JOSHUA COOK

A CLOUD INFRASTRUCTURE FOR PROCESSING IMAGES

Contents

From a product perspective, we are performing a fairly straightforward task. Users are able to upload a photograph from their mobile phones to our servers for processing. Those image will be processed by our servers, associated with users and stored for retrieval, before being returned to users.

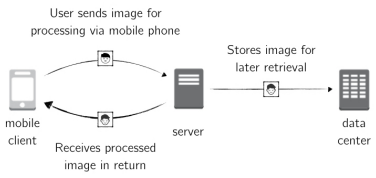


Figure 1: A rough sketch of our system

Operational Infrastructure

A Minimal System

The components for building a minimal system – one that can handle small- to medium-sized loads – are widely available. We recommend using Amazon Web Services (AWS) as a reliable, inexpensive, and well-documented cloud-based hosting solution. Amazon has a wide variety of systems abstracted to a very high level which allows the system architect to concern themselves with the design of their system rather than the implementation of that system.

Elastic Beanstalk

We recommend using AWS' Elastic Beanstalk service because it makes for a simple, rapid development period and because it is easily scalable based on load requirements. Elastic Beanstalk offers an abstraction to EC2[1] and EBS[2]. Beanstalk handles the most common configurations with minimal user input and according to AWS best practices with regard to:

- EC2 instance instantiation
- Security Group configuration
- Auto Scaling group configuration
- S3[3] instantiation

Furthermore, should additional configuration be required, the EC2 and EBS instances that are created for our Beanstalk application are fully configurable as any standard EC2 or EBS instance would be.

[1] Elastic Compute Cloud
[2] Elastic Block Store
[3] Simple Storage Service

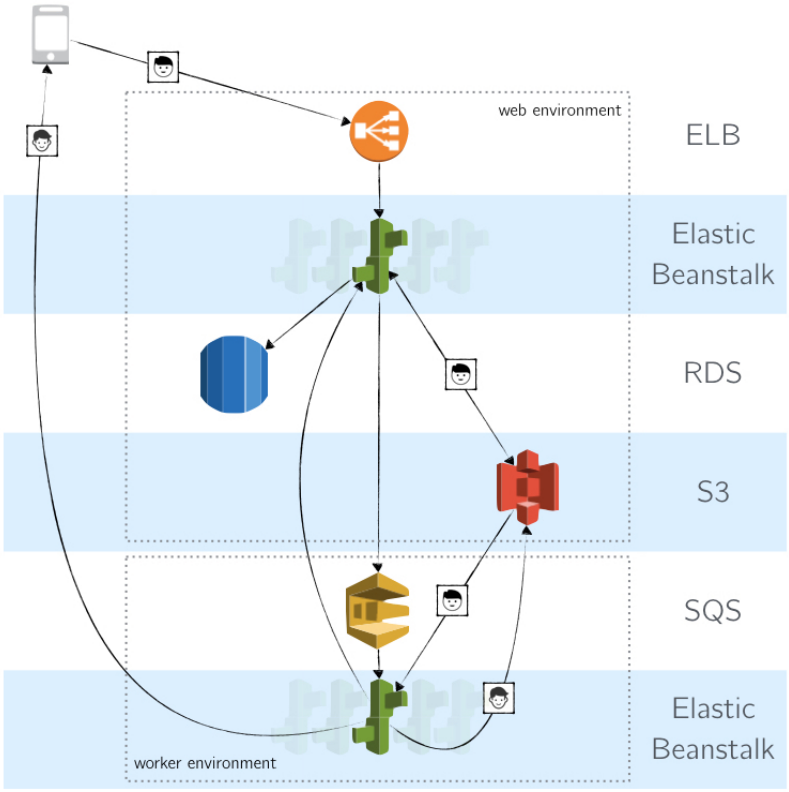
Elastic Beanstalk applications consist of many environments. Environments are available in one of two types:

- a web environment
- a worker environment

Each of these types of environments have different supporting system requirements and are by default configured with necessary systems. Both environments are built on top of an EC2 instance with an attached EBS volume and using the AMI[4] of our choosing. Web environments are configured with an ELB[5] instance, an RDS[6] instance, and an S3 instance for persistent storage. Worker environments are configured with an SQS[7] instance.

We set up a web environment – the Front-End. This will give us an ELB attached to a public IP and DNS record. Via either of these,

[4] Amazon Machine Image
[5] Elastic Load Balancer
[6] Relational Database Service
[7] Simple Queue System



we can send RESTful requests to our system. We are able to have n-instances of our web environment, a number that is easily dialed up or down using the AWS control panel, API, or command line interface on fifteen minutes notice. Each of these systems will be connected to a shared RDS instance, our common database.

We set up a worker environment – the Worker. This will give us an SQS instance which will behave much as the load balancer does for the web tier. We can configure our SQS instance, however, to not be publicly available, and only accessible to a system within the property security group and on the same VPC[8]. The web environment will communicate with the Worker environment via the SQS instance.

[8] Virtual Private Cloud

A Single Transaction – Minimal System

A single transaction will make its way through our system in the following manner:

1. the Mobile Client sends a RESTful request containing photograph metadata to our DNS
2. the ELB receives the request and sends the request to the next available Front-End node
3. the Front-End node receives the request and uses the metadata to creates a new object in RDS before responding with a success code to the Mobile Client
4. the Mobile Client initiates an upload to S3 as a background process
5. upon completion of upload the Mobile Client sends a RESTful request containing upload completion metadata
6. the ELB again routes the upload metadata[9]
7. the Front-End node updates the RDS object with the reference to the photograph's location in S3, then queues a processing job in SQS before responding with a success code to the Mobile Client
8. SQS makes the queued job message available
9. daemons on each Worker node periodically poll SQS for available messages
10. one or more Worker nodes execute the queued job[10]
11. upon completion of the job, the Worker node saves the processed image to S3, sends job completion metadata to a Front-End node[11], and sends the processed photograph to the Mobile Client
12. a Front-End node updates RDS with information on the completed job including a reference to the location of the processed image on S3 and removes any duplicates created by the Worker environment

[9] the request need not be routed to the same node

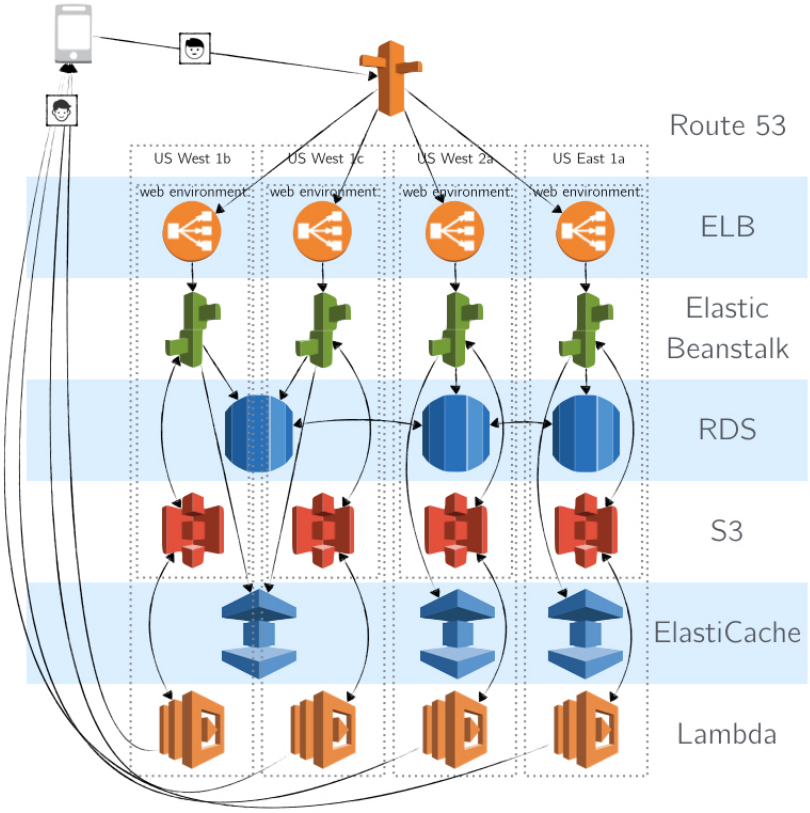
[10] the fact that more than one node may be executing the job is an aspect of SQS that must be handled properly

[11] via RESTful API request and thus through ELB ...

In the situation where all requests are made within a narrow time window – requests occur once per day or twice per day – we recommend the use of AWS' `boto` library to script the scaling of Elastic Beanstalk nodes. Approximately an hour before the anticipated influx of traffic, `boto` can be used to instruct Elastic Beanstalk to add as many nodes as will be required to handle traffic. Approximately an hour after the rush of traffic has completed another script can be used to tear nodes down. In between traffic burts, a minimal system can be kept in place to handle the errant request or for testing of the production environment during off-peak hours.

Additional Horizontal Scaling

Such a system is well scalable, particularly considering the use of load balancers and the ease of adding additional nodes via Elastic Beanstalk. There may be situations, however, in which additional considerations are required to allow for better handling of a large volume of requests.



In this system, we have duplicated our Elastic Beanstalk web environment across multiple availability zones[12] and geographic regions[13]. We use the Route 53 service to direct traffic to a single DNS entry-point to the appropriate system based upon availability and the Mobile Client's location. We have replaced the SQS service with ElastiCache and our Elastic Beanstalk worker environment with Lambda.

[12] e.g. US West 1b, US West 1c
[13] e.g. US West 2, US East 1

A Single Transaction – Scaled System

A single transaction will make its way through our system in the following manner:

1. the Mobile Client sends a RESTful request containing photograph metadata to our DNS
2. Route 53 receives the request and routes it to the appropriate ELB
3. the ELB receives the request and sends the request to the next available Front-End node
4. the Front-End node receives the request and uses the metadata to creates a new object in ElastiCache before responding with a success code to the Mobile Client
5. the Mobile Client initiates an upload to S3 as a background process
6. upon completion of upload the Mobile Client sends a RESTful request containing upload completion metadata
7. the ELB routes the upload metadata[14]
8. upon completion of uploading the Elastic Beanstalk web environment node updates the ElastiCache object with the reference to the photograph's location in S3, then queues a processing job in an ElastiCache PubSub before responding with a success code to the Mobile Client
9. the ElastiCache PubSub makes the queued job message available
10. via an EC2 proxy instance[15] we route the queued job message to Lambda
11. a Lambda process executes the queued job, processing the image, saving the processed image to S3, sending job completion metadata to a Front-End node
12. a Front-End node matches completed job metadata to an ElastiCache object and uses it to create an RDS with information on the completed job includng the location of the processed image on S3

[14] the request need not be routed to the same node

[15] Elastic Beanstalk web environment

This iteration of our first design provides for several advantages over the earlier system. Should significant requests be initiated across multiple geographic regions, this system will handle this seamlessly via easily scaled Elastic Beanstalk applications in each geographic region. Should our application require a no-fail success rate, our ElastiCache PubSub will prove a more robust option than SQS. As we scale to $10x$ or $100x$ in our requests, applications distributed across availability regions will provide another horizontal scaling mechanism for handling the increase in traffic. Finally, the use of Lambda to do our image processing will prove more inexpensive[16] than using full Elastic Beanstalk worker environments, necessitating full EC2/EBS instances.

[16] monetarily and computationally

Software Infrastructure

We make an unorthodox recommendation for the software upon which we will build our system. Our system has three primary needs in terms of software:

1) a *Front-End* that can handle RESTful requests, interact with our database, and queue delayed jobs
2) a *Worker* that can handle RESTful requests, execute jobs, return results from executed jobs
3) an *image processor*

It is not unusual to use more than one programming language for a web application. We point at the very common practice of using JavaScript for client-side tasks coupled with a more traditional "back end" language such as Ruby, Python, or Java for server-side tasks. In this case, our client is principally a Mobile Client and we are solely concerned with the substance of the "back end" or server-side programming. For this we recommend the use of two separate programming languages and three different frameworks:

- the *Front-End* will use Ruby on Rails
- the *Worker* will use Sinatra, a minimal Ruby wrapper to the Rack webserver
- the *image processor* will use `scikit-image`, a library built on top of numpy/scipy

We are aware that there are full-stack web application frameworks comparable to Rails written in Python – Django and Pyramid to name two. That said, having worked in both, we have found Rails "convention over configuration" ethos to be the fastest, most developer-friendly practice in writing a web application. Furthermore, having worked with Sinatra in Ruby and Flask in Python, both lightweight web application frameworks, we have found the same to be true for Sinatra. In the arena of web application frameworks, the ease of writing human readable domain specific languages, the ease of meta-programming (Ruby's "monkey patching" vs. Python's decorators), and above all the ease of working with databases, an area in which we have found `active-record` to be superior to `sqlalchemy`, Ruby is far and away the obvious choice for building a web application.

The same can not be said for computation. SciRuby is a *very* late entrant to the computation game, having been brought into creation at the same time as, in our humble opinion, numpy/scipy were supplanting MatLab as *the* library to use for matrix algebra. Beyond this, we have a very specific need in terms of computation. We are processing images. While SciRuby would give us the abstraction to BLAS and Lapack we would need to perform our image processing, we would need to implement our methods from scratch. numpy and

scipy offer the same abstractions but beyond this, image processing is a solved problem. An extension library in its 12th major revision called `scikit-image` provides simple, well-documented methods that are open-sourced, based on best practices, and freely available.

the Front-End

Our application is principally concerned with five discrete classes of object – `User`, `Image`, `ImageType`, `JobType`, and `Job`. We use the Rails `generator` command line tool to build a `scaffold` for each class. A Rails `scaffold` for a given class of object consists of a model, a database migration for that model, a controller to manipulate it, views to view and manipulate the data, and a test suite for each of the above. We would run the following commands in order to build the scaffolds for our five objects. The `active-record` gem at the core of the Rails application will provide sufficient abstraction to interact with any of the most common relational database offerings – MySQL, PostgreSQL, MariaDB, Aurora. We will use PostgreSQL in our system because of its reputation for fault tolerance and because we will make specific use of its `array` data type[17].

[17] http://blog.plataformatec.com.br/2014/07/rails-4-and-postgresql-arrays/

The scaffold creates three fields implicitly – a primary key, `id`, and two date-time fields, `created_at` and `updated_at`. Foreign key relationships are created in the model using the methods `has_one`, `has_many`, `belongs_to`, and `through`.

We generate our scaffold and then update the created `app/models/<class>.rb` file to reflect the associations between our data. First the `User` class,

```
$ rails generate scaffold User \
    username:string \
    email:string \
    encrypted_password:string \
    authentication_token:string \
    jobs_count: integer \
    image_count: integer
```

```
# app/models/user.rb
class User < ActiveRecord::Base
  has_many :images
  has_many :jobs
end
```

then the ImageType class,

```
$ rails generate scaffold ImageType \
    type_name:string
```

```
# app/models/image_type.rb
class Image < ActiveRecord::Base
  validates :type_name, presence: {
    message: 'A name must be assigned to the image type.'}
end
```

the Image class,

```
$ rails generate scaffold Image \
    status:string \
    s3_key:string \
    image_type:references \
    user:references \
    original_id:integer
```

```
# app/models/image.rb
class Image < ActiveRecord::Base
  belongs_to :user, counter_cache: true

  validates :user, presence: {
    message: 'A user must be assigned to the image'}
end
```

the JobType scaffold, for which we don't need to modify the model,

```
$ rails generate scaffold JobType \
    job_name:string \
    language:string
```

and the Job scaffold, for which we must also modify the migration.

```
$ rails generate scaffold Job \
    status:string \
    environment:string \
    image_url:string \
    job_type:text \
    user:references \
    image:references
```

```ruby
# app/models/job.rb
class Job < ActiveRecord::Base
  has_one :histogram
  has_one :image
  belongs_to :job_type
  belongs_to :user

  validates :user, presence: {
    message: 'A user must be assigned to the job'}
end

class CreateJobs < ActiveRecord::Migration
  def change
    create_table :jobs do |t|
      t.string :status
      t.string :environment
      t.string :image_url
      t.references :user, index: true, foreign_key: true
      t.references :image, index: true, foreign_key: true
      t.text :job_type, array: true, default: []

      t.timestamps null: false
    end
  end
end
```

Finally, we run the generated database migrations to update the database with the schema we have just created. To do this we use the Ruby command line tool, `rake`[18] and a defined procedure included with the Rails framework `db:migrate`.

```
$ rake db:migrate
```

[18] I think of `rake` as Ruby `make`; to define commands we use a `Rakefile` much like `make` uses a `Makefile`

Application Routes

Using the `rails generate scaffold` will continue to pay dividends as we build out the architecture of our Front-End. For example, should we wish to know the current endpoints for all routes in our application, we have a simple `rake` task for that. In the interest of brevity we will not hear display those routes associated with Views. Strictly speaking the principal client to our application is a Mobile Client. We think that the views generated by rails are "free" and may be useful to system administrators. They should be appropriately concealed behind administrator access and may for the time being be otherwise ignored.

```
$ rake routes
```

Map of RESTful API endpoints

Verb	URI Pattern	Controller#Action
GET	/api/users.json	api/users#index
POST	/api/users.json	api/users#create
GET	/api/users/:id.json	api/users#read
PUT	/api/users/:id.json	api/users#update
DELETE	/api/users/:id.json	api/users#destroy
GET	/api/image_types.json	api/image_types#index
POST	/api/image_types.json	api/image_types#create
GET	/api/image_types/:id.json	api/image_types#read
PUT	/api/image_types/:id.json	api/image_types#update
DELETE	/api/image_types/:id.json	api/image_types#destroy
GET	/api/images.json	api/images#index
POST	/api/images.json	api/images#create
GET	/api/images/:id.json	api/images#read
PUT	/api/images/:id.json	api/images#update
DELETE	/api/images/:id.json	api/images#destroy
GET	/api/job_types.json	api/job_types#index
POST	/api/job_types.json	api/job_types#create
GET	/api/job_types/:id.json	api/job_types#read
PUT	/api/job_types/:id.json	api/job_types#update
DELETE	/api/job_types/:id.json	api/job_types#destroy
GET	/api/jobs.json	api/jobs#index
POST	/api/jobs.json	api/jobs#create
GET	/api/jobs/:id.json	api/jobs#read
PUT	/api/jobs/:id.json	api/jobs#update
DELETE	/api/jobs/:id.json	api/jobs#destroy

You will no doubt notice a common pattern amongst our endpoints. Each class has five endpoints, each associated with a method written in an associated Rails controller file.

api/<class>s#index This method will return a JSON object containing the last n instances of that type added to the database. n could optionally be passed to the endpoint as a query parameter e.g. `?count=20`. Will return success code `200 OK` and the JSON object.

api/<class>s#create This method will create a new instance of the type and save it to the database. A successful request will be accompanied by a properly formatted JSON object included all required attributes. Will return success code `201 CREATED` and a JSON object describing the newly created instance.

api/<class>s#read This method will return a JSON object containing a single instance of that type. The instance is referenced by the trailing `id` in the url. Will return success code `200 OK` and the JSON object.

api/<class>s#update This method will update an existing instance of that type and save it to the database. The instance is referenced by the trailing `id` in the url. Will return success code `200 OK` and a JSON describing the updated instance.

api/<class>s#destroy This method will destroy an existing instance of that type, removing it from the database. The instance is referenced by the trailing `id` in the url. Will return success code `200 OK` and a JSON object describing the destroyed instance.

Application Controllers

Let us next examine how the Mobile Client might interact with our system using this API. Methods called by these APIs are defined in controller files associated with each object. When we ran the scaffold, a controller was generated for each object using the pattern `app/controllers/<class>s_controller.rb`. Note, however, that these controllers are associated with and designed for rendering views in a browser. We are primarily interested in interacting with our server as a pure API provider and using JSON objects. We have thus created our own set of controllers sitting inside of a module called `API`.

We will use a generator to create these controllers as well.

```
$ rails generate controller api/users
$ rails generate controller api/image_types
$ rails generate controller api/images
$ rails generate controller api/job_types
$ rails generate controller api/jobs
$ tree app/controllers
app/controllers/
|-- api
|   |-- image_types_controller.rb
|   |-- images_controller.rb
|   |-- job_types_controller.rb
|   |-- jobs_controller.rb
|   |-- users_controller.rb
```

Each of the controllers we have defined here will begin as follows:

```
module Api
  class Api::<Class>Controller < ApplicationController
  end
end
```

When we make another statement (re)defining the module `Api` in the next controller file, it will "monkey patch" the additional module definitions into the `Api` module. In this way, we define our API across five files. Our file structure mirrors the RESTful nature of the API itself.

Let's take a look at meta-programming in Ruby. Consider the following:

```
$ ipython
In [1]: class Fib:
  ....:     def bar(self):
  ....:         return 1
In [2]: a = Fib()
In [3]: a.bar
Out[3]: 1
In [4]: class Fib:
  ....:     def baz(self):
  ....:         return 2
In [5]: a = Fib()
In [6]: a.bar

AttributeError: Fib instance
  has no attribute 'bar'
In [7]: a.baz
Out[7]: 2
```

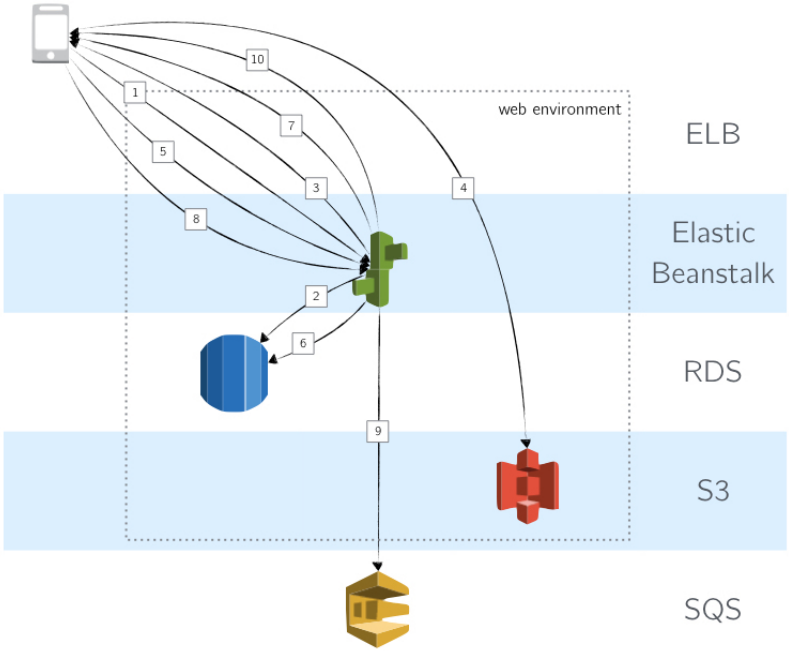
Note that in Python, when we redefined the class it wiped out our earlier definition of that class.

```
$ irb
irb(main):001:0> class Foo
irb(main):002:1>   def bar
irb(main):003:2>     return 1
irb(main):004:2>   end
irb(main):005:1> end
irb(main):006:0> a = Foo.new
irb(main):007:0> a.bar
=> 1
irb(main):008:0> class Foo
irb(main):009:1>   def baz
irb(main):010:2>     return 2
irb(main):011:2>   end
irb(main):012:1> end
irb(main):013:0> a = Foo.new
irb(main):014:0> a.bar
=> 1
irb(main):015:0> a.baz
=> 2
```

Note that in Ruby, when we re-defined the class it added to our earlier definition of the function. This is informally known as "monkey patching" and is one method for meta-programming in Ruby. We will use this pattern while defining our API.

Saving a Photograph

Here we have reproduced just the elements of our minimal system that are involved in the uploading of a single photograph from a Mobile Client.



1. Mobile Client sends photo meta-data
2. Front-End creates photo in database
3. Front-End responds `201`
4. Mobile Client uploads photo to S3
5. Mobile Client sends S3 url to server
6. Front-End updates photo in database
7. Front-End responds `200`
8. Mobile Client sends processing request to server
9. server queues processing job on SQS
10. Front-End responds `201`

The first step in the processing of a photograph is for the Mobile Client to `POST` a JSON object as below to the `/api/photographs.json` endpoint (Step 1)

```
{
  "user_id" : 78901
}
```

JSON sent to `api/photographs#create` (Step 1)

This will call the following method defined in the `photographs_controller` in the Front-End, which creates an object in the database (Step 2)

```
# app/controllers/api/images_controller.rb
module Api
  class Api::ImagesController < ApplicationController
    # POST /api/images.json
    def create
      @image = Image.new(user_id: params[:user_id], status: "created")
      if @image.save
        render json: @image.to_json(only: [:id, :status]), status: 201
      else
        render json: {error: "Image could not be created."}, status: 422
      end
    end
  end
end
```

before responding with this JSON object (Step 3)

JSON sent to Mobile Client (Step 3)

```
{
    "id" : 12345,
    "status" : "created"
}
```

It is beyond the scope of this discussion to provide the mechanism for building the Mobile Client[19]. When the Mobile Client receives a `201 CREATED` response code it begins to upload the photograph as a background process (Step 4). Upon completion, the Mobile Client sends a request to the `api/images#update` controller action e.g. `/api/images/12345` (Step 5).

[19] We might recommend the use of the Github repo, `lifuzu/react-native-uploader-s3`, if the Mobile Client was built using React Native

JSON sent to `api/images#update` (Step 5)

```
{
  "id" : 12345,
  "s3_key" : "image-string",
  "user_id" : 78901
}
```

The Front-End receives this object and executes the following method, updating the photograph in the database (Step 6)

```
# app/controllers/api/images_controller.rb
module Api
  class Api::ImagesController < ApplicationController
    # PUT /api/images/:id.json
    def update
      @image = Image.find(params[:id])
      if @image.update_attributes(s3_key: \
          params[:s3_key], status: "original stored")

        render json: @image.to_json(only: [:id, :status]), status: 200
      else
        render json: {error: "Image could not be updated."}, status: 422
      end
    end
  end
end
```

before responding with this JSON object (Step 7)

JSON sent to Mobile Client (Step 7)

```
{
    "id" : 12345,
    "status" : "original stored"
}
```

Finally, the Mobile Client sends a request to the `api/jobs#create` controller action e.g. `/api/jobs.json` (Step 8).

JSON sent to `api/jobs#create` (Step 8)

```
{
  "image_id" : 12345,
  "user_id" : 78901,
  "job_type" : "grayscale"
}
```

The Front-End will receive this object and executes the following method, queueing a job to be processed by the Worker (Step 9)

```
# app/controllers/api/jobs_controller.rb
module Api
  class Api::JobsController < ApplicationController
    # POST /api/jobs.json
    def create
      @job = Job.new(status: "Queued")
      sqs = Aws::SQS::Client.new
      if sqs.send_message({
        queue_url: ENV["SQS_URL"],
        message_body: {
                        "job_type" : params['job_type'],
                        "image_id" : @image.id,
                        "image_url" : @image.s3_key
                      },
        delay_seconds: 1})
        render json: @job.to_json, status: 201
      else
        render json: {error: "Job could not be created."}, status: 422
      end
    end
  end
end
```

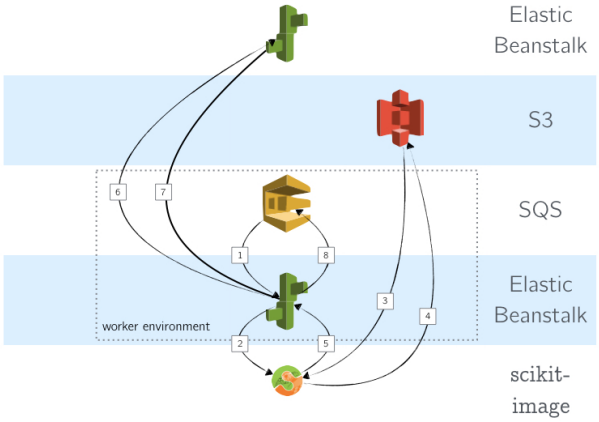
before responding with this JSON object (Step 10)

JSON sent to Mobile Client (Step 10)

```
{
    "id" : 2131,
    "status" : "job queued"
}
```

the Worker

We will explore the Worker component of our system by examining a usage scenario. Here is the typical workflow for a job executed by the Worker.



1. the Worker receive queued job request from SQS (as an API call)
2. the Worker executes `scikit-image` script
3. `scikit-image` grabs photograph for processing from S3 and executes processing
4. `scikit-image` saves result to S3 in a new location; the original photograph remains untouched
5. `scikit-image` returns results of executed job (url of result on S3, status, etc.)
6. the Worker sends JSON of updated job status to the Front-End
7. the Front-End responds `200` to the Worker
8. the Worker responds `200` to SQS

Consider a scenario in which the system receives an input image from a client, then is asked to convert the image to grayscale and compute the histogram of the converted image, storing both results. In order to engage the Worker, the Mobile Client will have uploaded the photograph to S3 and the Front-End will have queued a job of job type ["grayscale", "histogram"]. Let's follow this workflow from the perspective of the Worker.

JSON sent to Worker (Step 1)

```
{
  "job_id" : 231,
  "user_id" : 78901,
  "image_id" : 12346,
  "s3_key" : "http://s3.amazon.com/original-image",
  "job_sequence" : ["process", "grayscale", "histogram"],
  "environment" : ""
}
```

The Worker will be running a simple Sinatra application which will receive this at end endpoint we have configured via Elastic Beanstalk, `/worker`. Here is the Sinatra application[20]. Note that we have made

[20] yes, the entire application

use of a domain specific language to make our worker more readable. Each of the methods `Worker.<method>` is defined in the file `lib/worker.rb`.

```
require 'sinatra'
require 'rest-client'
require_relative 'lib/worker.rb'

post '/worker' do
  request_payload    = Worker.receive_json request

  job_id,         \
    user_id,      \
    image_id,     \
    s3_key,    \
    job_sequence, \
    environment      = Worker.parse_payload request_payload

  output             = Array.new
  job_type           = job_sequence.shift

  if job_type == 'process'
    job_sequence.each do |job|
      if s3_key = Worker.run_job environment, job, s3_key
        output.push([job,s3_key])
      else
        raise "#{job_id} #{job} failed."
      end
    end
  elsif job_type == 'query'
    query_class  = job_sequence.shift
    query_string = environment
    job          = job_sequence.shift
    query_array  = `#{environment} curl #{ENV['FRONT_END_URL']}/api/#{query_class}?$QUERY`
    if s3_key = Worker.run_job environment, job, query_array
      output.push([job,s3_key])
    else
      raise "#{job_id} #{job} failed."
    end
  elsif job_type == 'search'
    query_class  = job_sequence.shift
    query_string = environment
    job          = job_sequence.shift
    query_array  = `#{environment} curl #{ENV['FRONT_END_URL']}/api/#{query_class}?$QUERY`
```

```
    if search_array = Worker.run_job environment, job, query_array
      output.push([job,search_array])
    else
      raise "#{job_id} #{job} failed."
    end
  end

  response = Worker.assemble_response job_id, user_id, image_id, output_images
  status 200
  body response
end
```

The method `Worker.run_job` passes environment variables, a job title, and an s3_key to `scikit-image` (Step 2). The method `Worker.assemble_response` builds a JSON object to send back to the Front-End (Step 6).

```
# lib/worker.rb
module Worker
  def Worker.run_job environment, job, asset=''
    s3_key = `#{environment} python #{job}.py #{asset}`
  end
end
```

Since we iterate over an array containing job types, this has the effect of

1. opening a new `bin/sh`, running
   `$ python grayscale.py original-image`
   and returning the url of the resulting `image_url` e.g.
`http://s3.amazon.com/grayscale-image`
2. opening a new `bin/sh`, running
   `$ python histogram.py grayscale-original-image`
   and returning the url of the resulting `image_url` e.g.
`http://s3.amazon.com/histogram`

Note that we have made use of a domain specific language to make our script more readable. We have abstracted our `boto` methods into a file `boto_worker`.

```
#boto_worker.py
import boto3

module py_worker:
    def s3_download(s3_bucket, key):
        local_file = \
          "tmp/s3_download/{0}".format(key)
        s3        = boto3.resource('s3')
        bucket    = s3.Bucket(s3_bucket)
        bucket.download_file(key, local_file)

    def s3_upload(s3_bucket, key):
        local_file = \
          "tmp/s3_upload/{0}".format(key)
        s3        = boto3.resource('s3')
        bucket    = s3.Bucket(s3_bucket)
        bucket.upload_file(local_file, key)
```

Steps 3, 4, and 5

```
# grayscale.py
from os             import  environ
from py_worker      import  s3_download, s3_upload
from skimage        import  io
from skimage.color  import  rgb2gray
from sys            import  argv

s3_key           = argv[1]
grayscale_s3_key = "grayscale-{0}".format(s3_key)
original_image   = s3_download("tmp/s3_download/{0}".format(s3_key))

image_in_memory  = io.imread(original_image)
grayscale_image  = rgb2gray(image_in_memory)

io.imsave("tmp/s3_uploads/{0}".format(url))
s3_upload(environ['S3_BUCKET'], grayscale_s3_key)

return grayscale_s3_key
```

```
# histogram.py
from   numpy            import  save
from   os               import  environ
from   py_worker        import  s3_download, s3_upload
from   skimage.exposure import  histogram
from   sys              import  argv

s3_key           = argv[1]
histogram_s3_key = "histogram-{0}".format(s3_key)
original_image   = s3_download("tmp/s3_download/{0}".format(s3_key))

image_in_memory     = io.imread(original_image)
histogram_im_memory = histogram(image_in_memory)

save(histogram_s3_key, histogram_im_memory)
s3_upload(environ['S3_BUCKET'], histogram_s3_key)

return histogram_s3_key
```

Running Queries

The beauty of Rails and `active-record` is the ease with which we can query our system. Using the `<class>.find` method we to implement request methods at the end of our various RESTful endpoints. To extract the histograms of the current week for a single user, we would make a request against the `api/images#index` controller. This would contain a JSON object containing an array of `s3_keys` to `histograms`. Our controller would then contain

At the endpoint `/api/images.json` with query parameters `user_id=12345&days=7&image_type=histogram`

```
# app/controllers/api/images_controller.rb
module Api
  class Api::ImagesController < ApplicationController
    # GET /api/images.json
    def index
      if (params.keys.include? 'n')
        n = params[:n]
      else
        n = 20
      end

      if (params.keys.include? 'user_id')
        @images = Image.find(user_id: params[:user_id])
      elsif n != 'all'
        @images = Image.find().last(n)
      else
        @images = Image.find()
      end

      if (params.keys.include? 'image_type')
        @images = Image.where(image_type: params[:image_type])
      end

      if (params.keys.include? 'days')
        if params[:days] == 'today'
          @images = Image.where("created_at >= ?", Time.zone.now.beginning_of_day)
        n_days = params[:days].to_i
        @images = Image.where(created_at: n_days.days.ago..Time.now)
      end

      render json: @images.to_json, status: 200
    end
  end
end
```

Queries and Jobs

Some queries may require processing after querying the database. In these cases, we propose calling the job, then having the job make the query.

To extract the median histogram of the current day for all users we hit the `api/jobs#create` controller action

```
{
  "job_sequence" : ["query", "histogram", "median"],
  "environment" : "QUERY='n=all&days=today&image_type=histogram'"
}
```

The Worker issues a query to the Front-End which returns a JSON object as a **string**

```
{
  "histograms" : [
    { "user_id" : 12,
      "s3_key" : "histogram-grayscale-12"},
      ...
  ]
}
```

then executes

```
#! /bin/sh
QUERY='n=all&days=today&image_type=histogram' \
  python median.py \
  "{\n \"histograms\" : ..."
```

```
# median.py
from   json       import  loads
from   datetime   import  date
from   numpy      import  array, load, median, save
from   os         import  environ
from   py_worker  import  s3_download, s3_upload
from   sys        import  argv

query_string = argv[1]
query_dict  = loads(query_string)

for hist in query_dict['histograms']:
  s3_download("tmp/s3_download/{0}".format(hist['s3_key']))

np_histograms    = [
                    load("tmp/s3_download/{0}.py".format(hist['s3_key'])) \
                    for hist in query_dict['histograms']
                   ]
median_histogram = median(np_histograms, axis=0)

median_s3_key    = "median-{0}".format(str(datetime.date.today()))
save(median_s3_key, median_histogram)
s3_upload(environ['S3_BUCKET'], median_s3_key)

return median_s3_key
```

To return the n users with the most similar histograms to a given histogram `api/jobs#create` controller action again with the following JSON

```
{
  "job_sequence" : ["query", "histogram", "nearest_neighbor"],
  "environment" : "QUERY='n=all&image_type=histogram' USER=123 TARGET_S3_HIST=123-histogram.npy"
}
```

The Worker issues a query to the Front-End which return a JSON object as a **string**

```
{
  "histograms" : [
    { "user_id" : 12,
      "s3_key" : "histogram-grayscale-12"},
      ...
  ]
}
```

then executes

```
#! /bin/sh
QUERY='n=all&image_type=histogram' USER=123 TARGET_S3_HIST=123-histogram.npy K=10 \
  python nearest_neighbor.py \
  "{\n \"histograms\" : ..."
```

Note that we have made use of the `scikit-learn` library `neighbors.NearestNeighbors` in order to do an unsupervised fit on our target_histogram.

```
# nearest_neighbor.py
from   json              import  dumps, loads
from   datetime          import  date
from   numpy             import  append, array, load, median, save
from   os                import  environ
from   py_worker         import  s3_download, s3_upload
from   sklearn.neighbors import  NearestNeighbors
from   sys               import  argv

query_string = argv[1]
user         = environ['USER']

target_histogram = s3_download("tmp/s3_download/{0}".format(environ['TARGET_S3_HIST']))

query_dict  = loads(query_string)

for hist in query_dict['histograms']:
  s3_download("tmp/s3_download/{0}".format(hist['s3_key']))

np_histograms = [
                 load("tmp/s3_download/{0}.py".format(hist['s3_key'])) \
                 for hist in query_dict['histograms']
                ]

nbrs      = NearestNeighbors(n_neighbors=2, algorithm='ball_tree').fit(np_histograms)
trgt_nbrs = nbrs.kneighbors([target_histogram], environ['K'] , return_distance=False)

return dumps(query_dict[trgt_nbrs])
```

Development Timeline

Minimal System

the Front-End

Ruby on Rails application on Elastic Beanstalk with ELB. Time to minimum viable product: Two weeks

the Worker

Sinatra application on Elastic Beanstalk with SQS. Time to minimum viable product: One week

Scikit

Scikit scripts on Elastic Beanstalk. Time to minimum viable product: One week

End-to-End

Full end-to-end system with unit, functional, integration, and acceptance test coverage. Additional time to minimum viable product: Three weeks

Total time: Six Weeks

Recommend three months of alpha testing at this level.

Scaled System

Deploying App Instances

Adding app instances to Elastic Beanstalk across availability zones and regions. Time to minimum viable product: Three weeks

ElastiCache System

Configuring ElastiCache as job queue system. Time to minimum viable product: Three weeks

Lambda

Configuring Scikit to run on Lambda. Time to minimum viable product: Two weeks