Michael Chase

East Bay, CA ~ Email ~ LinkedIn ~ GitHub | v09.17

About

Favorite Quote: *"A man who knows something knows that he knows nothing at all" - Erykah Badu*

Favorite Language: Python

Favorite Editor: vim

Values: Challenges, Experimentation, Deep Learning, Trendsetting, Career Building, Culture

Skills: python2.7/3.6, flask, sanic, NodeJs, Restify, Express, React-Native, Javascript, React, Angular 1.x/2, MySQl, MongoDB, Redis, Docker, EC2, ECS, Terraform, Jenkins, Architecture, Problem Solving

Experiences

Collecting and notifying on gdax.com trade statistics

Around December, 2016 I got into cryptocurrency trading. By early February 2017 I had a functional, single exchange, arbitrage bot written in `python2.7` using `gevent` for threading. This bot was very basic and not very profitable. This project was the first time I used the `Decimal` Python library. I ended up making the bot a service that could run against multiple accounts via a `flask` API. More research and tinkering led me to abandon the Arbitrage method.

By the end of May, I had refactored this project to `python3.6` using `sanic` and `asyncio`. In its current but evolving state, you can signup/in and view some poorly displayed data on various pairs. Data is stored in `mysql` and I am utilizing `alembic` to maintain schema migrations. In the future, it will allow you to setup custom notifications.

Hobby	Closed Source	Jan 2017 - present

Redis, HIPAA, and AWS

My teammates and I currently maintain a stack which consists of NodeJS, Restify, MySQL, and Memcached. This stack powers a localized, custom built survey application. With our migration to AWS and commitment to Privacy, our data was instructed to follow HIPAA requirements. At its core, this meant Encryption at Rest and in Transit. I researched some solutions and led our team to initially implement `stunnel` and a custom deployed `redis`

instance on `ec2` (orchestrated through `terraform`). After some trials, tribulations, and a couple group discussions, we ended up utilizing `ElastiCache` and handling (en/de)cryption at the app tier with a `kms` data key. In addition, we created a `python2.6` tool to help manage secrets. It uses `iam roles`, `ssm parameter store`, and `kms` to securely store and retrieve sensitive app runtime data.

Senior Software Engineer	Ancestry	Jan 2017 - present

Handling orders at scale

On a previous team, we maintained a full-stack e-commerce site that took a variety of orders in a variety of languages. This site was initially architected with `angular1.x`, `python2.7`, `tornado`, `sockjs`, `celery`, and `rabbitmq`. This site only handled couriering data inputs to a backend ordering system. In addition to helping build and push a few client side features, I proposed we refactor and re-architect the application to no longer use websockets and queues. In addition, I wrestled `pika` to work with `tornado` in order to get rid of a mix of threading, polling, and in-memory cache which was used to connect `sockjs` to `celery`. Afterwards I was able to design and implement a system that used classes to define and register `rpc` action handlers as the first phase in swapping long-polled `websockets` for `http`.

Software Engineer	Ancestry	Aug 2015 - Jan 2017

Giving people control over their data

A few friends and I got together with a plan to help simplify basic and overlooked data communication. The outcome of this project was a private beta app called Query. I led the architectural design of the iOS app which was developed using `React-Native` while also developing the backend via `python2.7`, `flask`, `mongodb`, and `redis`. We knew we'd want to one day launch the app on multiple platforms, so I split out the core data-to-server logic of the app into a separate project. The design consisted of stores which were `fbemitter`'s, Object-Oriented to re-use common logic, and multiple `dispatcher`'s to separate `state` updates in the various `stores`. This allowed for the app logic to react to state changes in sibling stores which was very helpful when dealing with object updates over `websocket`. In order to keep `Promises` at bay and handle flows such as OAuth, I created and open-sourced a library called StaceFlow.

In order for people to have control, the api had to handle permissions. Initially, these permissions were simple "`Share`" objects which gave `usera` access to `itemb` which is owned by `userc`. Later, we realized user groups and item groups would be a thing as well as permission types. So I went back to the drawing board and created a

permission system backed by `mongodb`. This system is able to understand a complex permissions structure where a group of people can have permissions over a group of items, which was provided by a single user. If, for example, that user's permissions are revoked, so are the groups.

Query later transformed into Sequel which still uses the original `react-native` and `python` backends.

Consultant	**Useful Labs Inc.**	**Jan 2017 - Present**
Co-Founder/Lead Engineer	**Useful Labs Inc.**	**Aug 2015 - Jan 2017**

Notifications at web scale

On the first day of my first job at a startup, the CEO (my direct manager) announced I would be working with one other engineer to finish rewriting the core api. (Prior to that I had never touched a production api in life). At the time, the api was written in `python2.7`, used parse for data storage and push notifications, and did all of its requests (including sending notifications) on the request thread. The senior engineer had been re-writing the api in `python2.7`, `gevent` and `mongodb`. This gave us some good initial benefits and allowed us to get notification sending off of the request thread. However, together we took this a step further by implementing `redis-queue` to create a horizontally scalable distributed message queuing system. While there, I was able to find and fix a bug in `mongoengine` which would, in some circumstances, cause $O(n^2)$ comparisons.