Michael Tomer

Phone: 617-842-0905
Address:
800 West St.
Braintree MA, 02184

Email: michaeltomer@gmail.com
Blog: http://actsasbuffoon.com
Github: actsasbuffoon
Twitter: @michael_tomer

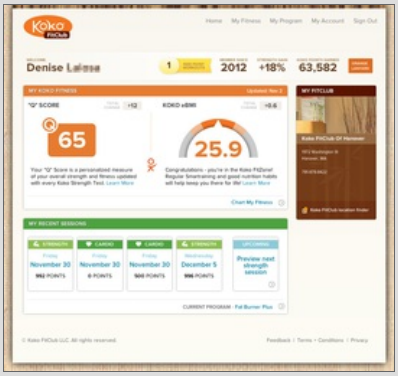
I've been programming since I was eleven years old, and using Ruby for around four and a half years. I also work with a lot of JavaScript/CoffeeScript, and I'm very comfortable with event driven programming. For the last 14 months I've been a developer at Koko Fitclub in Rockland Massachusetts.

I've deployed many Rails, Sinatra, and Rocket (a web framework I created) apps. I've used various SQL databases, MongoDB, and Redis. My experience with MongoDB is particularly interesting, as I've actually made extensive use of its schemaless nature.

I've created client-side JavaScript applications with Backbone and Ember, and consider myself to be a capable JavaScript/Coffeescript developer. I've also got a lot of experience developing RESTful APIs to drive client-side applications.

I enjoy TDD, and have worked extensively with RSpec, MiniTest, and Cucumber. I've also spent a lot of time with capybara-webkit. I like working with a close-knit team and pair programming. Agile/Scrum is preferred, though I'm flexible.

October 2011 - December 2012 — Koko Fitclub



A screenshot of the Koko Fitclub website.

I developed some very interesting software during my time with Koko. I was brought in to assist with a complete rewrite of Koko's backend. The project was a fascinating combination of web development, API work, and embedded devices.

My proudest achievement at Koko was the creation of Apiary, an incredibly flexible and powerful system for building REST APIs. Apiary is able to create highly consistent, secure, searchable APIs. I created a query language that lets users filter, sort, limit, and nest data arbitrarily, but is secure enough that you can't gain access to data you're not authorized to see.

Apiary also allows you to securely nest model data on POST requests. It lets you nest model data arbitrarily deep, while protecting you from nested data pointing at foreign keys the user isn't authorized to access.

Apiary can generate HTML documentation that details the URLs, HTTP methods, and attributes available for each endpoint. It also allows you to quickly create many different APIs to target individual clients. For instance, you can have an API with one set of models, attributes, and permissions just for an iPhone app, and another API with a different set of models/attributes/permissions for a Backbone app.

Here's an example of an endpoint configuration file in Apiary. Note that this is all the code necessary to set up two separate APIs for a single model.

```
module Api
  class WorkoutsController < ApplicationController
    include Apiary
    respond_to :json

    private

    def self.model
      Workout
    end

    # The API definition for a Backbone.js app on the user-facing site.
    client('user_site') do

      # Attributes can be white-listed. When used this way, all other attributes are disallowed.
      allow_readable :weight, :duration

      # You can also restrict access to a subset of REST actions.
      actions :index, :show

      # Apiary uses ActiveRecord query proxies to ensure only a subset of data can be accessed.
      scope do |query_proxy|
        query_proxy.where user_id: current_user.id
      end
    end

    # The API definition for iOS, and Android apps.
    client('mobile_app') do
```

```
      # You can also take a black-list approach. Any attributes not mentioned here will be exposed.
      allow_all_readable_except :something_private
      allow_writable :comment
      actions :update, :index, :show

      scope do |query_proxy|
        query_proxy.where user_id: current_user.id
      end
    end
  end
end
```

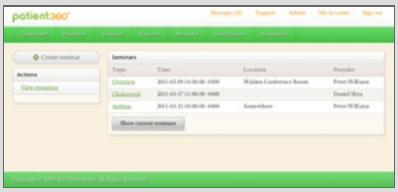
I also built a Backbone.js app to replace an aging .NET desktop app. The app was responsible for managing member data and creating special maintenance keys to update our embedded devices in the field. It also had to deal with reading and writing data to USB keys, which obviously isn't something you can normally do from a webpage.

Though the app was basically a branching wizard with roughly a dozen pages, it was a great opportunity to dog-food Apiary and the APIs I had created with it.

Here's a link to a brief video walkthrough of the Backbone app. It demonstrates the API and USB key interaction.

http://actsasbuffoon.com/smartsetup

October 2009 - October 2011 Patient360



A screenshot of the Patient360 website.

I was the lead developer at Patient360 for two years. I developed a number of services for P360, mostly aimed at physicians and other medical professionals. They were one of the first government certified PQRS registries, which is a program that pays incentives to doctors for improving the health of their patients.

I built a clinical decision support system which helped doctors diagnose patients, as well as an automated system for contacting and scheduling patients via phone, email, or SMS.

Implementing a pay for performance system is no small feat. PQRS alone had about 180 metrics (as of 2011), with new metrics issued each year. Each metric involves a series of questions and some rather convoluted logic to determine eligibility and quality of care. Some metrics are as simple as a yes or no question, while others involve half a dozen questions and computing averages for various test results.

I opted to solve the problem by leveraging the schemaless nature of MongoDB. I also created a DSL that greatly simplified creating and maintaining metrics. The DSL allows us to create a 15-30 line file that handles data storage, acts as a simple ORM, creates the HTML forms for users to enter data, determines eligibility, and which billing codes to report to the payers.

```
PQRIEngine.create do
  metric_year 2011
  metric_number 45
  metric_type :claims, :registry
  group :e
  title "Perioperative Care: Discontinuation of Prophylactic Antibiotics (Cardiac Procedures)"
  description "Percentage of cardiac surgical patients aged 18 years and older undergoing procedures
  with the indications for prophylactic antibiotics AND who received a prophylactic antibiotic, who
  have an order for discontinuation of prophylactic antibiotics within 48 hours of surgical end time"
  age_above? 18
  cpt_codes %w[33120 33130 33140 33141 33250 33251 3325633261 33305 33315 33332 33335 33400 33401]

  ask :q1, :label => "Patient given prophylactic antibiotics within 4 hours prior to surgical
  incision, or given intraoperatively?", :as => :boolean

  ask :q2, :label => "Please choose the option that most closely applies", :as => :radio, :collection => [
    "Prophylactic antibiotics discontinued within 24 hours of procedure",
    "Prophylactic antibiotics not discontinued for medical reason",
    "Prophylactic antibiotics not discontinued, reason not specified"
  ]

  numerator do
    if q1
      if q2 == "Prophylactic antibiotics discontinued within 24 hours of procedure"
        [["4043F", "4046F"], :pass]
      elsif q2 == "Prophylactic antibiotics not discontinued for medical reason"
        [["4043F-1P", "4046F"], :exclude]
      elsif q2 == "Prophylactic antibiotics not discontinued, reason not specified"
        [["4043F-8P", "4046F"], :fail]
      end
    else
```

```
      ["4042F", :exclude]
    end
  end
end
```

Realtime Web Application Framework — Rocket

Screenshot of an unreleased product developed with Rocket.

I'm the creator of the Rocket realtime web application framework. It's a hybrid Ruby/JavaScript framework that uses web sockets for nearly everything. It's completely event driven, but uses Ilya Grigorik's EM-Synchrony library to prevent users from having to write callbacks.

Rocket approaches web applications differently than any other framework that I'm aware of. The idea is to push as much processing to the client as you securely can. Even HTML generation takes place on the client.

I gave a presentation about Rocket at RailsCamp 2011, and approximately 40 minutes worth of screencasts have also been made available. You can watch the screencasts on my Vimeo channel: http://vimeo.com/28167729

This is the documentation site for Rocket.

Unlike most other web frameworks, Rocket isn't aimed at building traditional content-oriented sites. Rails, Sinatra, and many other excellent frameworks exist for serving that need. Instead, Rocket is meant for creating web applications that feel native.

Rocket also lives up to its name; it is incredibly fast. My favorite remark from a user is that the response times are, "almost unsettlingly fast."

Key/Value Store — Mnemosine

Mnemosine is a key/value store I created for a CodeBrawl competition (ultimately placing fourth out of eighteen entries). It's a client/server application, and the server is completely event driven.

Currently Mnemosine is a partial clone of Redis, but I'm working on adding more features. It already supports a map system (as in map/reduce, but without the reduce phase), but I'd like to add sharding and possibly indexed search.

Ruby probably isn't a great language to implement a database with, as it isn't the most performant choice. That said, Ruby makes it very easy to profoundly modify the way the database works. I think Mnemosine could be a great testbed to experiment with new ideas before going through the effort of modifying a more complicated C application.

That said, I'm strongly considering rewriting the core data storage/retrieval in C or C++ for speed, while keeping most operations written in Ruby. The extensive test coverage should make converting the application quite easy.

I'd like to take a moment to go over the Mnemosine client, as I think it's a great example of the power of Ruby. The entire client consists of 25 lines of code (excluding whitespace) despite the fact that it covers 30 API methods.

Since most methods consist of nothing more than JSON encoding the arguments and passing them to the server, it was very easy to define most of the methods automatically. I also made use of 1.9's new lambda syntax to allow a method defined via define_method to take a splatted array argument (a very nice trick to remember!).

Here's the code:

```
class Mnemosine
  class Client
    def initialize
      @socket = TCPSocket.open('localhost', 4291)
    end

    api_methods :set, :get, :delete_all, :delete, :keys, :exists, :rename, :renamenx, :incr,
                :decr, :incrby, :decrby, :append, :randomkey, :hset, :hget, :hmset, :hmget,
                :hlen, :hkeys, :hincr, :hincrby, :hdecr, :hdecrby, :hgetall, :hexists, :hdel,
                :hsetnx

    def select_keys(&blk)
      send_data({"select_keys" => blk.to_source})
    end

    def match_keys(regex)
```

```
        send_data({"match_keys" => [regex.source, regex.options]})
      end

    private

      def send_data(msg)
        JSON.parse(@socket.readline)["val"]
      end

      def self.api_methods(*mthds)
        mthds.each do |mthd|
          define_method mthd, ->(*args) do
            send_data({mthd => args})
          end
        end
      end
    end
  end
```

Other Work

Screenshots

Here's a sampling of some of the other work that I've done.