

def handle_in("load_state", _params, socket) do
  push socket, "state", GameState.generate(socket.assigns[:game])
  {:noreply, socket}
end

def handle_in("act", params = %{"action" => action}, socket) do
  game = socket.assigns[:game]
  ActionHandler.handle(action, game, params)

  broadcast! socket, "state", GameState.generate(game)
  {:noreply, socket}
end
```

How React Helped - render()

React is great at **taking a given state, and rendering it for you**. And when that state changes, React is also great at handling the DOM updates needed to reflect the changes. It does this incredibly well in fact.

So with React taking care of turning our state into an accurate visual representation, **all we need to worry about is telling React what the state is**. Luckily enough, we've built our websocket communication layer to do just that! Here's what our js code actually looks like:

```
let AdminApp = React.createClass({
  componentDidMount() {
    this.props.channel.on("state", payload => {
      this.setState(payload)
    })

    this.props.channel.join().receive("ok", resp => {
      this.props.channel.push("load_state")
    })
  },

  render() {
    ...
  }
})
```

Aside from the fact that this code is now operating withinin a React component, there is only one change from our boilerplate setup pasted above:

```
this.setState(payload)
```

That's it! Adding that one line means that whenever Phoenix broadcasts the updated state, our React app accepts it as it's own and immediately re-renders the view to match.

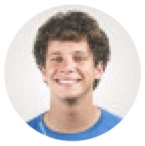
It. Is. So. Easy.

In Closing, or, Why The Combo Is Awesome

We've essentially **torn down the barrier** that typically exists between your front end client side app and your back end REST API. No longer is there a need to develop a comprehensive JSON API on the server as well as client side code to interact with the said API through isolated HTTP requests.

Instead, it's as if we've found a way for two technologies, each masters of their own craft, to finally speak each other's language. Working with Phoenix and React together feels like working on one cohesive project, not two that both know what an HTTP request is. Before I make any more semi-applicable analogies, I'll leave you with this haiku.

Actions, they come in.
The state, it goes right back out.
Smooth. Flowing. Data.



Eli uses his mathematics degree from Carnegie Mellon to blur the lines between the digital and physical worlds. He codes for Shure, Volunteers of America, and other clients from our Boulder, CO, office.

More posts by Eli

ADD A COMMENT

READ NEXT

Build Your Own Slack App and Bot

ALL CODE ARTICLES

Let’s get to work.

Have an unsolvable problem or audacious idea? We’d love to hear about it.

CONTACT US

hello@viget.com : 703.891.0670

Washington DC Metro / Durham, NC / Boulder, CO

The Viget Newsletter
A curated periodical featuring thoughts, opinions, and tools for building a better digital world.

CHECK IT OUT →