Whispr

Your medical info is *yours*. Period.

$8.3 Trillion

The amount of money passing through the U.S. healthcare system annually. Yet, maintaining your information, and passing it to various providers, is a byzantine process of calling up hospitals, rifling through papers, and remembering minute details.

Whispr wants to change that.

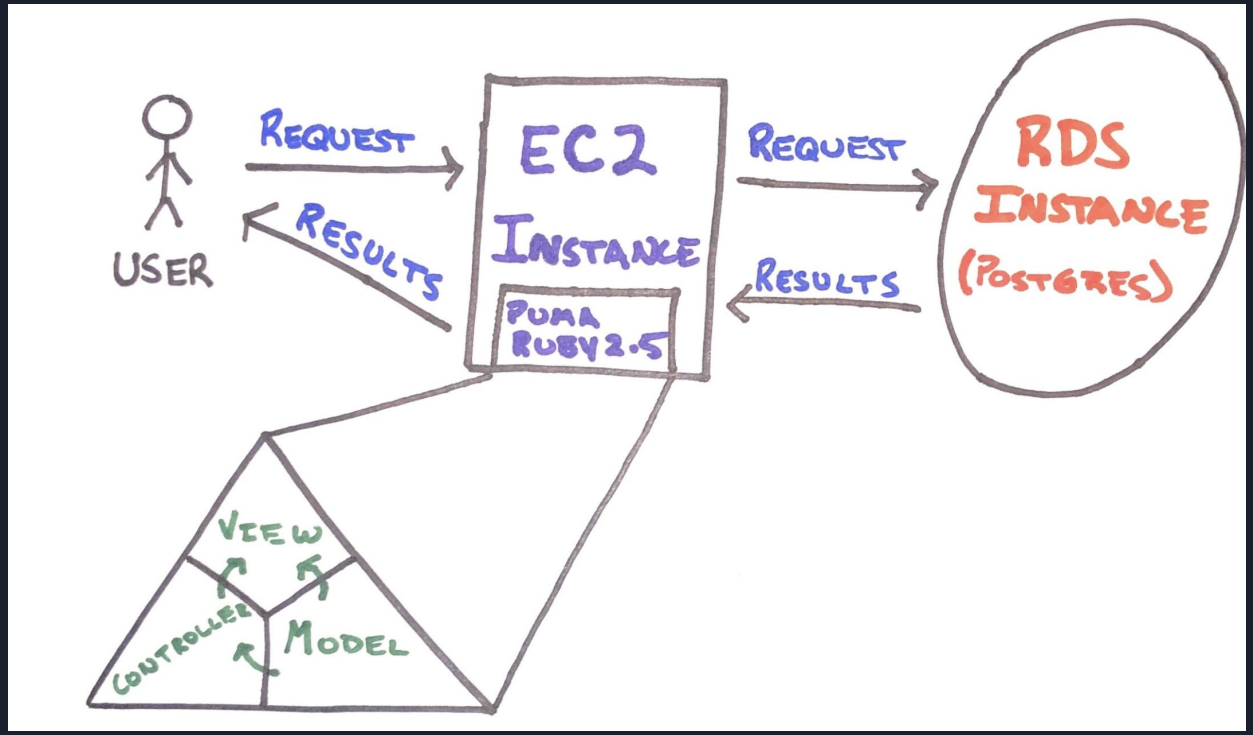
What is Whispr?

- 100% online, 100% available, 100% easy service to store and share medical info
- Securely store your data, and access it any time, anywhere
- Plugs into our network, allowing you to easily link with healthcare providers
  - Removes the hassle from sharing info across hospitals and insurance companies
  - One click:
    - Send your info to UCLA Health
    - Revoke USC Medicine's access to it
- From a provider perspective
  - Easy access to the healthcare data of users
  - Ability to search for and request data from users
  - Future goal of aggregated statistics filtered by demographic

How'd We Build Whispr?

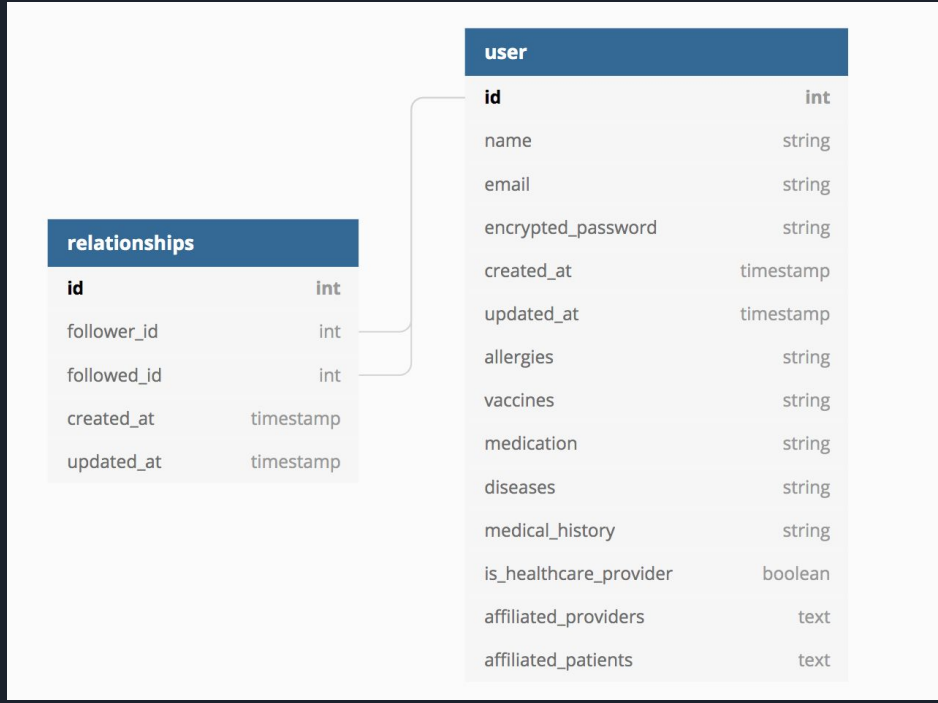
- Utilized Ruby on Rails, thereby pulling in the Model-View-Controller architecture
- Specifically, we used 3 main controllers:
  - an application controller to handle general tasks
  - a users controller to manage the actions tied to both patients and providers
  - a relationships controller to manage the relationships between providers and patients
- Each page received its own custom view

Broad Architecture Diagram



Whispr's Models

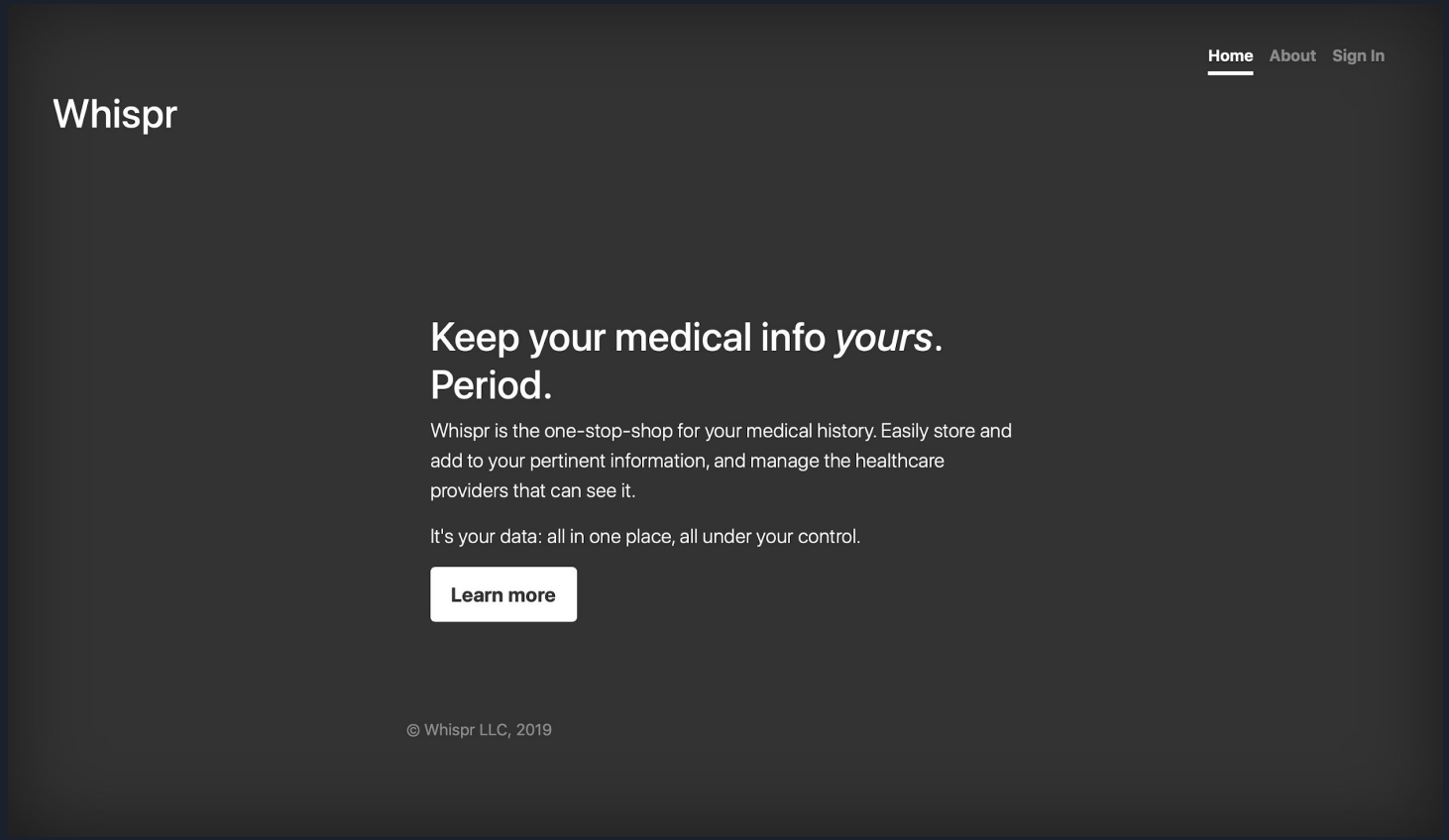
- Gear towards scalability
  - We worked to create as simple a set of models as possible
- We wound up with the two tables, as seen on the right
- As can be seen, we had a relationships table, and a users table -- that's it!



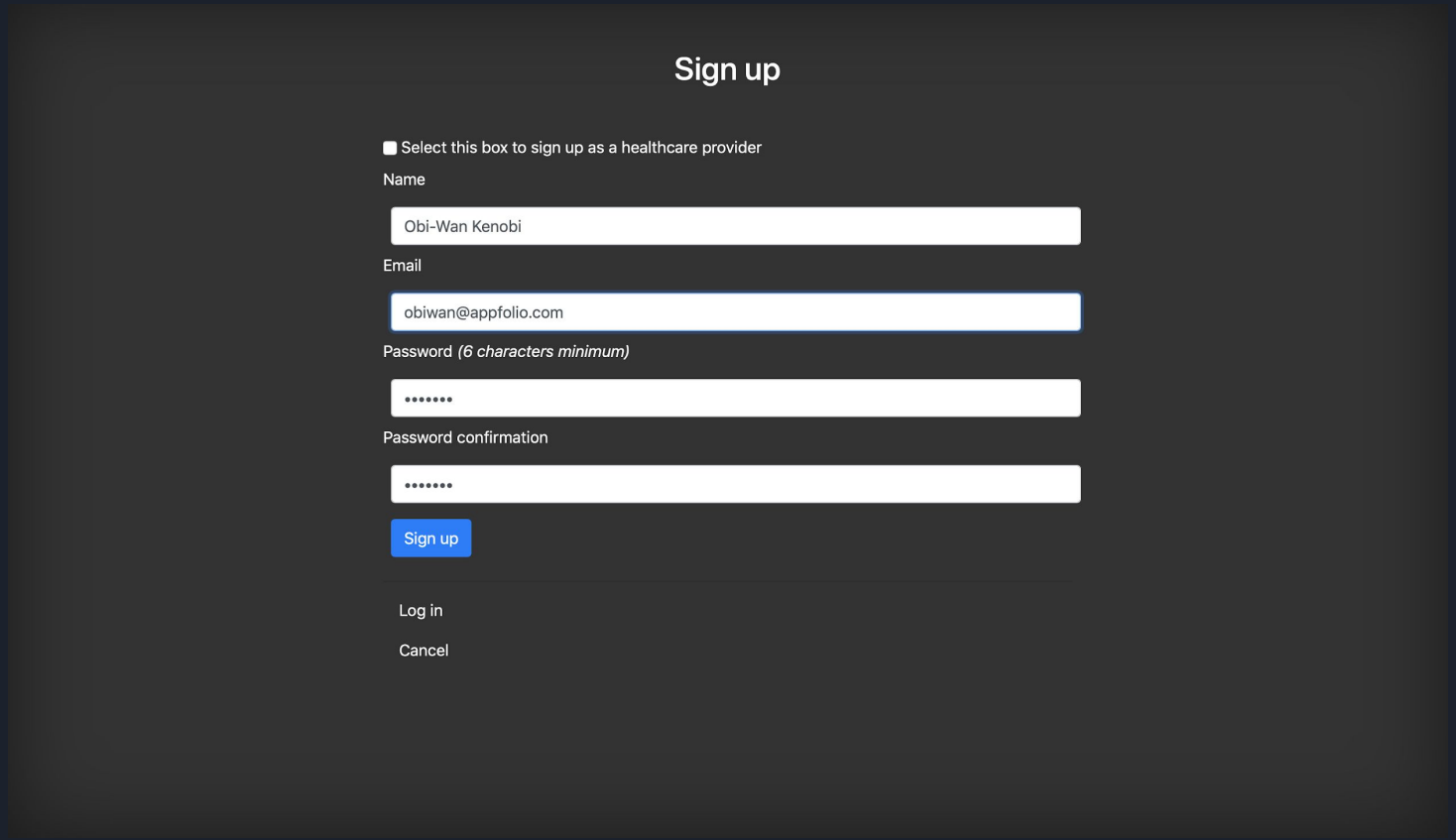
Why So Simple?

- From our perspective, we felt this was the best solution
  - Easy to manage, simplistic design
  - Easier to create relationships within one table than across many
    - In the same vein as the Ruby on Rails tutorials
- Coalesces all necessary information into two tables
  - We felt the benefit of utilizing a dedicated relationships table was not worth the added complexity
- Providers only need a subset of the information a patient needs
  - Providers will also be far fewer in number
  - Adding a dedicated provider table decreases digestibility
  - Would also have necessitated increased SQL complexity

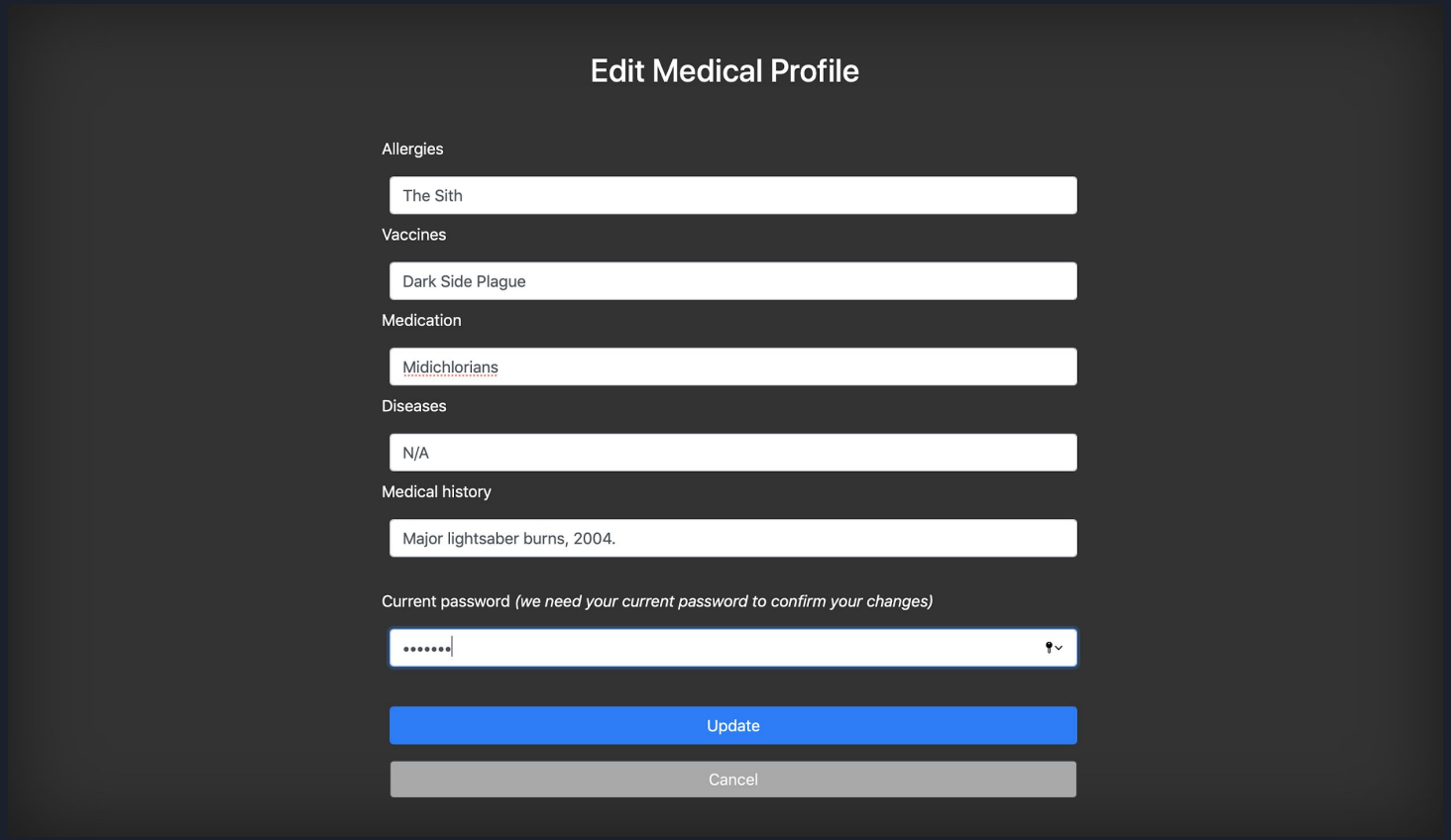
Whispr in Action!



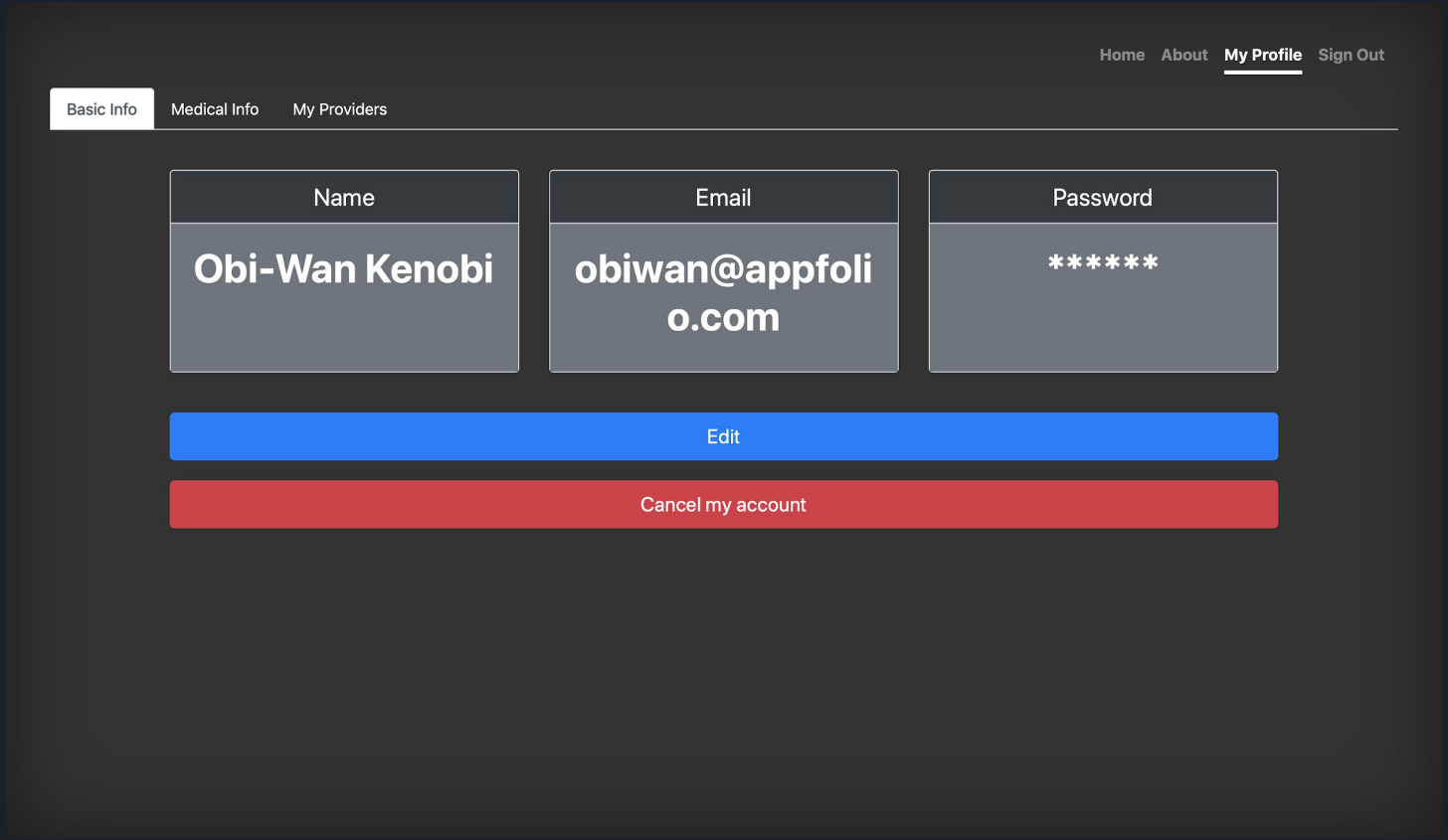
Whispr's landing page - note the dark theme (intended to imply the ability to keep a secret)



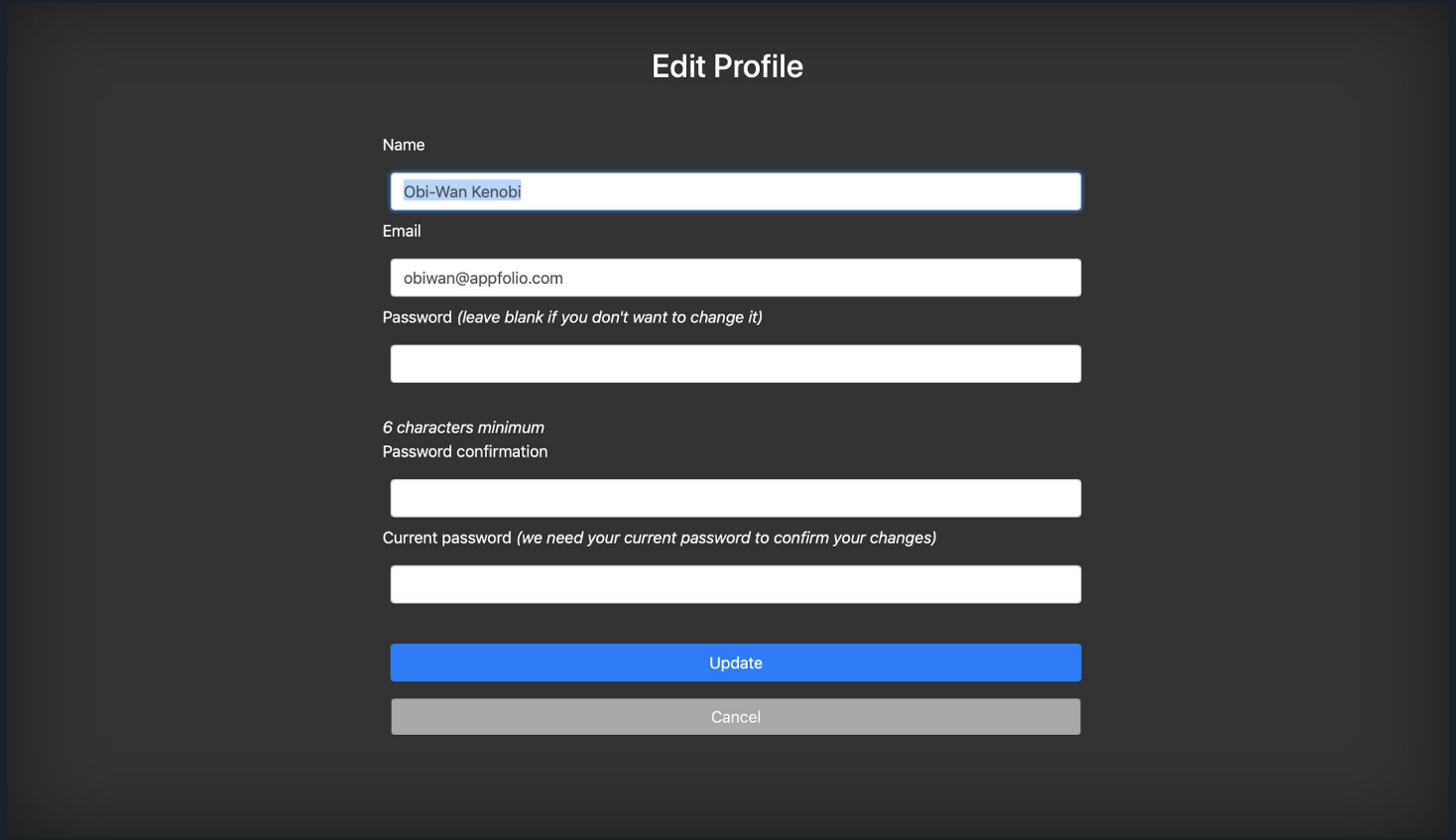
Sign-up, providing basic user information



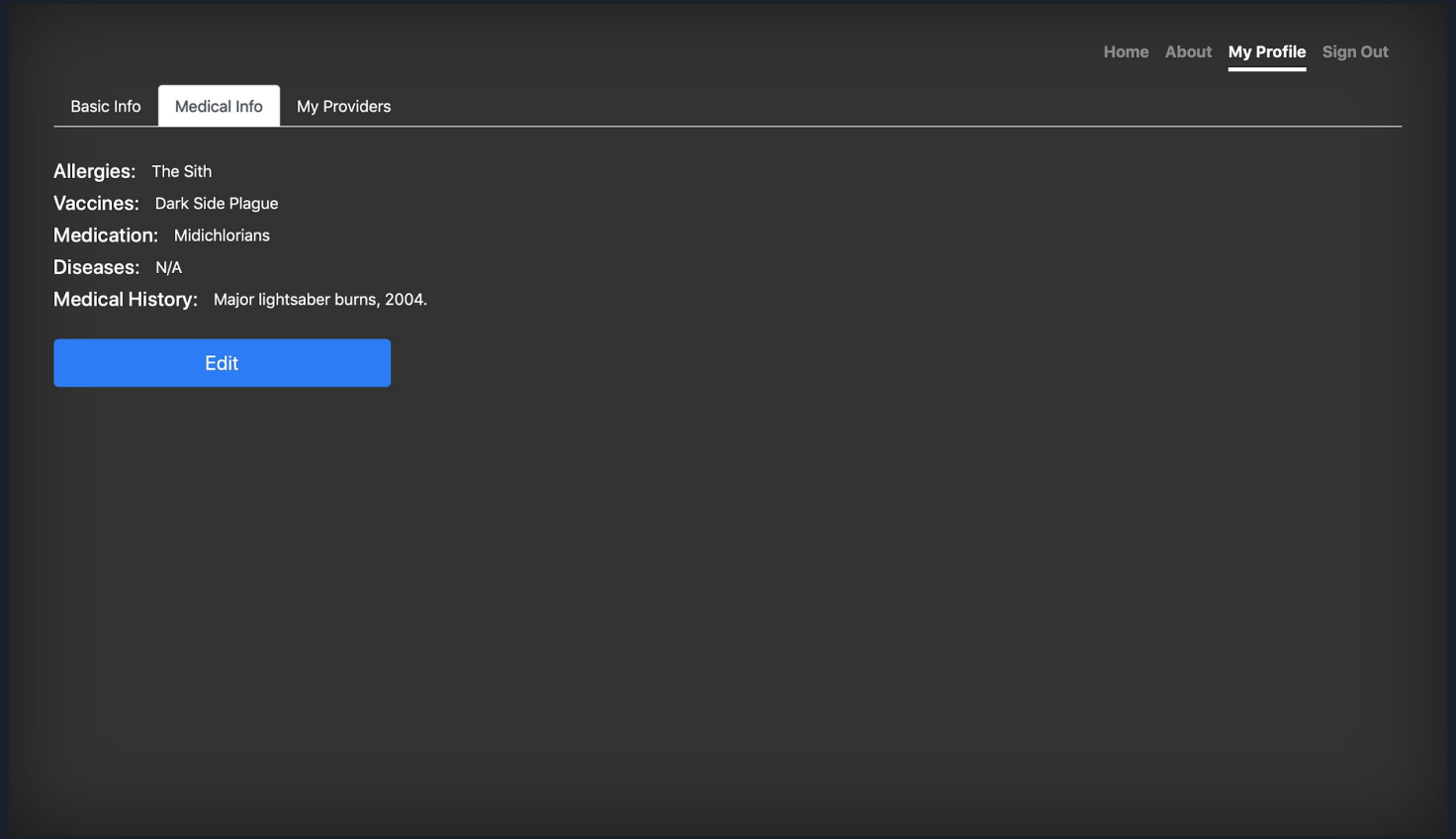
Immediately after sign-up, you're redirected to the edit medical profile screen



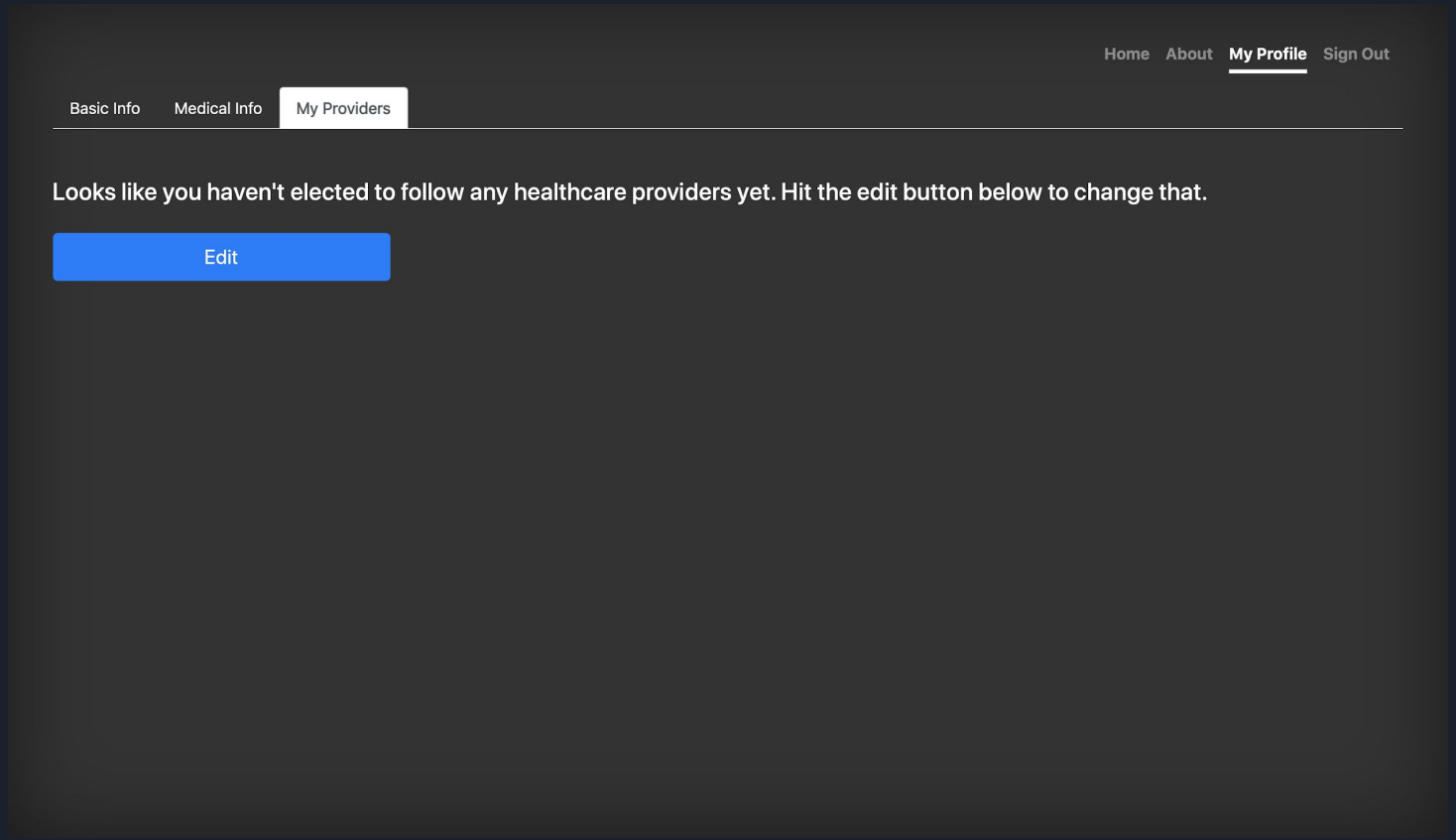
The profile dashboard - from here you can see the basics of your account, edit them, and cancel your account, if you desire



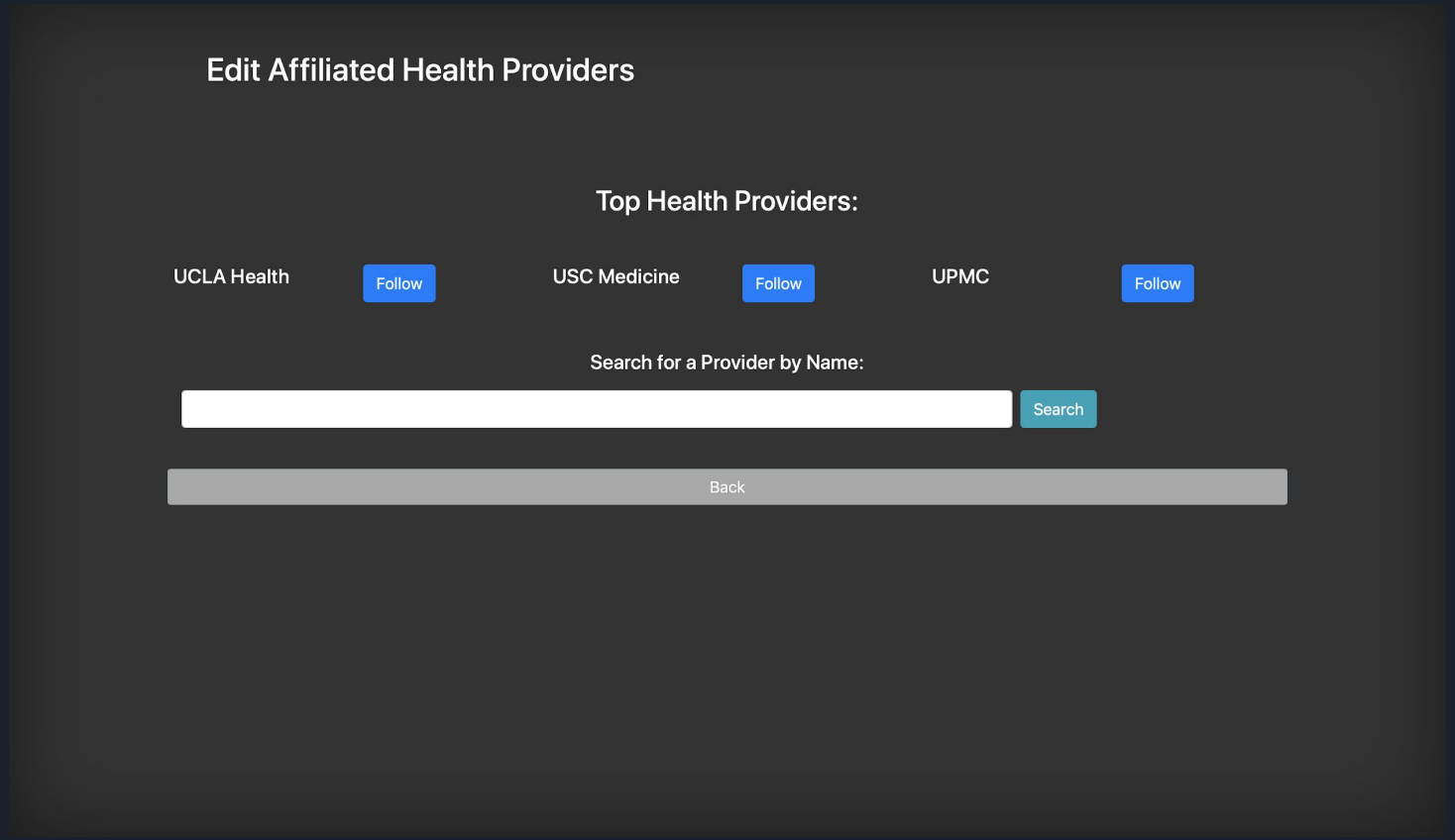
If you choose to edit your profile, you are brought to the Edit Profile screen



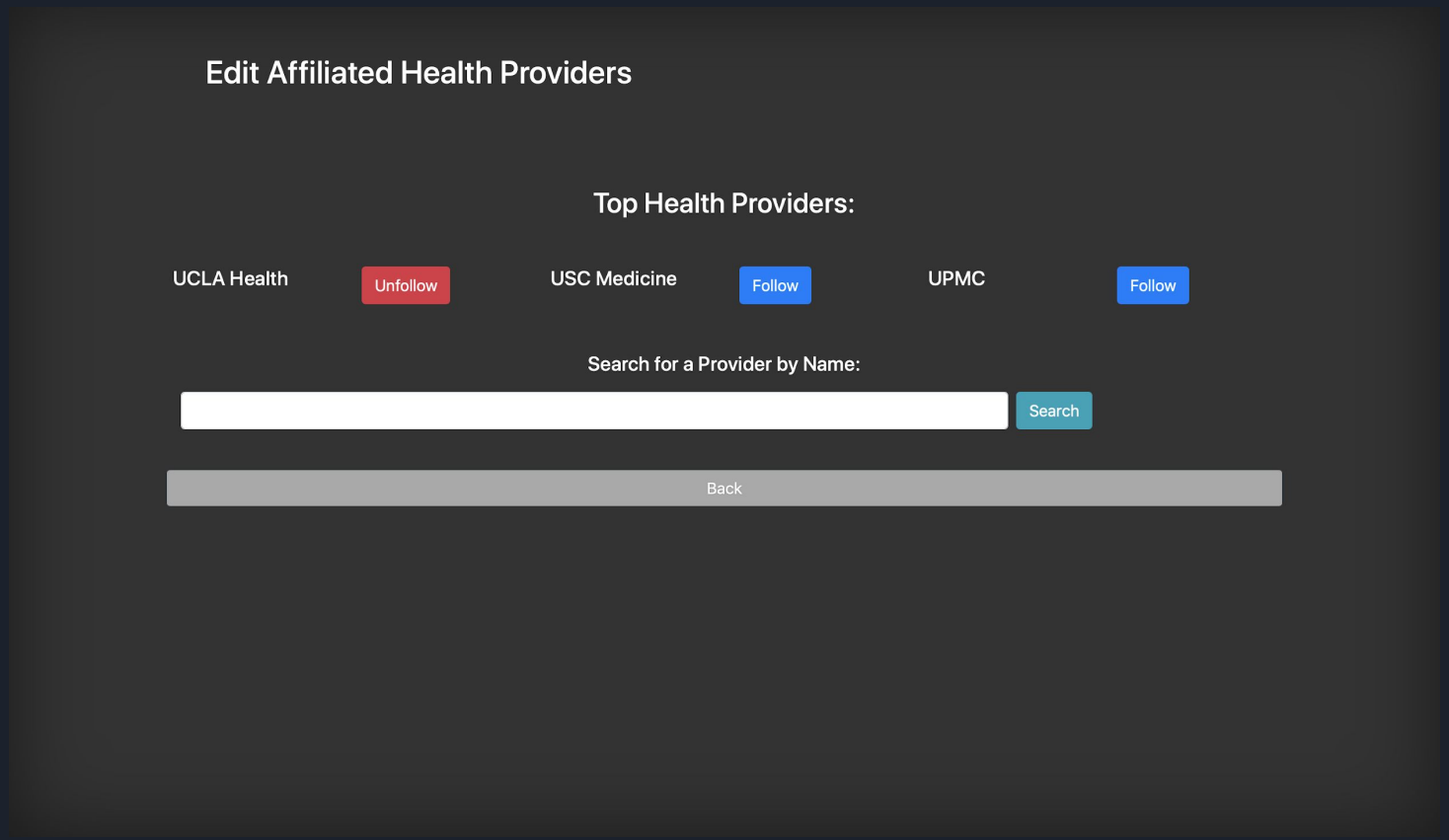
The Medical Info Tab - here you can see and edit your medical profile (which will bring you to the same page you saw immediately after signup)



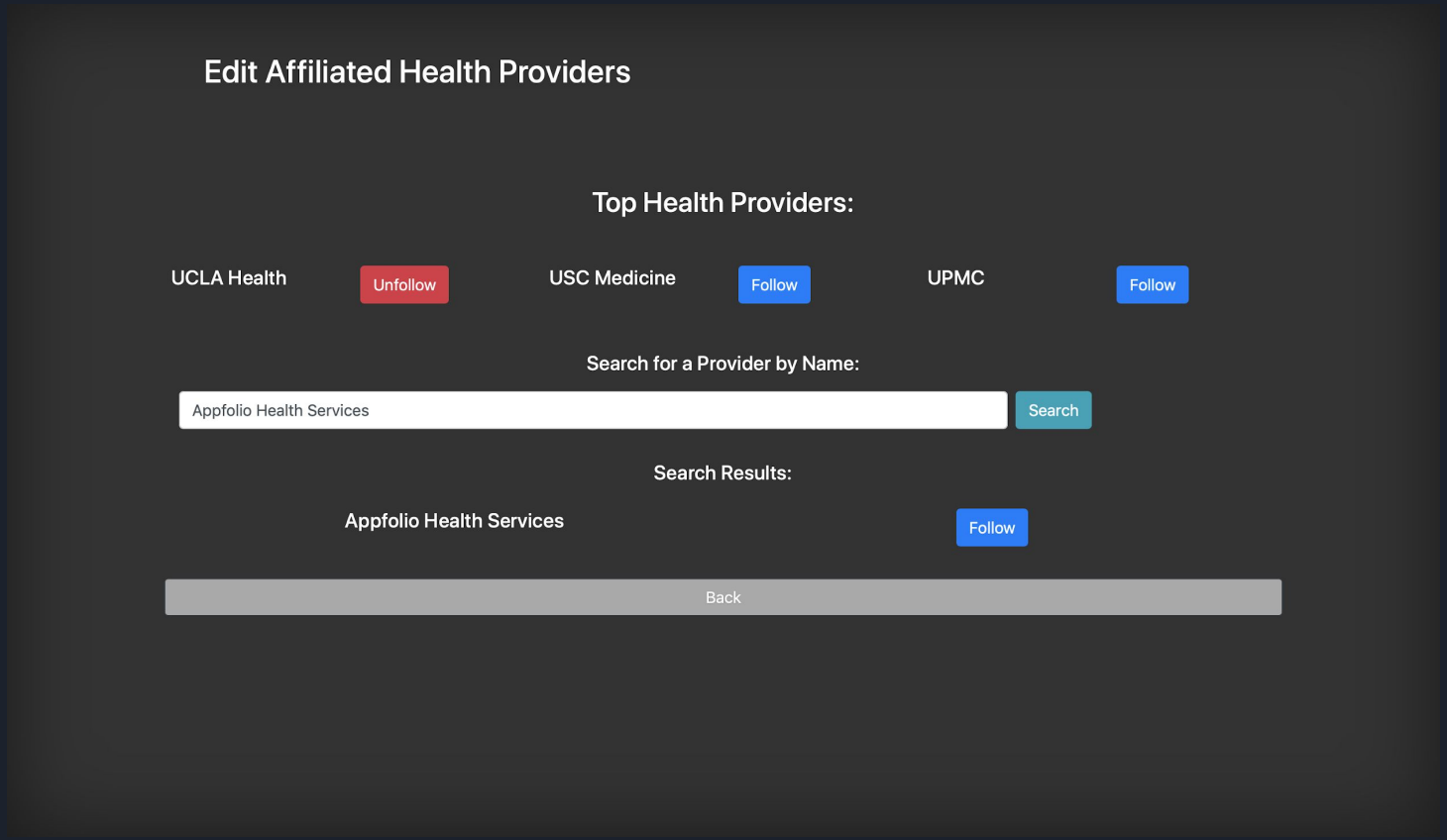
If no providers are interested in you, and you have not followed any providers, “My Providers” will look like this



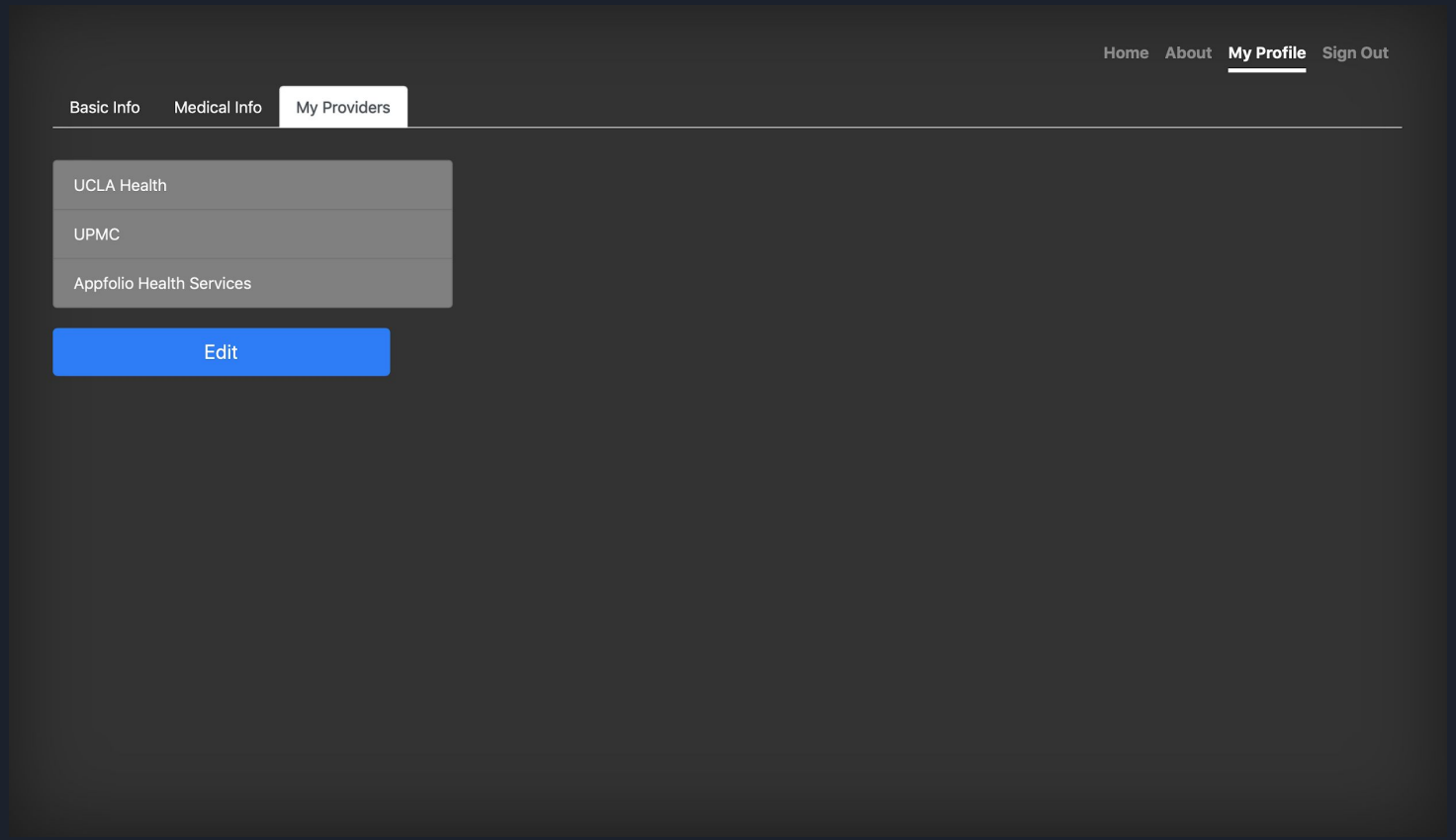
Clicking the edit button, you can easily find the top healthcare providers and...


Edit Affiliated Health Providers
Top Health Providers:
UCLA Health
Unfollow
USC Medicine
Follow
UPMC
Follow
Search for a Provider by Name:
Search
Back

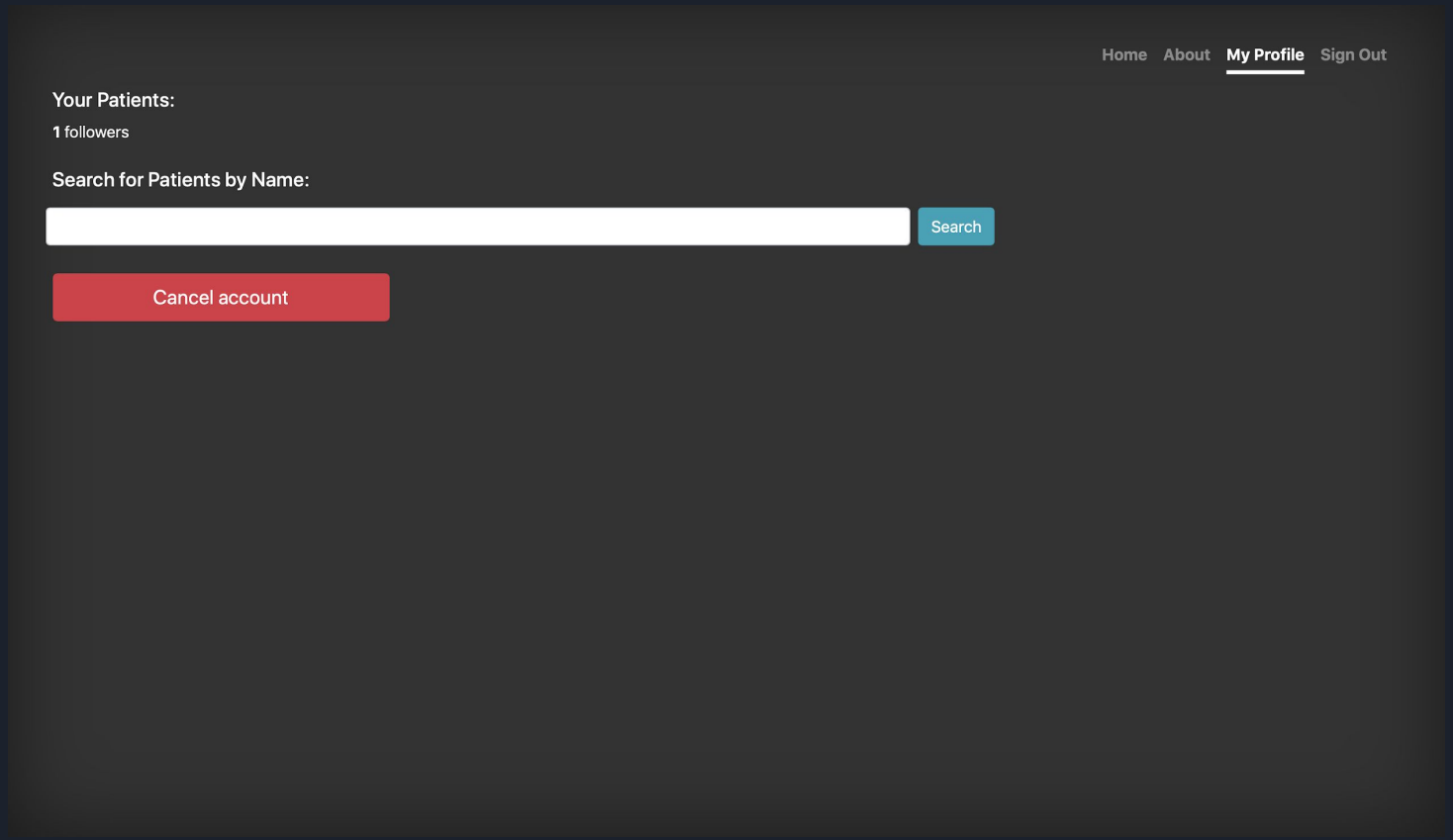
...follow one



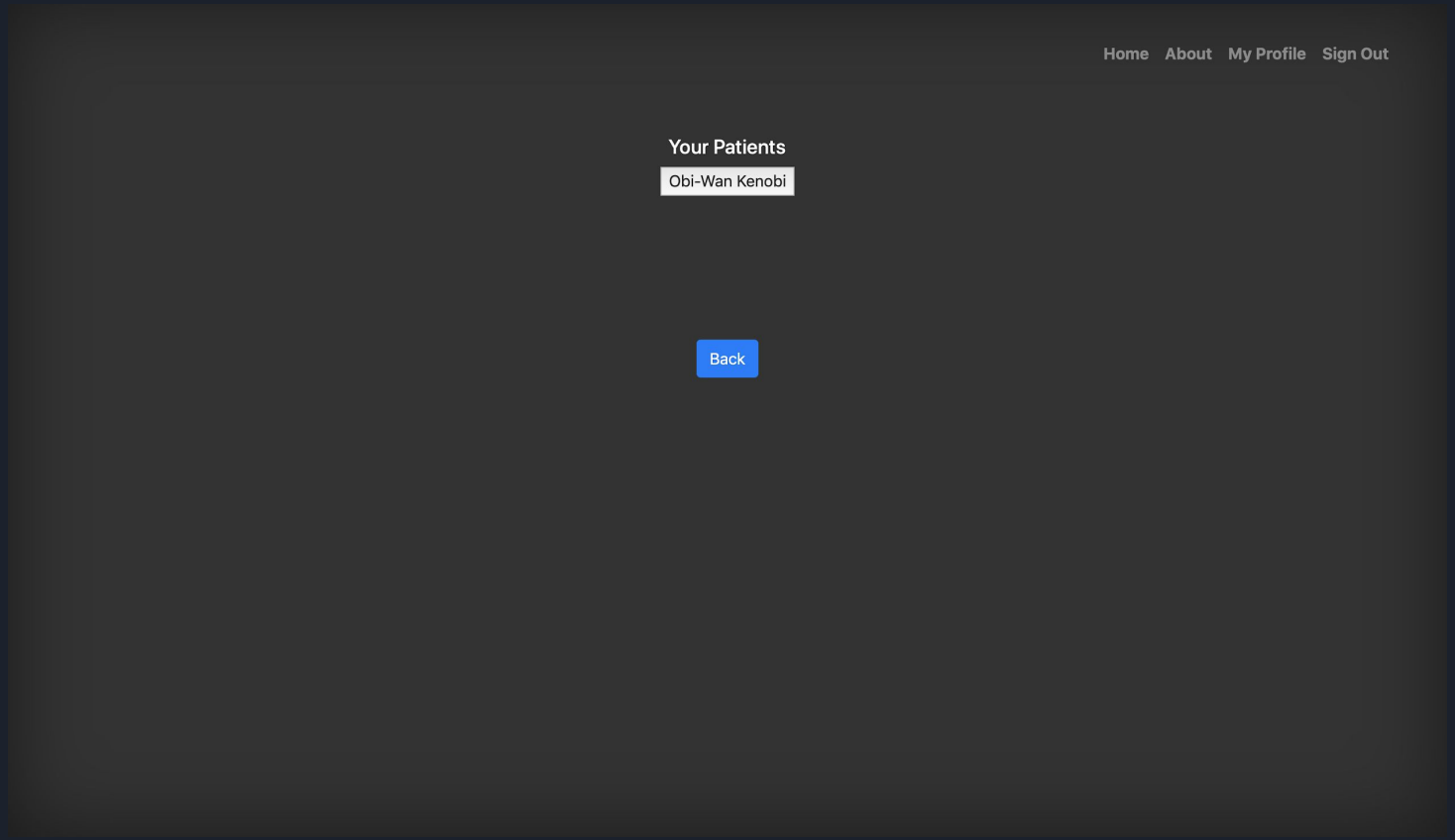
Or search for the one you want, and follow that one!



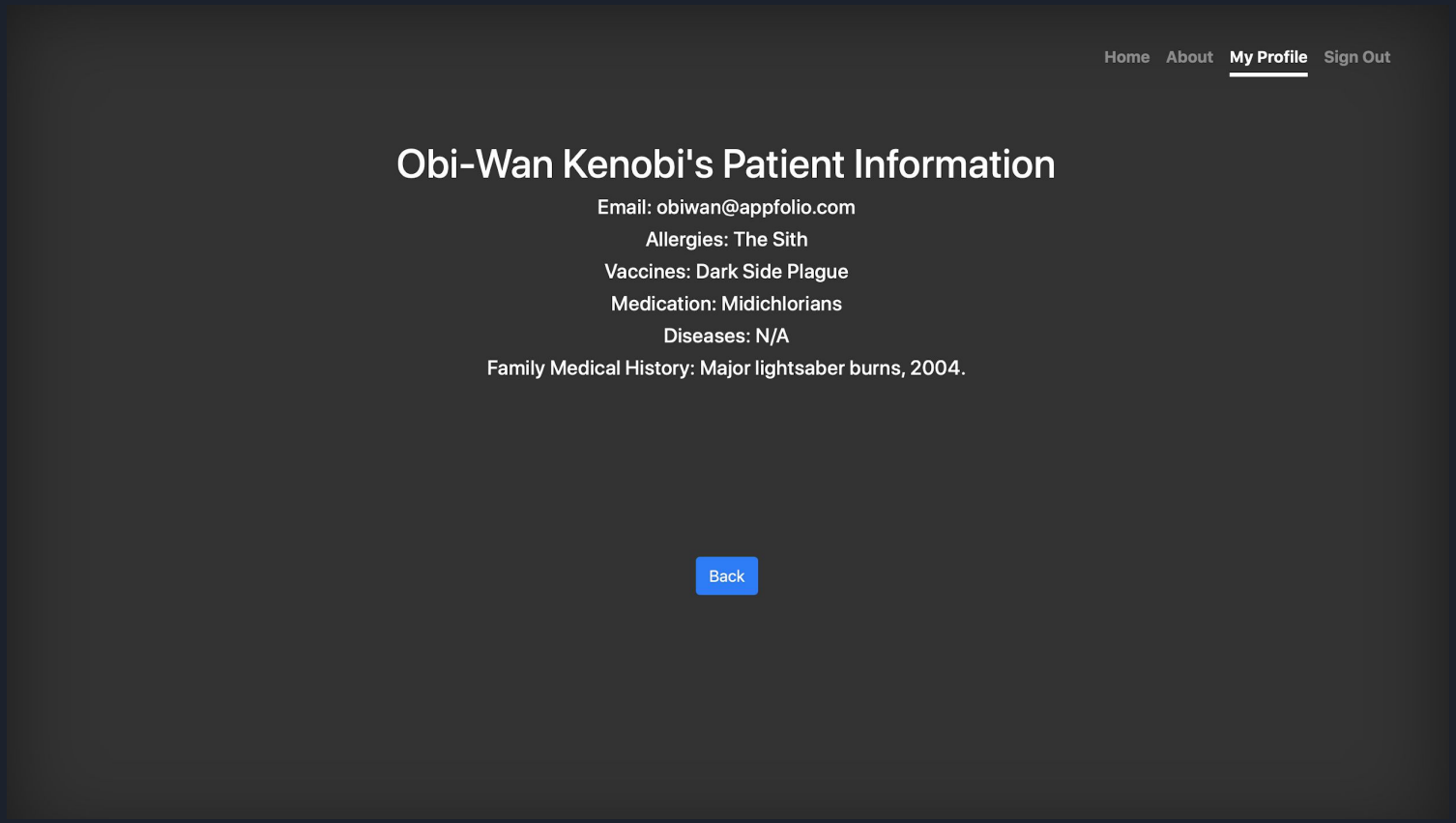
Once you have followed providers, “My Providers” updates to reflect this



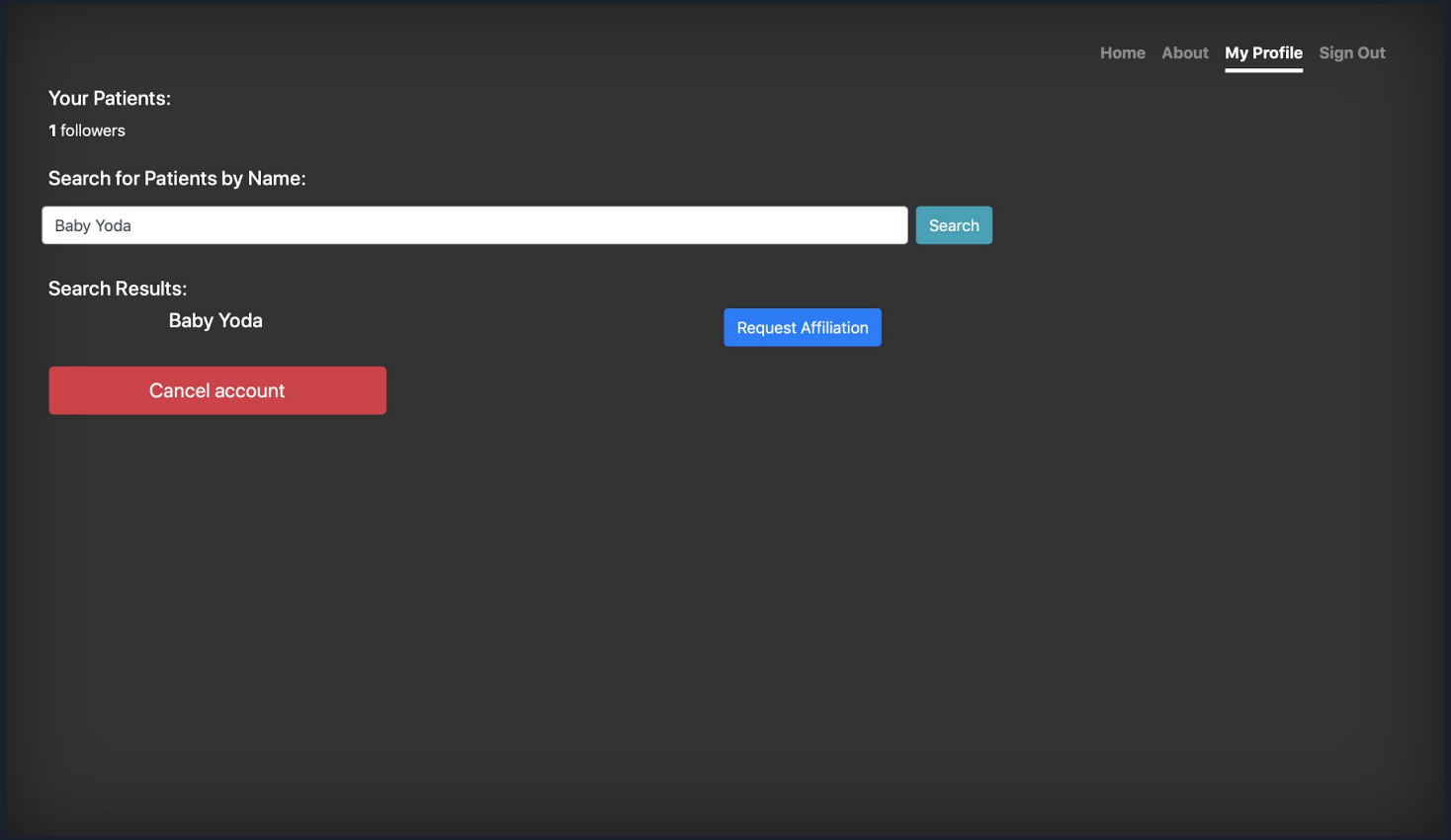
A provider's profile page allows them to view the patients that follow them, and search for new patients to request affiliation with



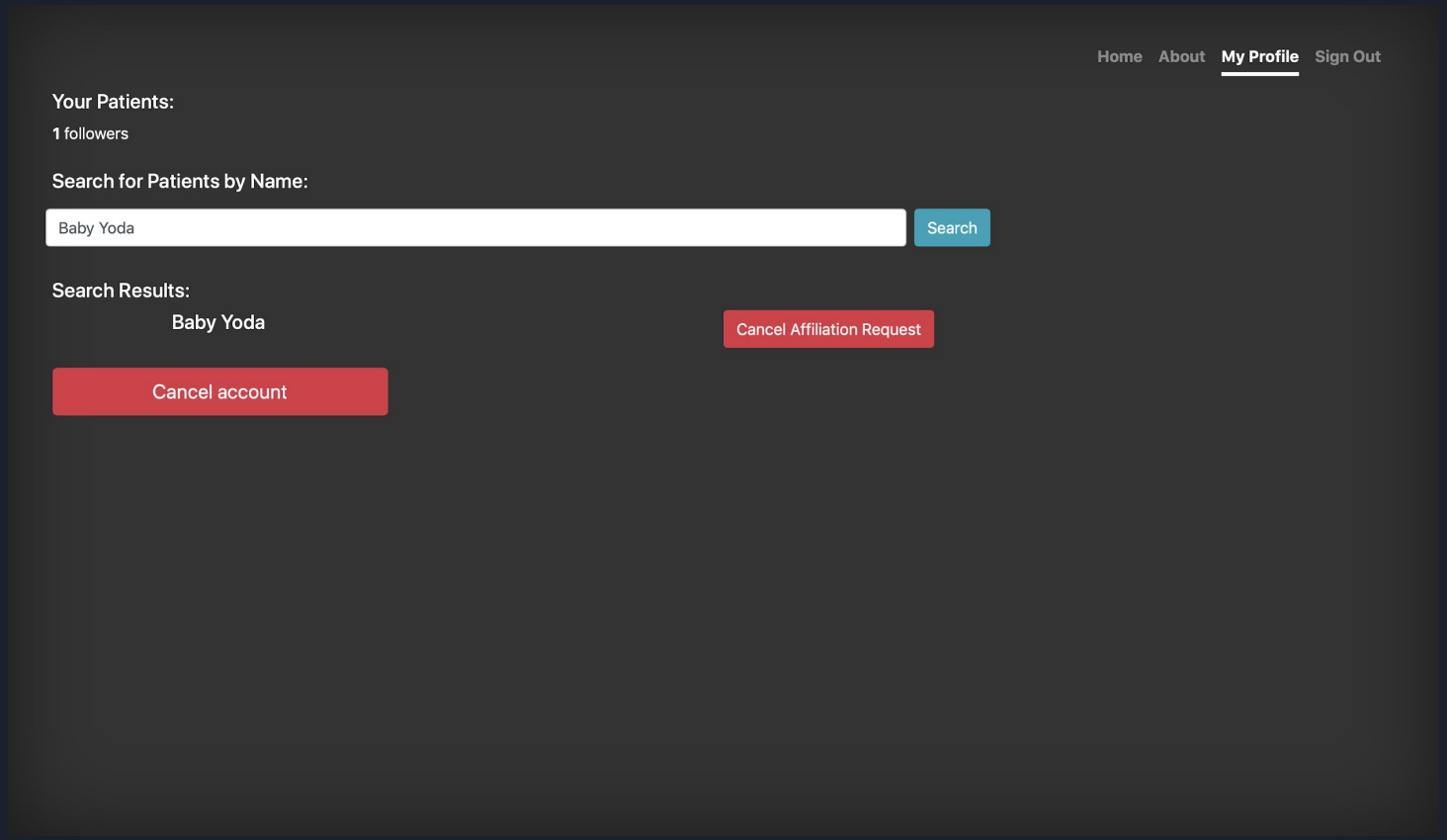
Here, we see that Obi-Wan Kenobi has followed Appfolio Health Services, and so we can access his info



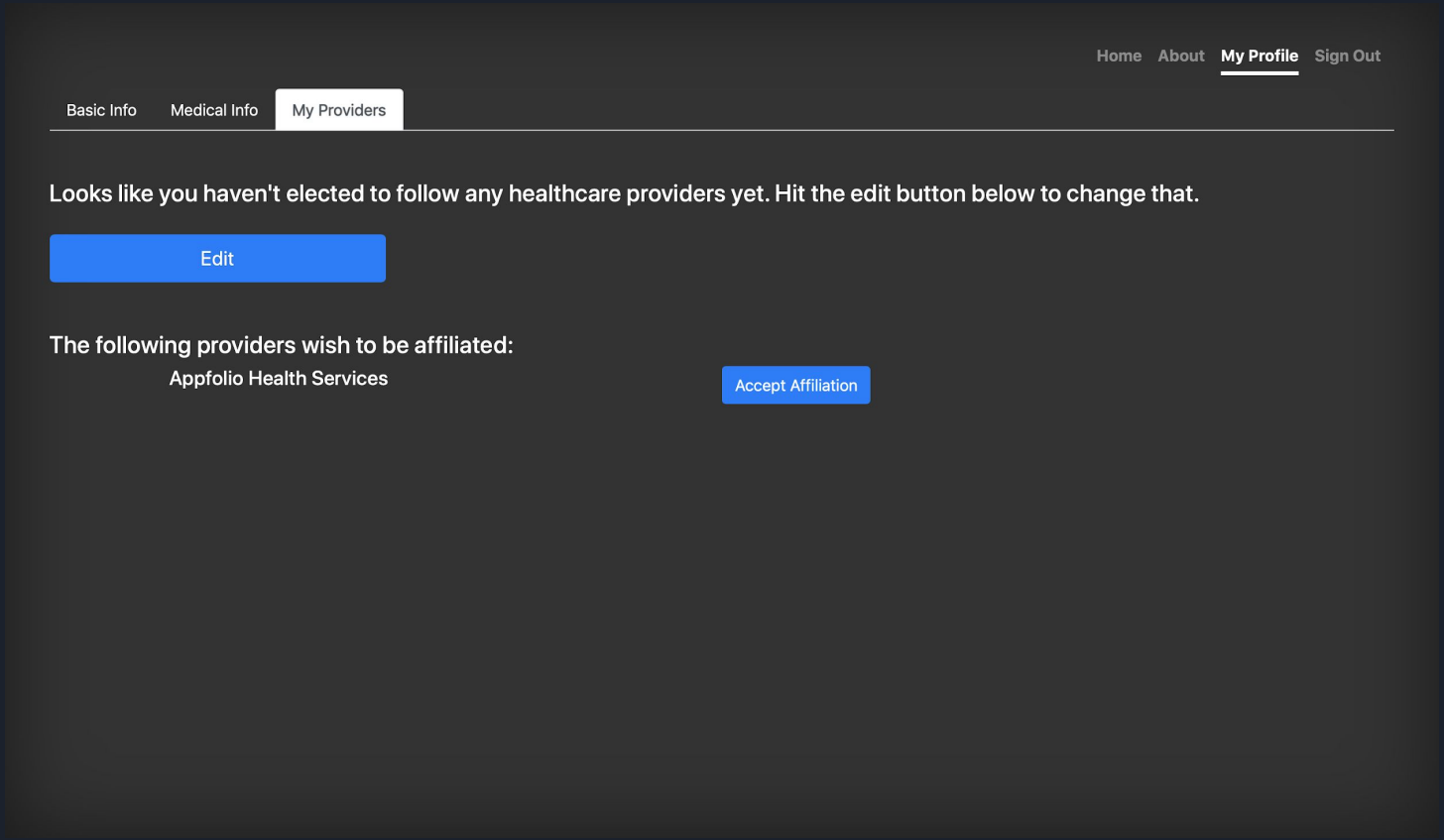
Now we know to keep Obi-Wan the heck away from the Sith!



Searching for a patient is also easy, just type in their name...



...and request affiliation!



On Baby Yoda's end, we can see that we have a pending request to be followed - which we can accept or ignore

Experiments and Results

Critical User Paths for Load Tests

- Our critical user paths were as follows:

1. `HTTPS GET`: Visit the Home page
2. `HTTPS GET`: Visit the Sign Up page
3. `HTTPS POST`: Sign up as a healthcare patient
4. `HTTPS GET`: Visit the Medical Profile page
5. `HTTPS POST`: Fill out the Medical Profile page with random 15-character strings
6. `HTTPS GET`: Visit the user's (Healthcare) Providers page
7. `HTTPS GET`: Search for one of the 10,000 pre-initialized healthcare providers
8. `HTTPS POST`: Randomly follow (Affiliate with) one of 10,000 pre-initialized healthcare providers
9. `HTTPS GET`: Visit the user's Profile
10. `HTTPS GET`: Visit the user's "My Providers" page (to see the affiliation made in #7 above)
11. `HTTPS POST`: Delete the user's account

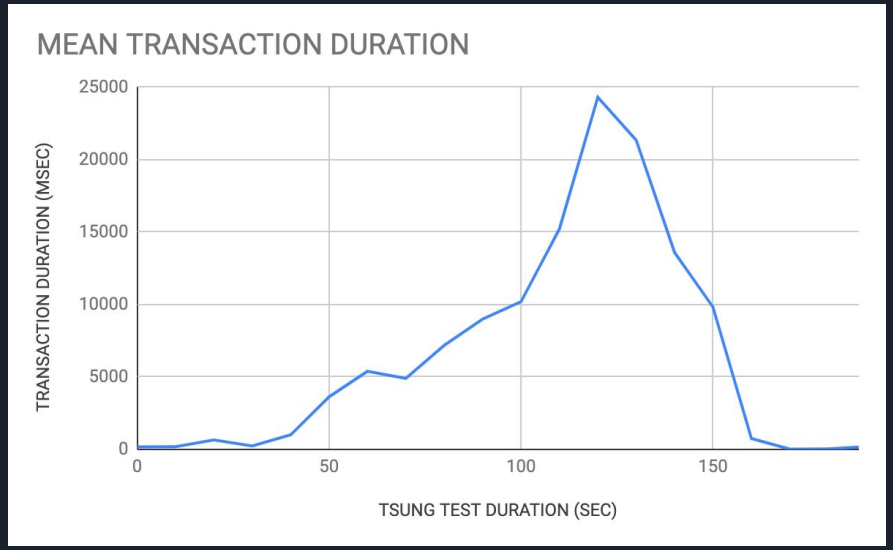
- The user waits either up to 2 seconds for simple actions, or up to 5 seconds for more complex actions
- 3 user arrival phases*
  - 1 user/sec → 2 users/sec → 4 users/sec

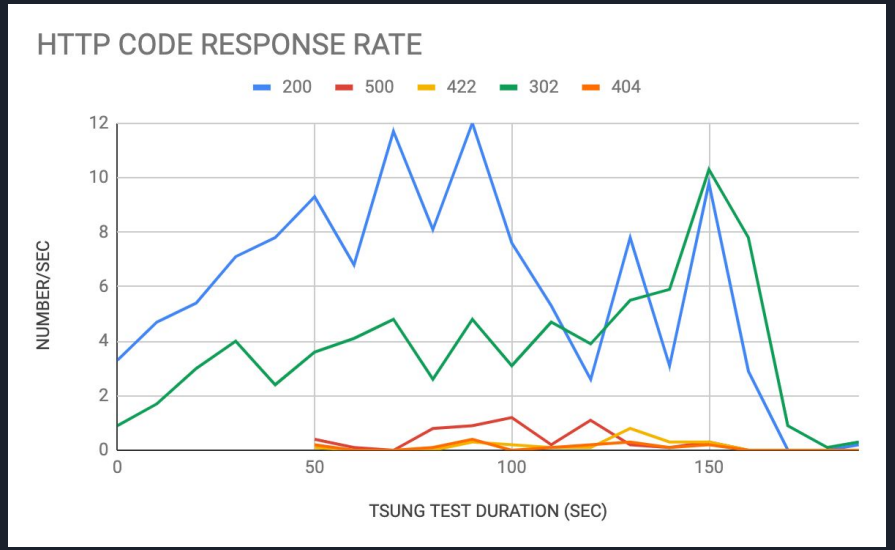
Database Seeding / Cleaning for Load Tests

- 1 seeding SQL script
  - Added 10,000 providers to database
  - Added 5,000 patients to database
    - Each patient followed one provider
- 1 cleaning SQL script
  - Cleared the *users* and *relationships* tables

Baseline Performance Test Results

- m3.medium
- Starts to break at **50 seconds** into the Tsung script
  - 20 seconds into second phase (2 users/sec)!



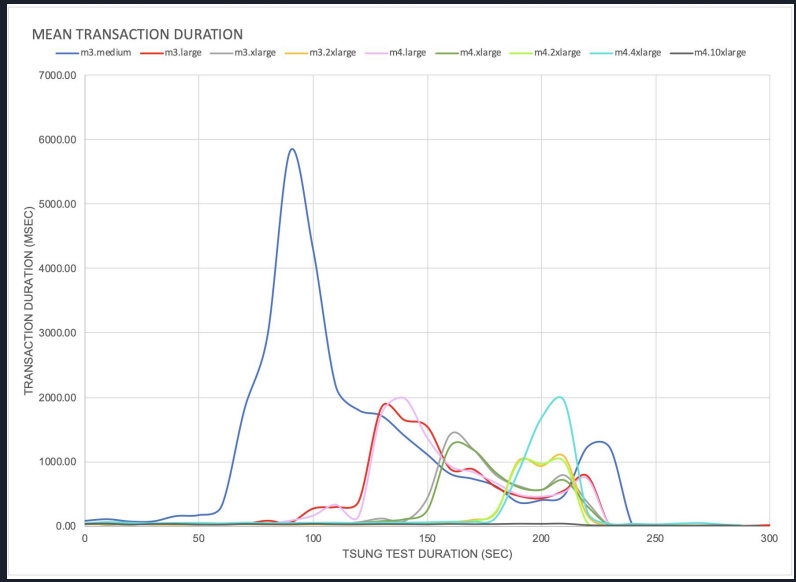


Vertical Scaling

- Utilized 9 different instance types:
  - M3.Medium, M3.Large, M3.XLarge, M3.2XLarge
  - M4.Large, M4.XLarge, M4.2XLarge, M4.4XLarge, M4.10XLarge
- Felt this provided a great range of hardware to see impact of vertical scaling
- Further, utilized a more intensive load script than for other parts of the app
  - Forced us to use some of the largest instances to prevent the app's failure
  - It had 7 phases, going up to 64 users/sec performing actions

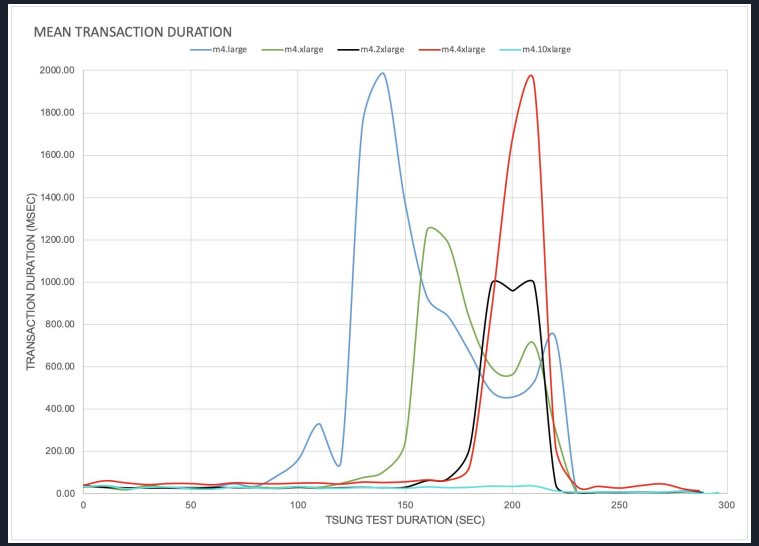
Vertical Scaling Results

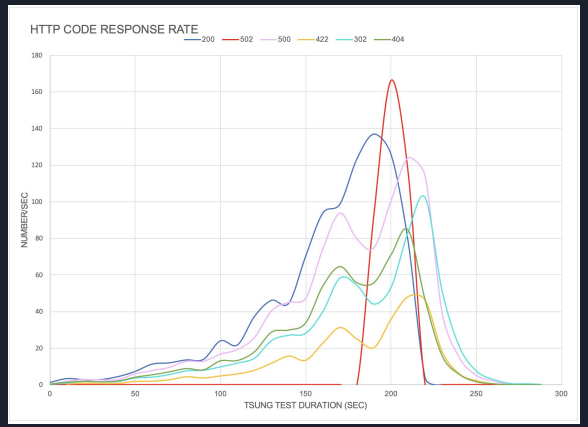
- The baseline instance of m3.medium is the blue line
  - This shows baselines performance with the more intensive load script used for this section
- Easily see that vertical scaling was a success
  - Using better instances shifts the spike right
    - i.e, beefier instances are failing further into the test, under even higher loads
    - This is good
  - They also spike to a lesser degree
    - You see baseline peaks at ~6,000 msec
    - All others beneath ~2,000 msec, at worst



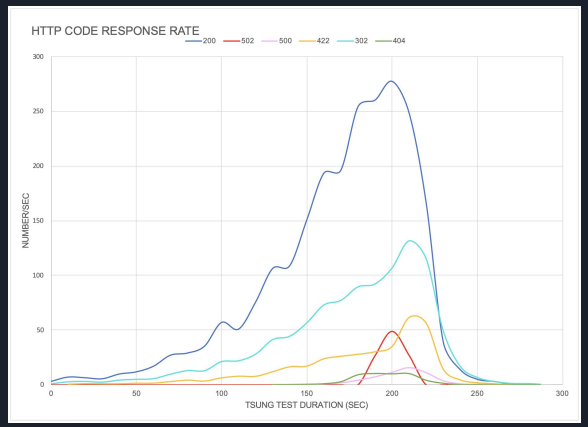
Mean Transaction Time for all Vertical Scaling Instances

- In more depth: the M4 instance variants definitely show the trend we are looking for
- Curious exception of M4.4XLarge
  - We would expect it to have a lower peak and be shifted further to the right than M4.2XLarge, but this is not the case
- Look at the two HTTP code rate graphs, though
  - M4.2XLarge is just throwing out 502s and 404s like John throws out candy
  - M4.4XLarge is taking longer to service requests, but is doing so successfully





M4.2XLarge HTTP Codes

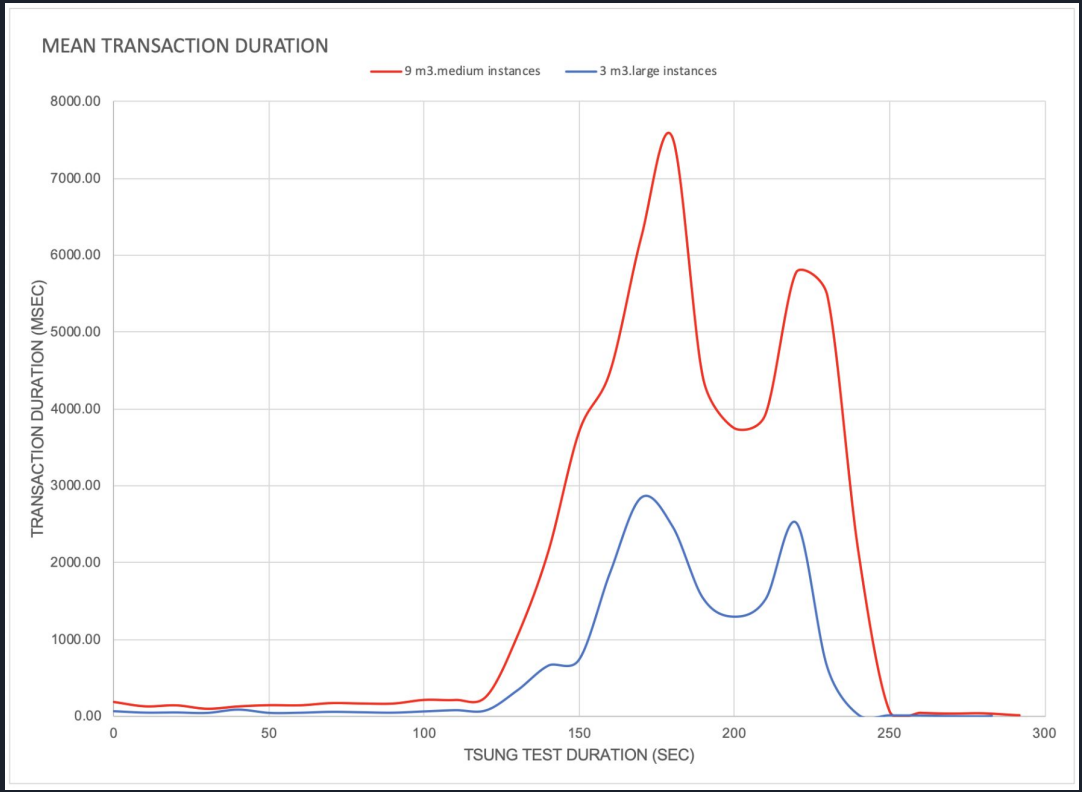


M4.4XLarge HTTP Codes

Horizontal Scaling

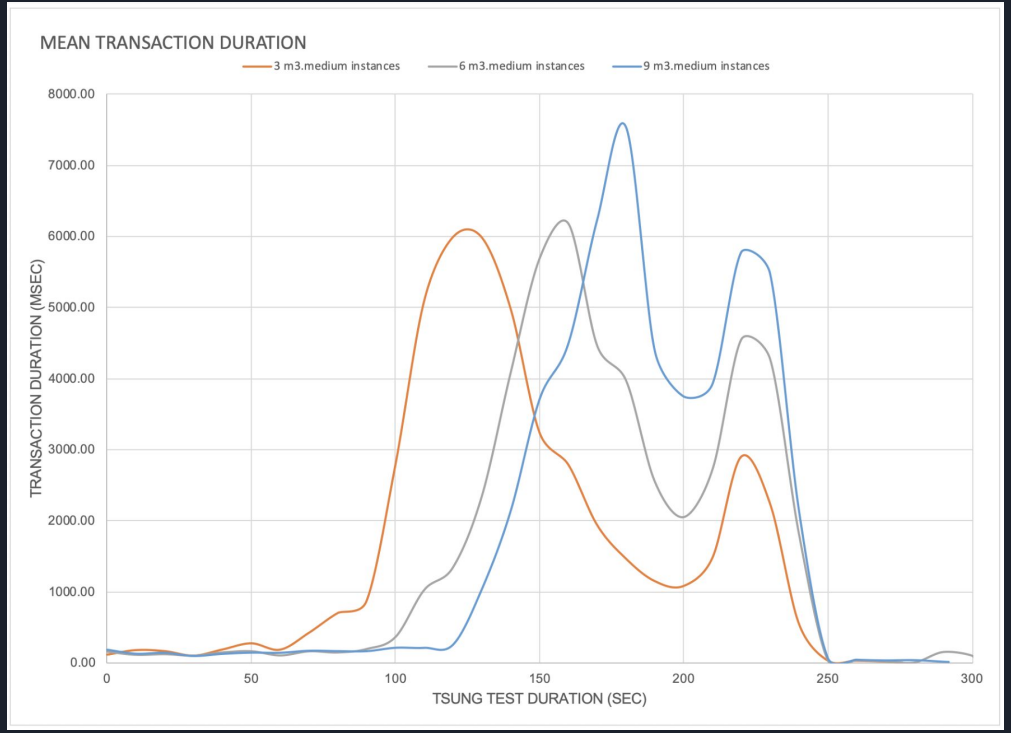
- We utilized 6 different arrangements for horizontal scaling:
  - 3 M3.Medium instances
  - 6 M3.Medium instances
  - 9 M3.Medium instances
  - 3 M3.Large instances
  - 6 M3.Large instances
  - 3 M3.XLarge instances
- Again, felt that this provided a nice variety of instance counts and instance power
- Also, again, utilized the more intensive, 7-phase load script
- All horizontal arrangements improved compared to the baseline
  - Broke far later into the test than the baseline 50 second mark

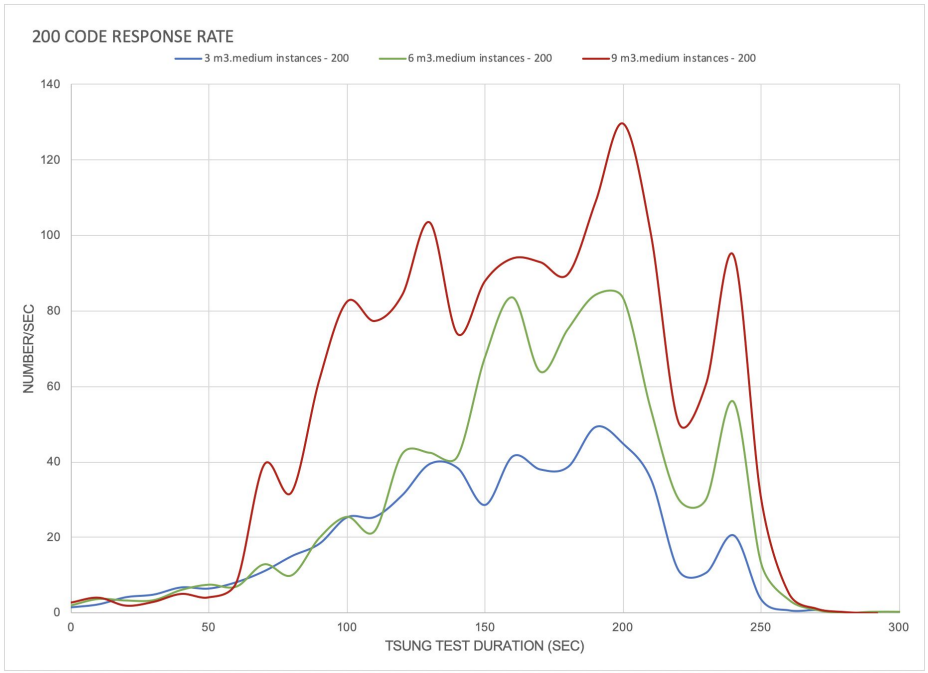
Horizontal Scaling Results

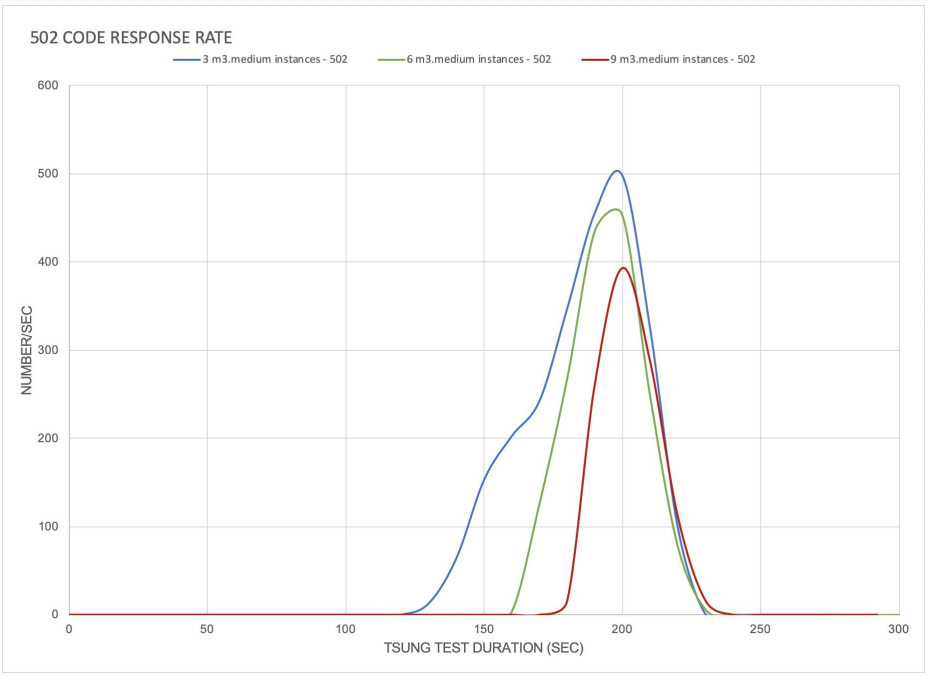


Generally, arrangements more powerful instances were more successful

- Curiously, response times actually increased when comparing scaling across more of the same instance type
- On the face of it, this is the opposite of what we would expect
  - 9 m3.medium instances should perform better than m3.medium instances
  - This graph does not seem to indicate that
  - Though the peaks moved in the right direction, they got larger
- Don't worry, there is an explanation







- Though more instances makes worst-case response time longer, they increase overall performance
- The arrangements with more of the same type serve far more HTTP 200s, and far fewer HTTP 502s

SQL Optimizations

- Analyzed underlying queries of all critical user actions
- Decided to optimize:
  - Adding (extra) indices
  - Preloading
  - Column Data Type Changes
- Each tested separately, had varying results

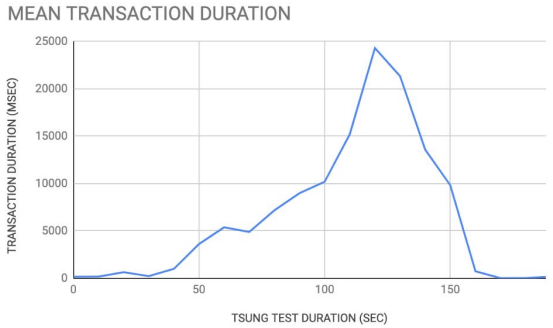
SQL Optimization #1: Adding Extra Indices

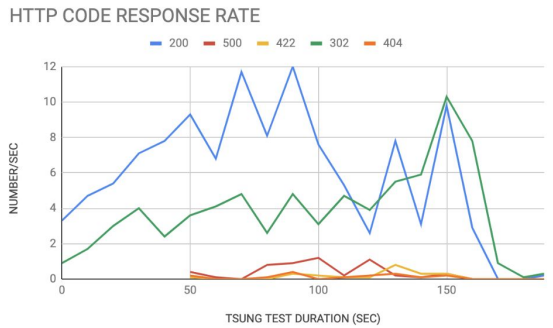
- Most queries used the default id index created by Rails
- So we added extra indices
  - E.g. index on the *name* column in the *users* table, since patients search for providers by name
- (Results shown in next slide)

Indices Results (cont.) - Good!

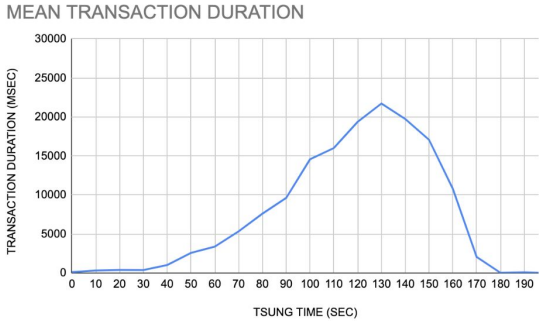
- Baseline breaks at 50 sec
- With indices breaks at 80 sec

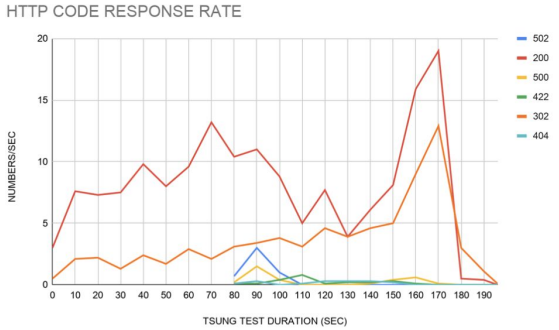
Baseline





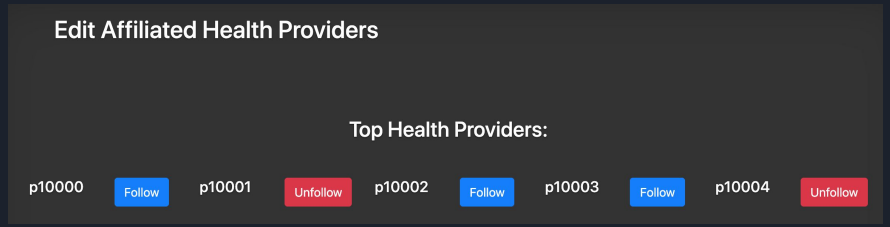
Adding Indices

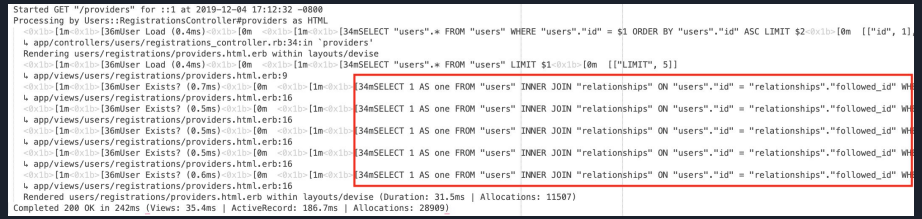




SQL Optimization #2: Preloading

- We found that one of our controller actions (GET */providers*) was very inefficient
  - Exhibited the N+1 problem with SQL joins
  - */providers* endpoint looped through a list of providers and checked whether or not the logged-in user was affiliated w/ each provider
    - Decide whether to show the “Follow” or “Unfollow” button
  - Each loop iteration inner joined users + relationships





SQL Optimization #2: Preloading (continued)

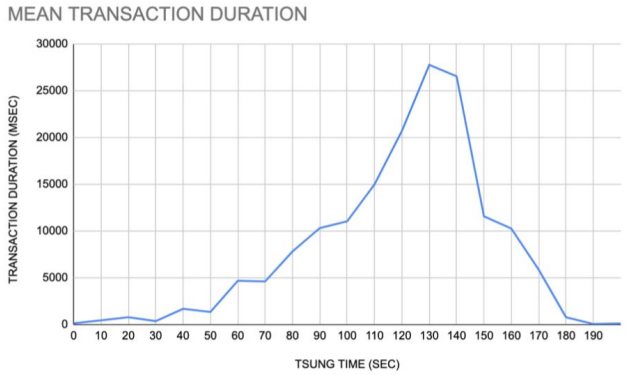
- One-line fix:
  - @current_user = User.includes(:following).find_by_id(current_user.id)
  - where “following” == “currently affiliated”

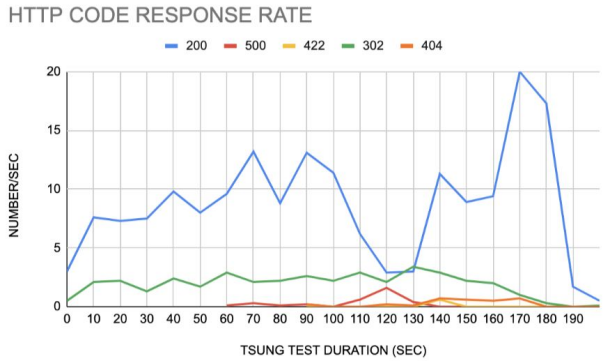
```
Started GET "/providers" for ::1 at 2019-12-04 17:14:16 -0800
Processing by Users::RegistrationsController#providers as HTML
  <0x1b>[1m<0x1b>[36mUser Load (0.4ms)<0x1b>[0m  <0x1b>[1m<0x1b>[34mSELECT "users".* FROM "users" WHERE "users"."id" = $1 ORDER BY "users"."id" ASC LIMIT $2<0x1b>[0m  [["id", 1], ["LIMIT", 1]]
  ↳ app/controllers/users/registrations_controller.rb:34:in `providers'
  <0x1b>[1m<0x1b>[36mUser Load (0.8ms)<0x1b>[0m  <0x1b>[1m<0x1b>[34mSELECT "users".* FROM "users" WHERE "users"."id" = $1 LIMIT $2<0x1b>[0m  [["id", 1], ["LIMIT", 1]]
  ↳ app/controllers/users/registrations_controller.rb:34:in `providers'
  <0x1b>[1m<0x1b>[36mRelationship Load (0.3ms)<0x1b>[0m  <0x1b>[1m<0x1b>[34mSELECT "relationships".* FROM "relationships" WHERE "relationships"."follower_id" = $1<0x1b>[0m  [["follower_id", 1]]
  ↳ app/controllers/users/registrations_controller.rb:34:in `providers'
  Rendering users/registrations/providers.html.erb within layouts/devise
  <0x1b>[1m<0x1b>[36mUser Load (0.3ms)<0x1b>[0m  <0x1b>[1m<0x1b>[34mSELECT "users".* FROM "users" LIMIT $1<0x1b>[0m  [["LIMIT", 5]]
  ↳ app/views/users/registrations/providers.html.erb:9
  Rendered users/registrations/providers.html.erb within layouts/devise (Duration: 14.9ms | Allocations: 5155)
Completed 200 OK in 118ms (Views: 72.3ms | ActiveRecord: 11.0ms | Allocations: 26550)
```

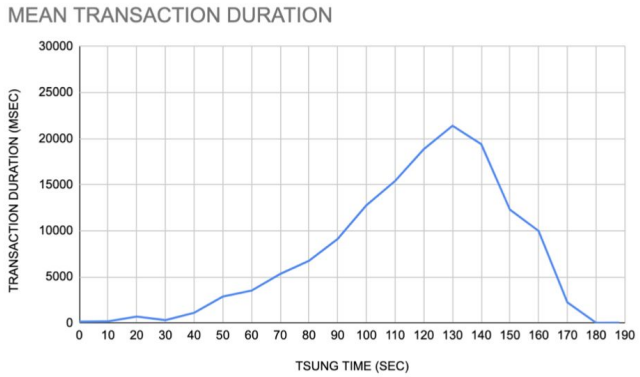
Preloading Results - Good!

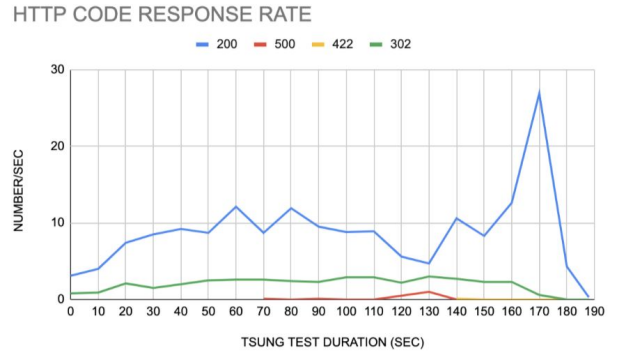
Baseline breaks @ 60 sec

Preloading Breaks @ 70 sec









SQL Optimization #3: Changing Column Data Types

- Changing the column types in our *users* and *relationships* tables to reduce row size
  - id columns: *bigint* type → *integer* type
  - variable-length string → fixed length string (20 chars max)
- Performance did not change much
  - Initial load tests did not insert large data values into database

Changing Column Type Results - Little Change

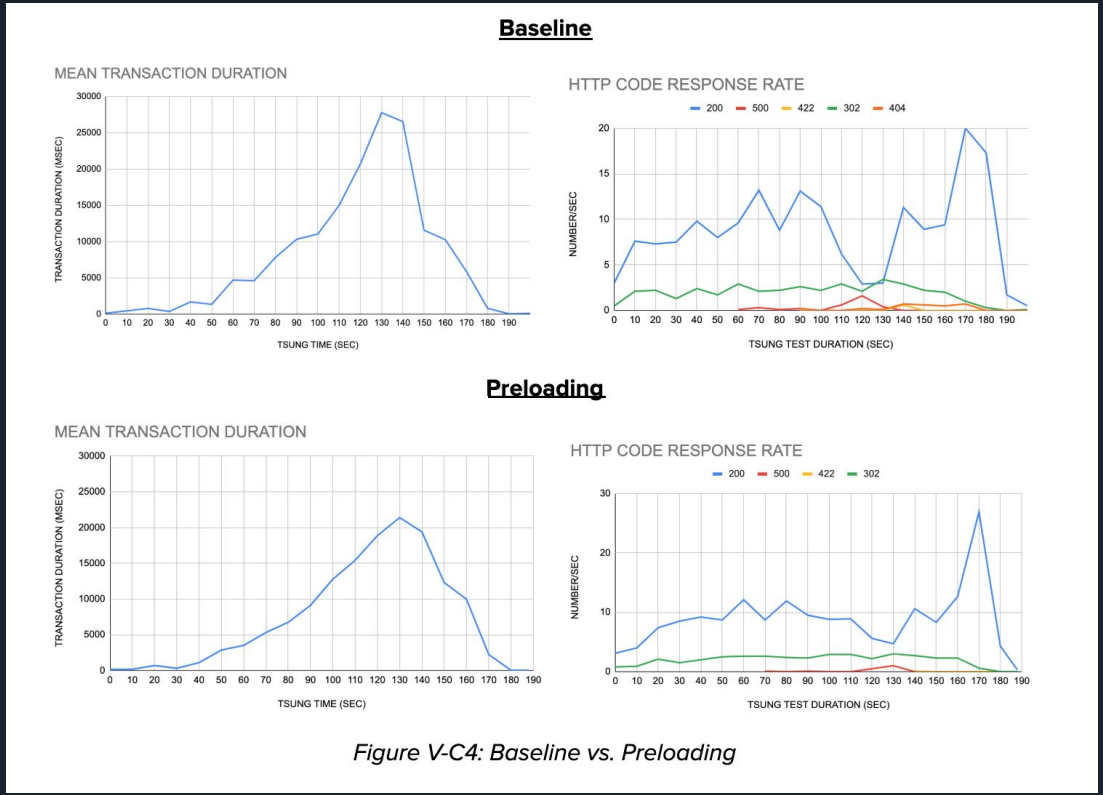


Figure V-C4: Baseline vs. Preloading

Caching

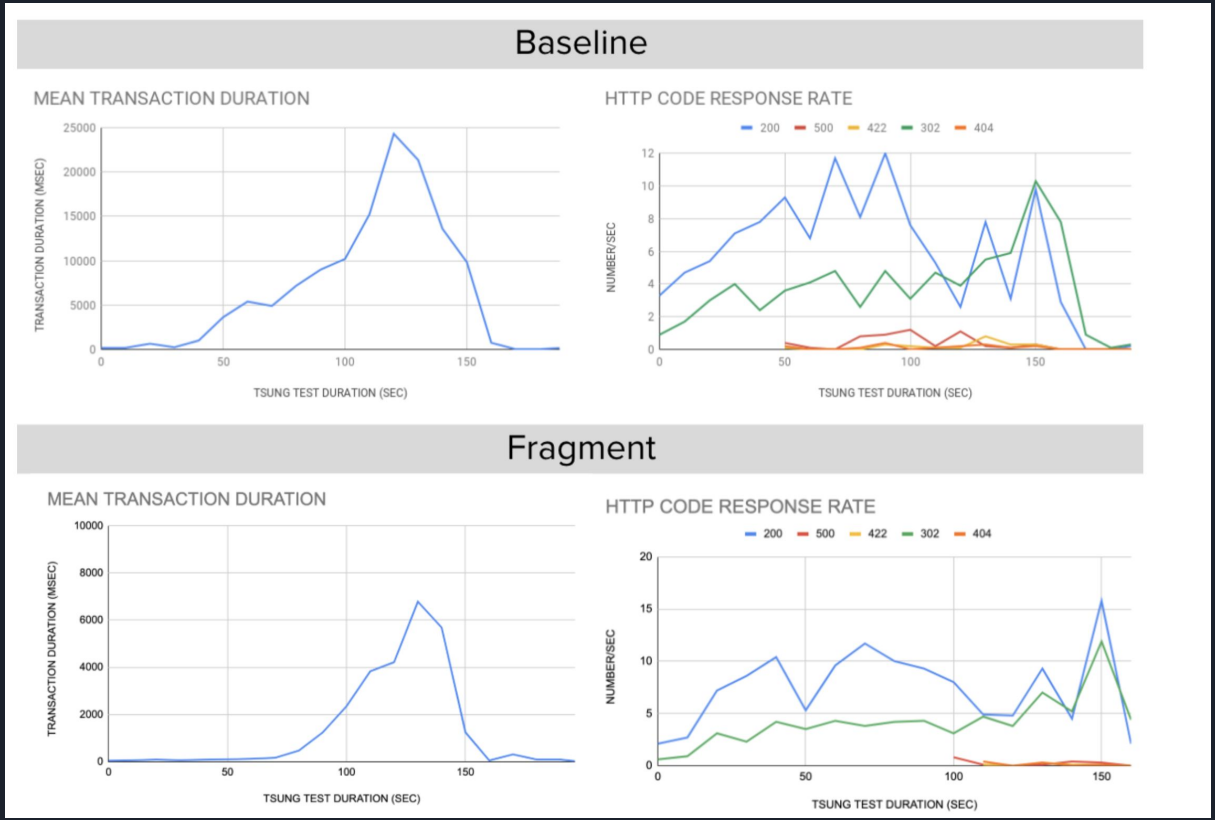
- We tried various methods of caching, all met with at least modest success
- These included:
  - Fragment Caching
  - Low-Level Caching
  - Memcaching
  - Client-side caching

Fragment Caching

- Form of server-side caching
- Allows apps to cache individual (*fragment*) components of a view
- Fragment caching can reduce server load and increase view generation
- We conducted fragment caching according to shared and unique buckets
- Shared view fragments are further broken into static and form pages
  - Static pages can be cached using a simple cache key - nothing
  - Form pages require care to ensure we cache the form, never the submission
- User-specific fragments are the unique ones
  - Use info unique to the user to tag caches, allowing their retrieval

```
def cache_key_for_profile_info
   "user_#{current_user.id}_#{current_user.updated_at}"
end
```

Fragment Caching - Results



Low-Level Caching

- When an app needs to cache query results instead of an entire view fragment.
- Generally caused by computationally expensive, relatively stable queries
- However, our application doesn't have many instances of this
  - The most relevant example would be our search functionality
- Despite this lack of a general use case, we pressed onward

```
@patients = Rails.cache.fetch("/user/#{params[:search]}", :expires_in => 5.minutes)
```

Low-Level Caching - Results

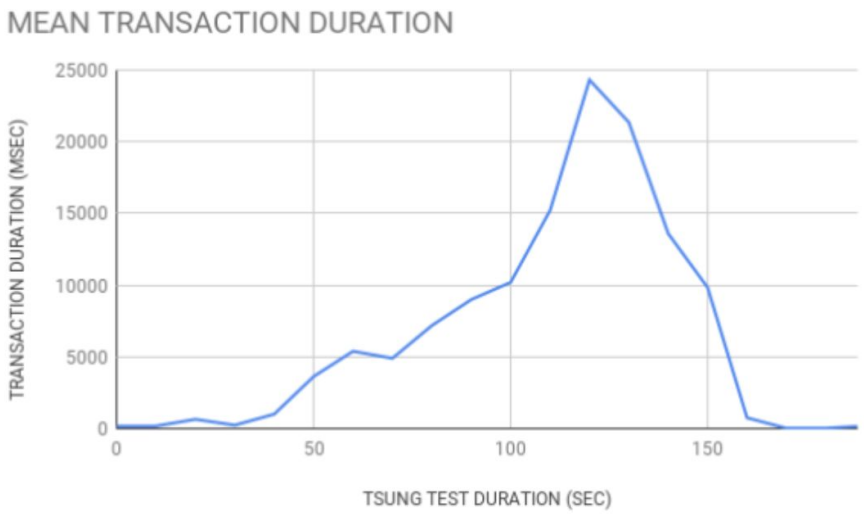
- Our application was slower with low-level caching!
- We think this is caused by two things
  1. Memory is limited to a certain size
     - Caching name-dependent items could result in new caches overwriting old caches, depending on name length
  2. Our Tsung scripts create names so that they are random
     - Due to randomness, there isn't much overlap between names to use
     - Results in increased rate of unique search queries
     - Thus also results in a reduced cache hit rate due to overwrites
     - So the app's basically wasting time caching info on each search, slowing it

Memcaching

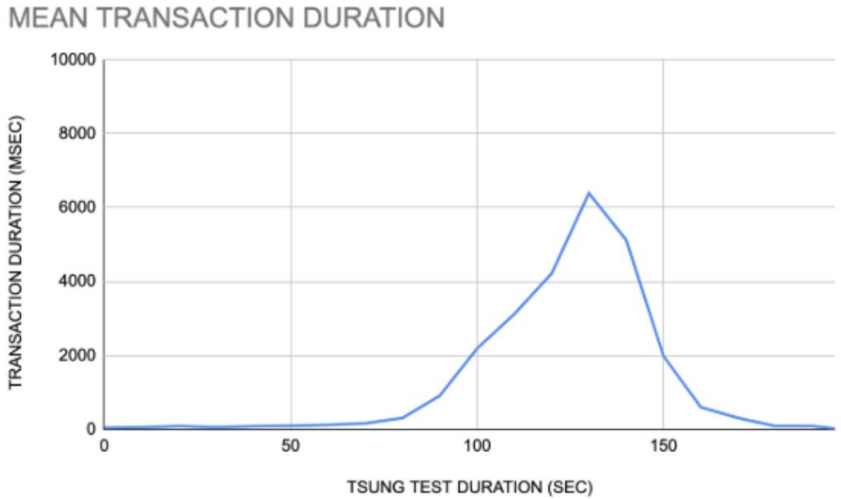
- By default, Rails caches info in memory
- This can be problematic: the cache bounded, by default, to a size of 32 Mb
- When the cache stores has exceeded the allocated size, it auto-cleans
- Enter: memcaching
- Memcaching provides a centralized cache for your rails application
    - Called a centralized cache cluster (CCC)
    - Boosts performance and reduces redundancy
- We used the *dalli* gem along with ElastiCache to create our CCC
- Then, continued using our fragment caching from before to utilize the CCC

Memcaching - Results

Baseline



Fragment (memcache)



Client-Side Caching

- Instead of caching view fragments server-side, we can have browser make GET requests
  - Req's dependent on whether the contents of a page have changed or not
- To do this, we implemented the **fresh_when()** method in our Application Controller

```
fresh_when([current_user.id, User.all, Relationship.all])
```

- **fresh_when()** sets etag and modified headers of user and relationships table
- The server can then determine whether or not a GET request is necessary
  - Based on whether the computed etag matches the etag computed upon request.

Client-Side Caching Results

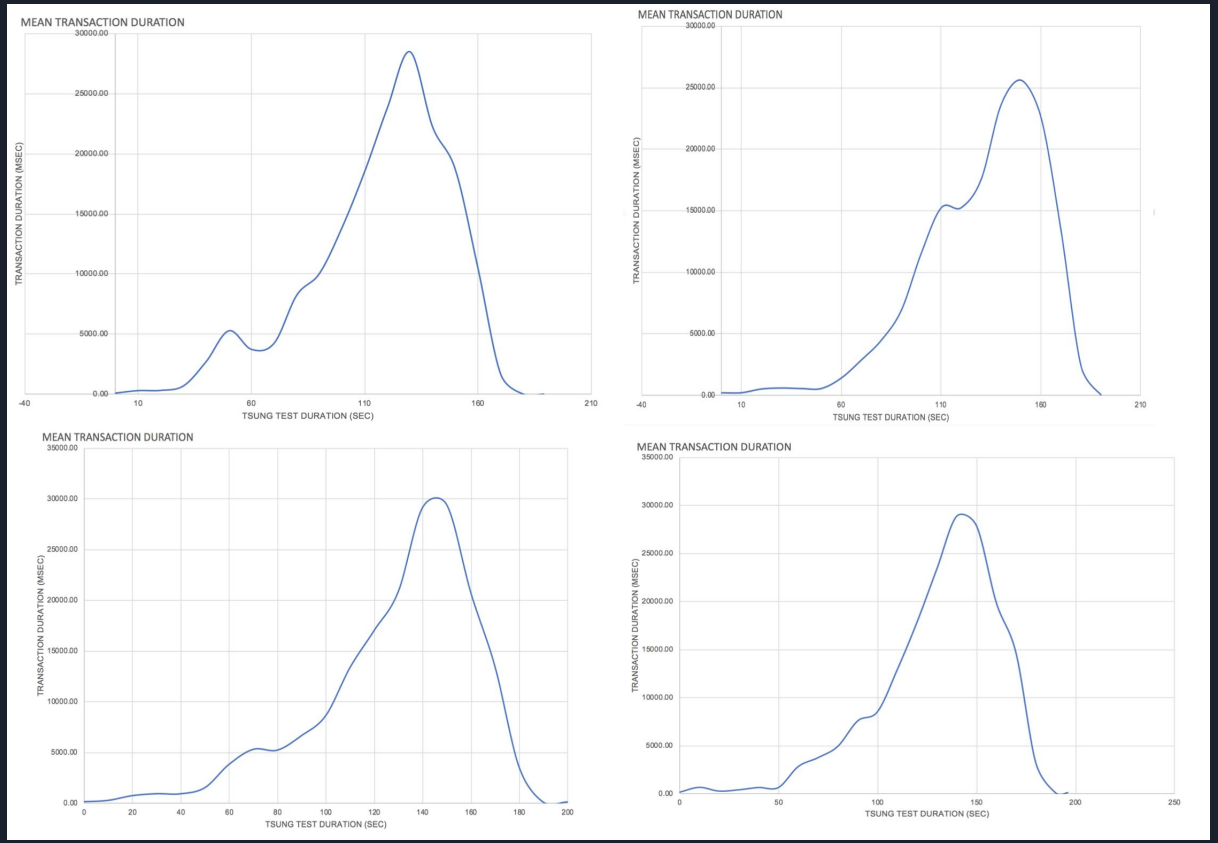
Name	Status	Type	Initiator	Size	Time
providers	200	xhr	turbolinks.js:154	3.3 KB	88 ms
providers	304	xhr	turbolinks.js:154	979 B	46 ms
medical	200	xhr	turbolinks.js:154	3.8 KB	158 ms
medical	304	docum…	Other	979 B	25 ms

- Upon the first request of our profile pages, we do a full GET request to receive info
- As long as the user doesn't edit their information, all subsequent requests hit the cache
  - Results in a complete bypass of view rendering and a status code of 304
  - As shown, can reduce response time by 125 ms in the case of the medical page

Manipulating Processes and Threads

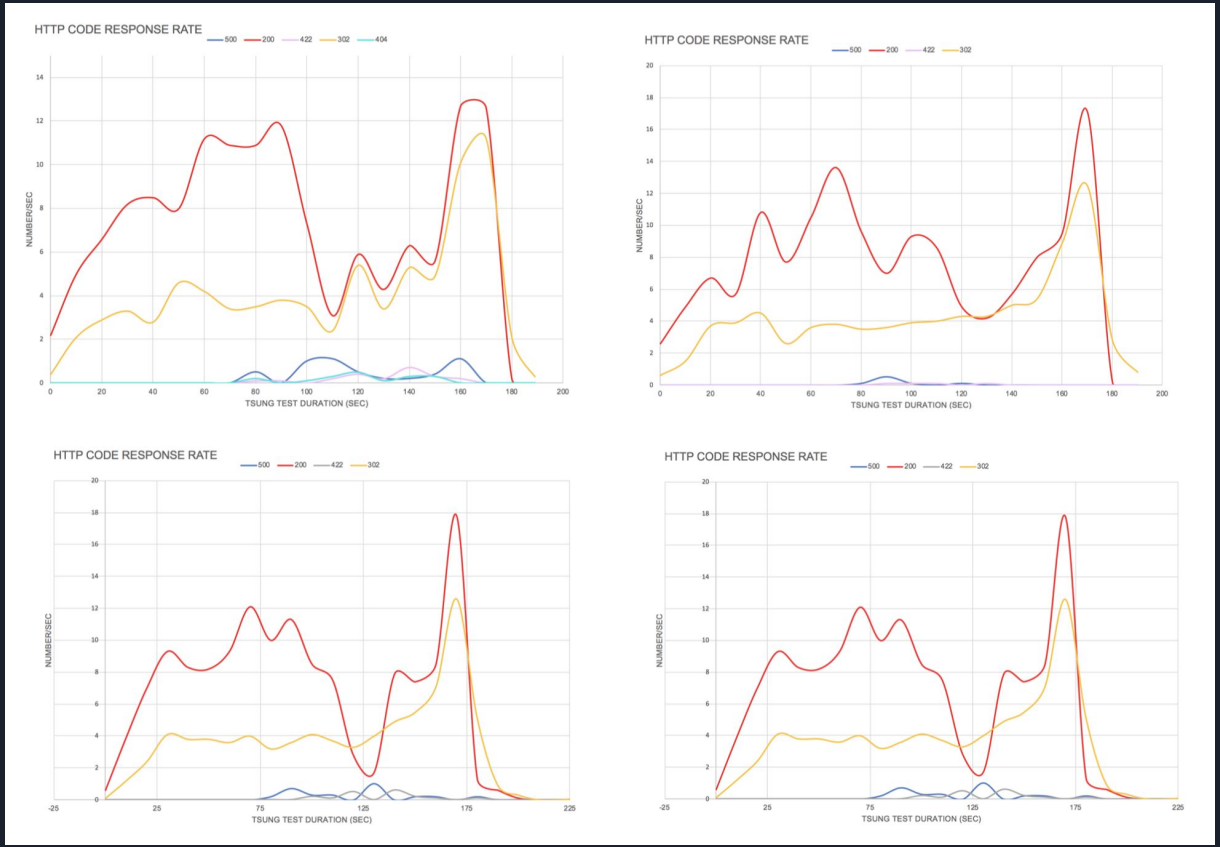
- Each instance is configurable in terms of the processor and thread count
  - Easily modifiable via a config file
- First used a baseline of 1 thread, and 1 process
  - As expected, this provided generally poor performance
- Used the following combos of threads/processes to measure impact:
  - 1 thread, 8 processes
  - 15 threads, 8 processes
  - 15 threads, 1 process

Manipulating Processes and Threads - Results



Mean Transaction Duration for 1 Thread 1 Process (top-left), 1 Thread 8 Process (top-right), 15 Threads 8 Processes (bottom-left), and 15 Threads 1 Process (bottom-right)

Manipulating Processes and Threads - Results



HTTP Code Response Rate for 1 Thread 1 Process (top-left), 1 Thread 8 Process (top-right), 15 Threads 8 Process (bottom-left), and 15 Threads 1 Process (bottom-right)

Read Slaves

- Originally, we were going to use the Octopus gem to configure read slaves
  - Turned to John's advice instead: creating a read replica manually on AWS' RDS
- The *config/database.yml* and activation record were changed to configure support
- The concept of 'automatic connection switching' was added in the application config
  - Allowed the app to switch access to the primary or replica based on the HTTP VERB
    - POST will switch to primary
    - GET requests will switch to replica - unless there was a recent write

Read Slaves - Results

Undone Optimizations (... so close!)

- Sadly, we were unable to implement sharding and the SOA optimizations
- This was simply a matter of time constraints - there wasn’t enough time to get it done
- Were we to continue explore scaling this app, these would be first on the list

Conclusions

- Ultimately, most of the scaling methods tried were at least somewhat successful
  - Horizontal and vertical scaling gave us the largest impact for the least effort
  - Well-targeted attempts at caching and SQL optimizations pulled their weight
- In general: use your design overview and user critical path analysis to ensure scaling methods are hitting the right spots
  - Fragment caching, preloading and indices boosted performance greatly
  - Changing data types, low-level caching and memcaching were less effective
- Always consider cost versus effectiveness, too
  - Do you really need an M5.Metal instance?

Lessons Learned

- If you want to build a meaningful app today, you better be ready to scale it
- Ensuring even a relatively simple app will work with an explosion in the user base is no small task
  - It requires thoughtful planning, careful analysis and targeted improvements
  - One of the largest difficulties is that it is hard to determine their eventual value
    - Horizontal scaling could result in a tradeoff between response times, and adequately-served requests
      - For Whispr, that might be acceptable
      - For another app, maybe not so much (a financial tracking app, perhaps)
- Scalability is perhaps the most important thing to consider when initially designing an application
  - Aesthetic choices can be easily reversed at any point in development
  - Changing baser aspects after-the-fact typically amounts to a from-scratch rewrite - i.e., sadness
- Team organization is key
  - Have a broad plan, and more specific, more flexible weekly plans
    - Otherwise, easy to get lost in the weeds, or be unresponsive to things learned during dev
- Use tech and practices the team is most comfortable with
  - Pair programming may no be for you, and that's okay!

Questions?