The Elixir Of Life

Rushi Shah

9 April 2016

> Haskell makes the easy things hard
> and the hard things easy.

I'm a big fan of functional programming, in case you haven't noticed. Specifically, I'm a big fan of Haskell, but I've been branching out a bit recently. I started branching out because I realized that sometimes Haskell makes the easy things too hard. Don't get me wrong, it makes the hard things really easy, but it also makes the easy things hard. For example, I had the time of my life whipping up a simple AI for Tic-Tac-Toe in Haskell but when it came time to take input and output I had to ask Stack Overflow how to do it.

I also have not been a fan of Haskell's package manager or Haskell's popular back-end web frameworks. They just seem too heavy for my taste. But I had an idea for a quick REST API I really wanted to create and I'm loathe to do so in your run-of-the-mill JavaScript Framework Of the Week™. So I decided to look into something new. Lucky for me, I discovered the elixir of life that reinvigorated functional programming for me.

1 Functional Sinatra?

When I made Github Chart API, I had originally used Ruby on Rails, but eventually switched to Sinatra (which I talked about in this blog post) because all I needed was to define a bit of back-end code for each route. I wanted something similar for this project, but I wanted a functional programming language.

2 The Idea

I guess I should explain what my idea was. It was partially inspired from the Github Chart API I made earlier. I wanted an API that would let you just GET request your latest commit message. I imagine people maybe having a banner or something on their personal website where they would display it with some pretty styling. This would be especially helpful for people who commit early and often and with interesting commit message. It also might encourage more clear commit messages, which is always a plus.

3 Early Setbacks

Reading this blog post inspired me to actually get started on my idea (I had been thinking about it for a while but hadn't gotten around to it). That post led me to this one about building a JSON REST API in Haskell. I started by trying to install Scotty with cabal, but it failed to install after taking a ton of time to install dependencies (I'm pretty sure those dependencies are still clogging up my computer). So after getting frustrated with that, I tried to install Servant instead, but I met similar errors. Haskell prides itself on beautiful, concise code and I wasn't seeing any of that with web programming in Haskell. Everything just felt like a hack together for people who were REALLY motivated to program web servers in Haskell.

4 Enter Elixir

That's not to say functional programming languages are bad as back-end languages. In fact, Erlang is probably the most powerful language for stuff like that out there. But perhaps people don't find Erlang very pleasant to write (I've never actually used it myself). So Elixir was created, which is (from what I've seen) basically Erlang that looks like Ruby. Elixir is great because it's fault-tolerant, concurrent, and really fast. Like **really** fast. Did I need a concurrent language that could scale to billions and billions of users? Of course not. But it was a pet-project anyways so I didn't see any reason to experiment with new technologies and branch out a bit.

5 Burn down the Phoenix

So I was a big fan of Elixir right off the bat, mainly because it was a compiled language that makes heavy use of the REPL. Those are two of my favorite parts of Haskell, so I was happy. Unfortunately, it isn't a strongly typed language, otherwise I'm pretty sure Elixir would be considered perfect in my mind. As I got started, I found the web framework Phoenix for Elixir. But I remembered what had happened when I made Github Chart API: I had tried to hammer in a nail with a jack-hammer. Yeah sure, it worked, but I really didn't need all that power, it was just overkill. Similarly, I didn't need to use Phoenix. Instead, I just needed a simple server, which is exactly what Plug was built for. Plus, I found this wonderful blog post that outlined exactly how to start my project. To define a route, all I needed to do was write this:

```
get "/hello" do
  send_resp(conn, 200, "<h1>world</h1>")
end
```

(Looks mysteriously like Ruby and Sinatra, right?!)

6 Getting the most recent commit

So now for the actual logic of the program. How do you get the most recent commit from the Github API? If there was an endpoint for that my program would be useless. But unfortunately, there is not. I completely overestimated the problem. My initial solutions were atrocious: I would get every single repo from a user, then get every single commit for each of those repos, then sort the repos by their most recent commit, and then take the head of that list and return the resulting most recent commit. Think about that for a moment. If you had n repos, I would need to make n+1 API requests. That would

- Be extremely slow for the user
- Kill my rate limit

Then I realized that I was dumb and that Github provided parameters that would let me sort a user's repos by how recently they were pushed to. Then, I could do something similar with the commits for that repo and I for a user with n repos I would still only need to make 2 API requests. The code I ended up with is:

```
def getLatestCommit(username) do
  client = Tentacat.Client.new(%{access_token: Commit.Keys.github_key})
  [repo | _] = Tentacat.Repositories.list_users(username, client, [sort: "pushed"])
  repoName = Map.get repo, "name"
  [commit | _] = Tentacat.Commits.list(username, repoName, client)
  commit
end
```

Elegant, am I right?

7 Plain Text and JSON support

So originally I had the following route:

```
# Send a plain-text response of just the message
get "/:name" do
  message = Map.get (Map.get getLatestCommit(name), "commit"), "message"
  conn
  |> send_resp(200, message)
  |> halt
end
```

But I realized that people would probably want the meta-information for their commit information too. For example, I would personally make it a link with the href being a link to the commit on github and the value of the tag the commit message. Luckily, Elixir's data strucure maps directly to JSON with a package called Poison, which I first heard about from this blog post. So my second route could be:

```
# Send JSON response
get "/json/:name" do
  conn
  |> put_resp_content_type("application/json")
  |> send_resp(200, Poison.encode!(getLatestCommit(name)))
  |> halt
end
```

8 Heroku

This buildpack made deploying to heroku painless, so I could offer a hosted version of the app at comet.rshah.org.

9 Cross Origin Requests (CORs)

After I had done so, though, I ran into a problem. I wanted people to be able to make a client-side request for their commit message and append it to their website even on a static site with Javascript. After I deployed to Heroku and tried to test it out though, I realized that I needed to enable Cross Origin Requests. Doing so was a synch, really just boiled down to one line: `plug PlugCors`

10 Using it

So now it works! An example (unfortunately using JQuery) of a static site that embeds your latest Github commit is just:

```
<html>
  <head>
    <script src="jquery.js"></script>
  </head>
  <body>
    <a id="commit"></a>
    <script>
      $.getJSON('http://comet.rshah.org/json/2016rshah', function(data) {
        $("#commit").text(data.commit.message);
        $("#commit").attr("href", data.html_url)$
      });
    </script>
  </body>
</html>
```

11 Conclusion

So that's simple enough to use, right? I had a ton of fun making it, and I'm glad that the easy things don't always have to be hard in functional programming, as long as you use the right tools.

If you think Comet is neat, I would really appreciate it if you would check out the project on github (and star it if you're real!). If you decide to embed your own commit somewhere using Comet, let me know on Twitter or something! And to do so all you have to do is GET request

```
comet.rshah.org/json/<YOUR-GITHUB-USERNAME>
```