Capstone Project

Due: Wednesday, June 3, 2020 at 5:00pm CDT

Submit: https://classroom.github.com/assignment-invitations/a019a46cda6ea5fe00f57e8a891de796

Total points: 150 (+ up to 23 bonus points)

Objectives

Gain experience with deployment approaches and techniques for scaling and testing cloud applications, as well as integrating external SaaS providers.

Background

Our course concludes with the build-out of a fully functional software-as-a-service for genomics annotation. We have been working on components of this Genomics Annotation Service (GAS) that make use of various clouds services running on Amazon Web Services. In this final project we will make the system a little more functionally complete, add capabilities for scaling the application, and integrate with an external cloud service for payments processing.

Key Functions

When completed, the GAS will allow a user to perform the following functions:

- **Log in (via Globus Auth) to use the service** -- Some aspects of the service are available only to registered users. Two classes of users will be supported: Free and Premium. Premium users will have access to additional functionality, beyond that available to Free users.

- **Upgrade from a Free to a Premium user** -- Premium users will be required to provide a credit card for payment of the service subscription. The GAS will integrate with Stripe (www.stripe.com) for credit card payment processing. No real credit cards are required for this project :-) ...we will use only a test credit card (4242424242424242) provided by Stripe.

- **Submit an annotation job** -- Free users may only submit jobs of up to a certain size. Premium users may submit any size job. If a Free user submits an oversized job, the system will refuse it and will prompt the user to convert to a Premium user.

- **Receive notifications when annotation jobs finish** -- When their annotation request is complete, the GAS will send users an email that includes a link where they can view the log file and download the results file.

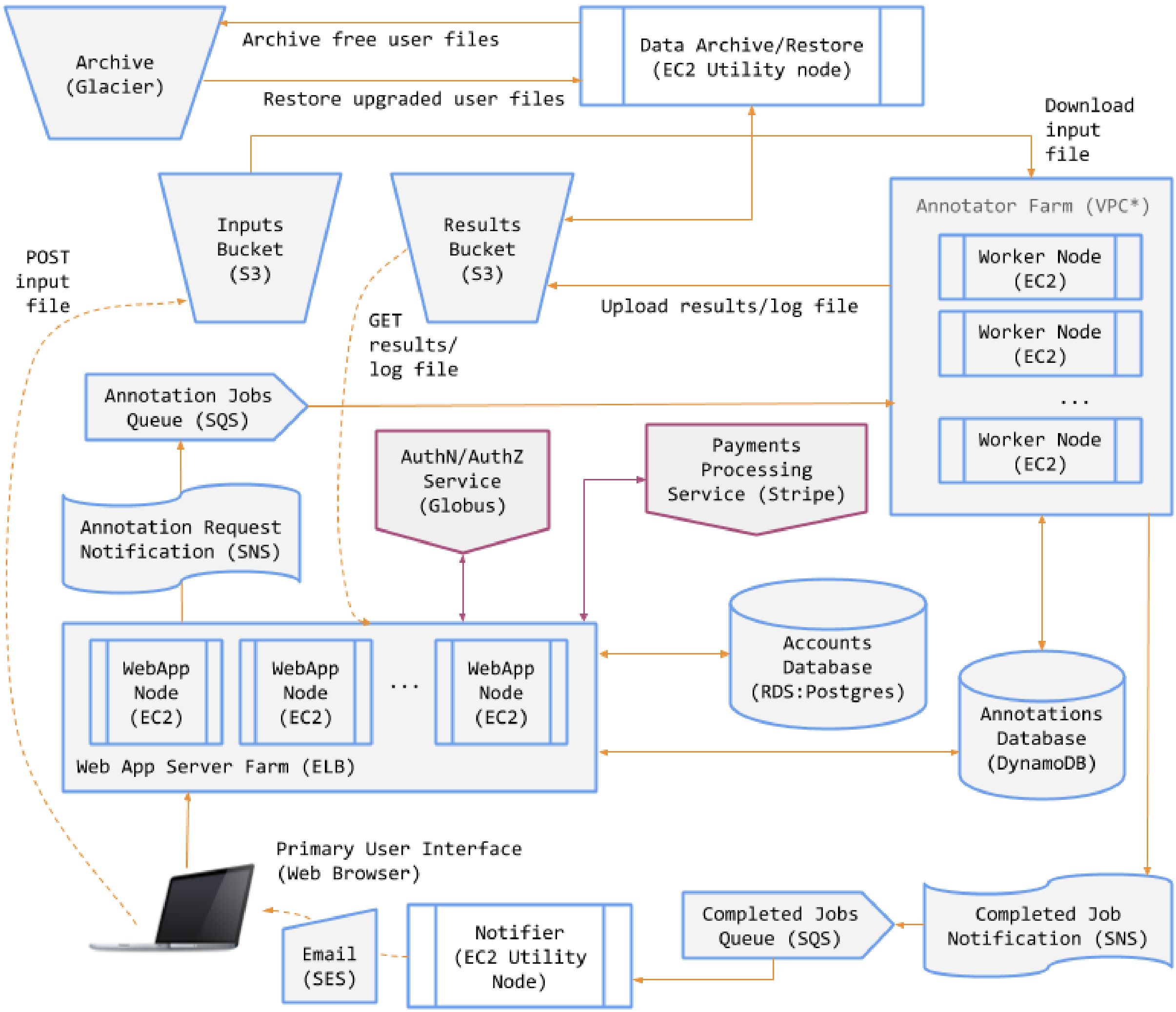
- **Browse jobs and download annotation results** -- The GAS will store annotation results for later retrieval. Users may view a list of their jobs (completed and running). Free users may download results up to 10 minutes after their job has completed; thereafter their results will be archived and only available to them if they convert to a Premium user. Premium users will always have all their data available for download.

System Components

The GAS will comprise the following components:

- An object store for input files, annotated (result) files, and job log files.
- A key-value store for persisting information on annotation jobs.
- A low cost, highly-durable object store for archiving the data of Free users.
- A relational database for user account information.
- A service that runs AnnTools for annotation.
- A web application for users to interact with the GAS.
- A set of message queues and notification topics for coordinating system activity.

The diagram below shows the various GAS components/services and interactions:



GAS Scalability

We anticipate that the GAS will be in very high demand (since it's a brilliant system developed by brilliant students), and that demand will be variable over the course of any given time period. Hence, the GAS will use elastic compute infrastructure to minimize cost during periods of low demand and to meet expected user service levels during peak demand periods. We will build elasticity into two areas of the GAS:

1. On the front end, the web application will be delivered by multiple servers running within a load balancer. All requests will be received at a single domain name/IP address, namely that of the load balancer. The load balancer will distribute requests across a pool of *identically configured, stateless*, web servers running on EC2 instances. At minimum, the load balancer will have two web server instances running across two availability zones, providing capacity and ensuring availability in the event of failure. If demand on either web server exceeds

certain thresholds, the GAS will automatically launch additional web servers and place them in the load balancer pool. When demand remains below a certain threshold for a specified period of time, the GAS will terminate the excess web servers.

2. On the back end, the annotator service will be delivered by multiple servers (optionally running within a separate virtual private cloud). At minimum this pool of so-called “worker nodes” will contain two nodes (EC2 instances). Additional instances will be launched and added to (or removed from) the worker pool, based on the number of requests in the job queue. The annotator servers store the state of running jobs locally (as implemented in homework assignments) -- in this sense they are not stateless like the web app servers. If a job fails to complete it will leave the system in an inconsistent state, but it’s a state from which we can recover relatively easily.

The GAS Today and Tomorrow

Over the course of the past few weeks we've built many of the GAS components. To get to our final destination, we will do the following:

- Create a robust (multi-threaded) and secure runtime environment for our web server.
- Add user registration, authentication, and authorization mechanisms.
- **Add mechanisms for handling completion and notification of annotation jobs.**
- **Enhance the listing of a user’s annotation jobs.**
- **Add ability to download results and view the annotation log file.**
- Integrate credit card payment functionality for signing up premium users.
- **Archive Free users’ data (to AWS Glacier), and restore it if they convert to Premium.**
- **Add load balancer and scaling rules to the web server farm.**
- **Add scaling rules to the annotator farm.**
- Evaluate GAS performance under heavy/variable load.

In addition, we will add some scripts to automate deployment of our entire execution environment.

Note: You are responsible only for the items in bold above. The remainder are already completed since we’re operating under tighter time constraints this quarter. In addition, we are providing you with all the HTML templates you will need so you can focus on core system functionality.

Preparation

Using the Enhanced Flask Web Framework

We want the GAS to look somewhat appealing so we are going to use an enhanced version of the Flask framework for this project. Here's how to get the framework:

```
git clone https://github.com/mpcs-cc/gas-framework.git /home/ubuntu/gas
```

The enhanced framework provides styled web pages using the [Bootstrap CSS library](https://getbootstrap.com) (v3.3). You don't need to change anything related to styling (and you will not get additional credit for gorgeous UI design), but if you want to experiment with a different look and feel, have at it!

Modular UI Templates: Our web app now has a more modular template structure so every page in your application shares a header, footer, and other HTML elements. In order to use this structure all your template (`.html`) files must reside in the `templates/` directory, off the root directory of your application. And, for those that previously added some styling or other elements to your upload form, you can remove those header/footer HTML elements from your template. Our templates will look like this from now on (https://gist.github.com/vasv/13fc3ea5386b152fd503596d96caa078):

```
{% extends "base.html" %}  <!-- must include this -->
{% block title %}Annotations{% endblock %}  <!-- optional -->
{% block body %}  <!-- must include this -->
  {% include "header.html" %}  <!-- must include this -->
  <div class="container">
    <div class="page-header">
      <h1>My Page Title</h1>
    </div>
    Your page content/form/etc. goes here....
    <div class="row">
      Use <div class="row">...</div> to enclose page elements
      and combine it with <div>s that specify column widths
      to lay out your page. Bootstrap splits pages into 12
      equal-width columns, e.g. if you want content to
      span one third of the page, enclose it in
      <div class="col-md-4"> .... </div>
      See: https://getbootstrap.com/docs/3.3/css/#grid
    </div>

  </div> <!-- container -->
{% endblock %}  <!-- must include this -->
```

Your source code for the web server will be added to `views.py`. This file currently has code for the home page and a few static error pages. You must add/move your existing code for uploading files, sending notifications to SNS, etc. to the `views.py` file (see Ex. 2 below).

Security: The GAS needs to be more secure than our web server to-date. From this point on the web server will respond only to HTTPS requests. We require HTTPS for things like authenticating users. Many websites/apps use secure connections for a subset of their pages, but it is considered a best practice to use HTTPS throughout the entire site/app, so we'll get into this good habit now!

Robust App Server: We will also substitute the default Flask WSGI server with the Gunicorn server which is a well-tested, multithreaded WSGI server suitable for production use. The standard Flask WSGI server is adequate for development but not for a production system (and it does not support HTTPS) so we have replaced it with something more robust. In addition, our application will now listen on port **4433** instead of 5000.

App Configuration: The GAS now uses the Flask `config` attribute to set and access various configuration values throughout the application. You can add and change configuration values in `config.py`. Then you can access these in your code using standard dictionary notation; for example:

```
app.config[“AWS_S3_INPUTS_BUCKET“]
```

will return the value “`gas-inputs`”.

Logging: If needed, the GAS also includes a logger that can be used to log events to a file and to the console. You may write to the log like so:

```
app.logger.info("Informational log message...").
```

User Management/Authentication and Authorization in the GAS

The GAS uses the [Globus Auth] service to implement user authentication and authorization. Globus Auth is an identity and access management service that allows users to access our application using an existing identity from hundreds of identity providers, including UChicago. Despite its ease of use, we decided to spare you the added effort of working with Globus Auth so it's already integrated into the application. All you need to do is log into the GAS with CNetID!

- You can access an authenticated user's attributes by calling the `get_profile()` function and passing it the user ID. The user ID is stored in the session and you can access it as: `session.get('primary_identity')`. Then use this to return the user's profile: `profile = get_profile(identity_id=session.get('primary_identity'))`

- The user profile has the following attributes:
    - `profile.name` - returns the user's name
    - `profile.email` - returns the user's email address
    - `profile.institution` - returns the user's institution (not used in our project)
    - `profile.role` - returns the user's role; we deal with only two roles in the GAS:
        - `free_user`: has access to pages/data that require authentication, e.g. the /annotations page.
        - `premium_user`: has access to pages/data only available to Premium users

- Profile attributes are stored in the session, so you can access them in your code as `session[<attribute_name>]`. The session dictionary is also available in your templates and may be accessed via `{{ session[<attribute_name>] }}`.

- You can restrict access to a route/function in the GAS by requiring that a user be authenticated. All you need to do is decorate the function with `@authenticated`. This will redirect unauthenticated users to Globus to authenticate.

- Similarly, if you want to restrict access to a function/page only to premium users, use the `@is_premium` decorator. This will check that the user is authenticated and that their role is set to `premium_user`, otherwise it will redirect them to the login page.

Tagging Instances

You will be using three types of instances: web server, annotator, and utilities. Please tag as follows:

- Web server instances: <CNetID>-gas-web
- Annotator instances: <CNetID>-gas-ann
- Utility instances: <CNetID>-gas-util

Exercises

1. Preparing the User Database and Running the GAS

Runtime environment configuration parameters are stored in OS environment variables, and read into the Config object at startup. These include important parameters like the port that the application listens on. Environment variables must be stored in a file named `.env` that is sourced when the GAS app runs.

(a) If you haven't already done so, SSH into your instance and clone the GAS framework repo:
`git clone https://github.com/mpcs-cc/gas-framework.git /home/ubuntu/gas`

(b) Create a file named `/home/ubuntu/gas/web/.env` and include the variables below, **replacing the highlighted values** (https://gist.github.com/vasv/07a0477ee2a6d096ca80b231a3b52ff1):

```
export GAS_SETTINGS="config.ProductionConfig"
export GAS_HOST_IP="<CNetID>.ucmpcs.org"
export GAS_HOST_PORT="4433"
export GAS_APP_HOST="0.0.0.0"
export GAS_LOG_FILE_NAME="gas.log"
export ACCOUNTS_DATABASE_TABLE="<CNetID>_accounts"
export GUNICORN_WORKERS="2"
```

(b) We will use a Postgres database running in RDS to store user account information. Create and source the .env file above, and then run the following commands (and don't forget to "`workon mpcs`"):

```
python manage.py db init
python manage.py db migrate
python manage.py db upgrade
```

This creates your user accounts table; we use SQLAlchemy and Alembic to access/manage the user database. These are very widely used Python packages so take a little time to read up about them if you're interested, but the project does not require you to access the database directly. Note: you only need to run these commands once to initialize the database; thereafter all your instances will be able to access the database without further configuration.

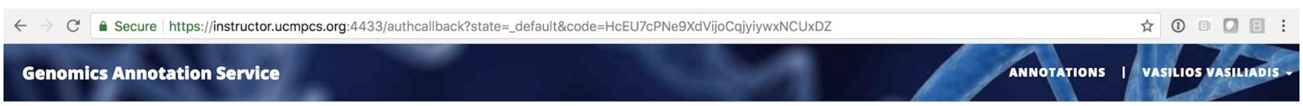
(c) Update `/home/ubuntu/gas/web/config.py` to reflect your CNetID—it's used in a number of places (e.g. the SQS queue name); then update your code to use the various configuration parameters listed in `config.py`.

You must no longer use hardcoded values of any sort within your code. Any hardcoding will result in points being deducted. As an aside, while you're updating `config.py` note the mechanism used to get credentials for the Postgres database. The admin username/password are stored in the [AWS Secrets Manager]() service, so you don't have to deal with managing them. This service can also conveniently rotate credentials for us based on user-specified policies.

(d) Copy the SSL certificate files required to enable HTTPS on your web server. These files are stored on S3 in the `mpcs-resources` bucket (prefixed by `/ssl`) and must be placed in the `/usr/local/src/ssl` directory on your instance. You can copy them by running:

```
aws s3 cp s3://mpcs-resources/ssl /usr/local/src/ssl --recursive
```

(e) Using the `mpcsdomain` utility, create a subdomain for your GAS at <CNetID>.ucmpcs.org. Now we're ready to start the web server for the GAS. Just do `./run_gas.sh`. Test your new setup by going to **https**://<CNetID>.ucmpcs.org:**4433**. (Note that we're now using HTTPS, not HTTP). You should see the GAS home page. You should be able to log in with your UChicago CNetID and view a blank "My Annotations" page:



When running the GAS using Gunicorn, the output is written to a log file - by default this is stored in `/home/ubuntu/gas/log/gas.log`. If you would like to see console output (e.g. to help with debugging) then you must run the GAS using `./run_gas.sh console`. This will be very useful while building/debugging your app.

IMPORTANT: Make sure your GAS app is working correctly before proceeding! If you cannot access the app and authenticate successfully do not continue with other exercises before resolving. Note: you may need to set the file mode on `run_gas.sh` to 755 to make it executable.

2. (9 points) Migrate and update existing code.

(a) (2 points) Move your existing code for uploading an input file to S3 and handling the redirect (saving the job to DynamoDB and publishing a notification for the annotator) into `views.py`.

(b) (4 points) Up until now we've hardwired a username for the annotation request. Modify your code so that it uses the authenticated user's ID and include that in the annotation database, as well as anywhere else where the username may be needed. Only authenticated users should be allowed to use the system; **beyond this point in the project, the only thing that unauthenticated users can do is view the home page**. See above for how to access the user's ID in the session.

(c) (3 points) Rename `hw5_annotator.py` to just `annotator.py`. Rename `hw5_run.py` to `run.py`. Move all hardcoded values such as bucket names, queue names, database names, to the file `ann_config.ini` and access that in your `annotator.py` and `run.py` scripts via Python's ConfigParser. You can find an example of how to do this by looking in any of the utility scripts included in /util, e.g. `/util/archive/notify.py` and `notify_config.py`.

BONUS POINTS:

- **3 points**: Implement a check for file size when a Free user is requesting an annotation. When a Free user tries to upload a file larger than 150KB, display a message and prompt them to subscribe. This aligns with the five sample input VCF files provided (the two free files are both below this limit). Your message may be displayed in a popup window or as a Flask "flash" message (see the `profile()` function in `auth.py` for an example).
  Note: This will require using Javascript.

Optional for Spring 2020 Quarter: Modify your annotator so that it doesn't repeatedly poll for job request messages. Instead, convert it to a simple Flask app with one route handler that acts as a webhook; then register your webhook as a subscriber to your job request SNS topic. This way, when a new job is submitted, SNS will push the notification to your webhook, triggering the annotation.

3. (16 points) Add mechanisms for handling notification of annotation jobs

We need some way to notify a user when the job is complete. We will do this by publishing a notification to an SNS topic (with a subscribed SQS queue, as we have for job requests) and running a separate Python script that sends emails to users.

(a) (1 point) As you did in HW5, create an SNS topic for results notification. The topic name should be of the form `<CNetID>_job_results`.

(b) (1 point) As you did in HW5, create an SQS queue to store results notification messages.

(c) (1 point) Subscribe your SQS queue to the SNS topic (again, just like you did in HW5).

(d) (3 points) Modify `run.py` and modify it so that it publishes a notification to this topic when the job is complete, i.e. after the database has been updated and the results and log files have been copied to S3.

(e) (10 points) Write a new Python script called `notify.py` that polls the SQS results queue, and sends an email to the user when their job is complete. This script will be similar to the `hw5_annotator.py` in HW5, except that it sends an email message instead of launching an annotation job. Note that this will not be a Flask app, just a plain ’ol Python script.

In order to send email, we will use the Amazon SES service (to see how SES works review http://boto3.readthedocs.io/en/latest/reference/services/ses.html#ses). I’ve provided a convenience function `send_email_ses()` in `helpers.py` that you can call to send email. The notification email must include the job ID, linked to the page that displays the job details (see Exercise 5 below). NOTE: To send SES emails without validation, **use <CNetID>@ucmpcs.org as the sender email**. You will need to configure this and the name of your accounts database in `notify_config.ini`.

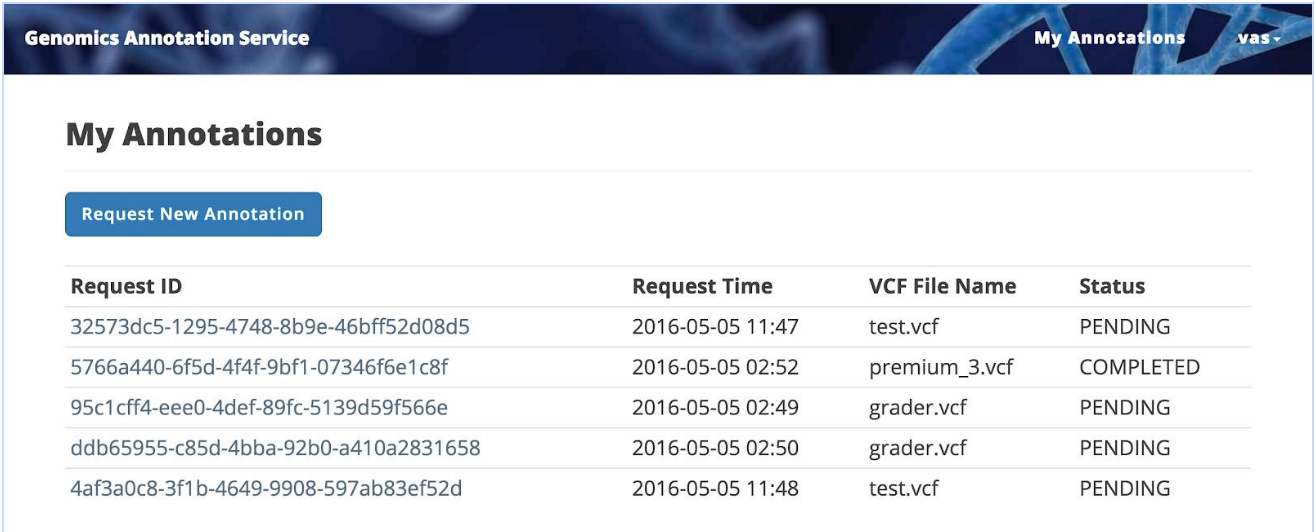
<u>Run the `notify.py` script on a separate instance</u> and tag this instance as `<CNetID>-gas-util` because we will run other utility scripts here also.

Using configuration files: Since `notify.py` is not a Flask application you will need to use a separate configuration file, `notify_config.ini`, to store settings. A shell `notify_config.ini` is included in the framework. You can access configuration settings as a dictionary, e.g. `config['aws']['AwsRegionName']`. See the `helpers.py` file for examples.

Important reminder: Do not hardcode settings in your utility scripts; add them to the appropriate config file and access as shown in `helpers.py`.

4. (6 points) Display a list of all the user’s jobs

Create a page that displays a list of all the jobs submitted by a user when the user navigates to `/annotations`; example below:



Request ID	Request Time	VCF File Name	Status
32573dc5-1295-4748-8b9e-46bff52d08d5	2016-05-05 11:47	test.vcf	PENDING
5766a440-6f5d-4f4f-9bf1-07346f6e1c8f	2016-05-05 02:52	premium_3.vcf	COMPLETED
95c1cff4-eee0-4def-89fc-5139d59f566e	2016-05-05 02:49	grader.vcf	PENDING
ddb65955-c85d-4bba-92b0-a410a2831658	2016-05-05 02:50	grader.vcf	PENDING
4af3a0c8-3f1b-4649-9908-597ab83ef52d	2016-05-05 11:48	test.vcf	PENDING

The page should display a table that contains four columns: the job ID, the date/time submitted, the input file name and the job status. The job ID should be hyperlinked to a page that shows all the details of the job (see Exercise 5). **Ensure you only return jobs for the currently authenticated user**. An HTML template, `annotations.html`, is already provided for this purpose.

Note:

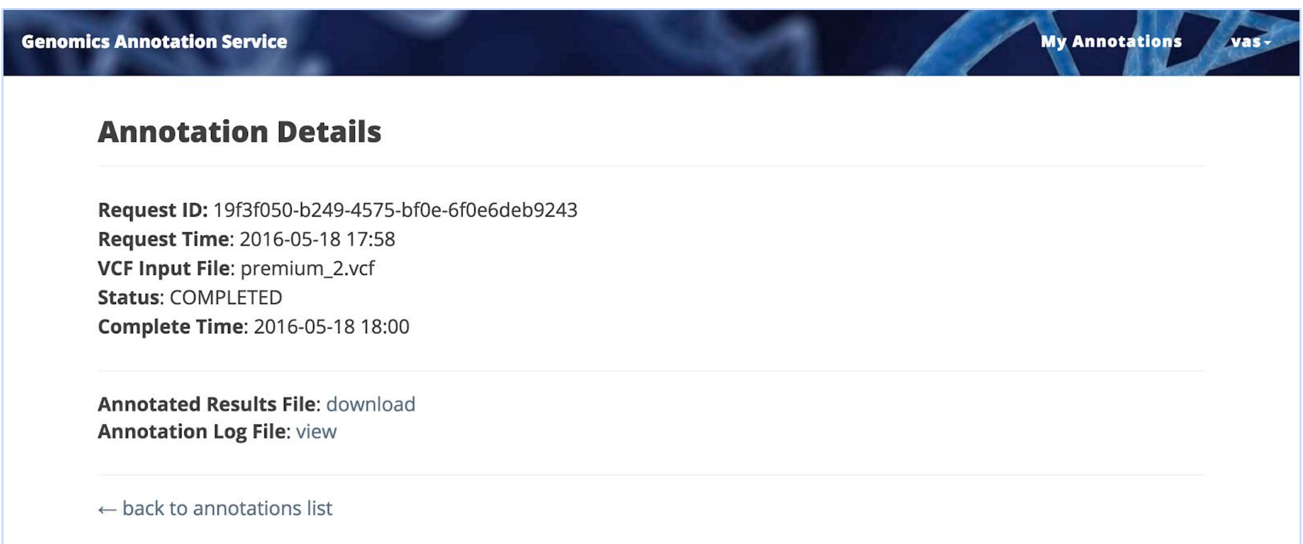
1. *You must use **query** functions (**not scans**) when retrieving items from DynamoDB*.
2. *“Request Time” must display the date in the **local timezone** (CST/CDT), not GMT or UTC.*

5. (6 points) Display job details

Create a page that displays the details of a requested job. The page should be rendered in response to a request like: `https://<CNetID>.ucmpcs.org:4433/annotations/<job_id>`. It should list the following attributes, retrieved from the annotations database:

> **Request/Job ID**: <UUID of job>
> **Status**: <job status> (i.e. "RUNNING", "COMPLETED", etc.)
> **Request Time**: <date/time when job submitted>
> **Input filename**: <input file name> (a hyperlink for downloading the input file)
> **Complete Time**: <date/time when job completed> (hide from display if job is not complete)
> **Results filename**: download (a hyperlink for downloading the results file; see Ex. 6 below; this field should be hidden from display if the job is not complete)
> **Log filename**: view (a hyperlink for displaying the log file, see Ex. 6 below; this field should be hidden from display if the job is not complete).

The page should look like this:



Once again, a template file called `annotation_details.html` is already provided. You will need to match the template variables with the arguments you pass to the `render_template()` function. Don't forget to check that the requested job ID belongs to the user that is currently authenticated; if it does not you must display an appropriate error, e.g. "Not authorized to view this job".

Ensure that you are converting epoch times to human-readable form in both Ex. 4 and 5 and that all displayed dates are in the local timezone.

6. (8 points) Provide a means for users to download results and view log files

When the user clicks a filename hyperlink in the job detail listing (see 5. above) do the following:

- If the user clicks on the results file name link, download the file from S3 to the user's laptop (hint: you will need to construct a pre-signed download URL, using a similar process to that for your input file upload).

- If the user clicks on the log file link, display (don't download) the log file in the browser using the template `view_log.html` (also helpfully provided :-).

7. (30 points) Archive Free user data to Glacier

Our policy for the GAS is that Free users can only download their data for up to 5 minutes after completion of their annotation job. After that, their results data will be archived to a Glacier vault. This allows us to retain their data at relatively low cost, and to restore it in the event that the user decides to upgrade to a Premium user.

Write a Python script called `archive.py` that moves the results files of Free users from the `gas-results` bucket to the Glacier vault called `'ucmpcs'`.

There are many ways to accomplish this type of periodic background task, including polling the database, using message queues, using a workflow service like Amazon SWF, etc. In general, polling approaches are simpler but less scalable, while other approaches introduce complexity into the system. For the purposes of this assignment, you must use a message queue in some form, and *you must describe the specifics of your approach and your rationale for choosing this approach* in the notes you submit with your final project.

Important: Glacier does not provide a nice interface for viewing/downloading your archived objects. For this reason you **must** capture the object's Glacier ID and persist it to the DynamoDB item for the annotation job. You will need to do this for only the results file, not the log file (since the log file remains accessible to free users indefinitely):

```
{
  'results_file_archive_id': 'xnJmVNzaIxd96GCMa8VHYHE1aSNsKY5p3SqRZ8Q5G7MUB....'
}
```

Again, since `archive.py` is not a Flask application you will need to use `archive_config.ini` to store configuration parameters such as the DynamoDB table name and SQS queue names.

Note: The terms "archive" and "Glacier object" are used interchangeably to refer to files stored in Glacier. They are all objects, but AWS calls objects in a Glacier vault "archives".

8. (5 points) Upgrading Free users to Premium via Stripe payments.

Our GAS application uses a subscription model whereby Premium users can pay $999.99 per week to run as many analyses as they want and store as much data as they want -- you can decide whether or not this is a realistic and sustainable business model!

Given the reduced project scope for this quarter, we did the bulk of this work for you. The GAS is already integrated with the Stripe service (www.stripe.com) to manage all subscription and billing functions. Stripe is one of many such services and it was chosen because it is very simple to integrate and has among the best API documentation of any SaaS (stripe.com/docs/api#intro). (Side note: For the vast majority of applications you should avoid building your own credit card billing functionality because it requires that you implement extensive PCI compliance processes, and exposes your organization to potentially very costly risks).

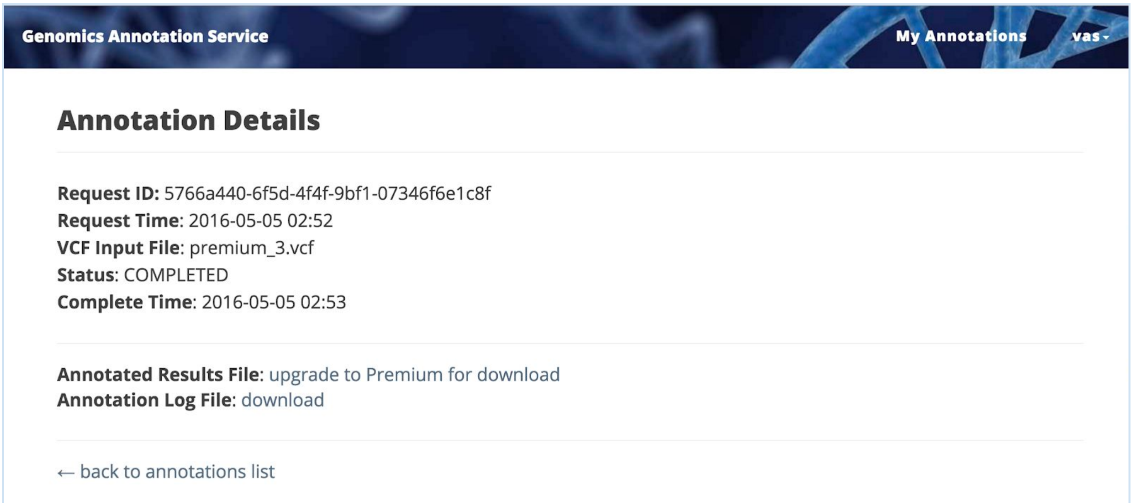
We used Javascript to invoke Stripe (see code added to the bottom of the `scripts.html` file). This provides us with some magic for the subscription form (otherwise we would need to do all sorts of complex things to ensure compliance with financial regulations). Specifically, it allows us to pass a set of fields containing sensitive information directly to the Stripe service, without hitting our web server.

A Flask route, `/subscribe`, in `views.py` processes the submitted billing information. It extracts a special-purpose field called `"stripe_token"` from the submitted form. That field is auto-magically created by the Stripe Javascript code and is a one-time use token for accessing the Stripe API securely. We use it to create a customer in Stripe and subscribe the customer to the premium plan. A successful call returns a Stripe `Customer` object. We then update the user's profile in our accounts database, changing the user's role to "premium_user". In a real application we would also store the Stripe customer ID (and perhaps some other details) in our user's profile so we can retrieve the user's billing records and otherwise link to the Stripe service, if necessary. However, for the purposes of this project we're ignoring this.

Now, here is where your work comes into the flow: we need to restore any annotation result files that were already archived for this user, and prevent any of the user's result files currently in the `gas-results` bucket from being archived. This work is described in Exercise 9 below.

Finally, the `/subscribe` route confirms a successful upgrade by rendering the `subscribe_confirm.html` template.

In the meantime, now that you can distinguish between Free and Premium users, we need to update the code you wrote in Exercises 5 since Free users may only download results for up to 5 minutes after completion. (Yes, it's ridiculously restrictive, but we're using such a short interval so we can test things in the limited time we have available). With this in mind, modify the job details page in Ex. 5 so that, instead of hyperlinking the results filename to start a download, a message is displayed offering the user the opportunity to upgrade to Premium (see the screenshot below as an example).



You will need to check if more than 5 minutes have passed since the annotation job completed. If so, display the link to upgrade; if not, display the a to download the results file. Note: The log file should always be available for viewing, irrespective of the user's Free/Premium account status.

Note: While testing data archival and restoration you will probably want to revert a premium user to a free user multiple times. You can do so by going to the `/profile` *page; if the user is currently a premium user click the "cancel my Premium plan" link to revert to a free user.*

9. (30 points) Restore data for users that upgrade to Premium.

When a Free user converts to a Premium user, you must move all of that user's results files from the Glacier vault back to the `gas-results` S3 bucket. Note that this is a two-step operation:

1. Initiate a Glacier archive "retrieval job" that asks Glacier to "thaw" the object and make it available for download. Your code must first attempt to use Expedited retrievals from Glacier; when Expedited retrieval requests fail, it must revert to using Standard retrievals. This illustrates an important concept called "graceful degradation". While objects are being restored, your system should display a message to that effect if a newly upgraded user tries to access previously archived data.

   VERY IMPORTANT NOTE: Standard retrievals take 4-5 hours to complete so plan accordingly when testing your code; if you leave this until the last couple of days you will not complete the project - trust me on this!

2. After the object is restored from Glacier, move the object to the `gas-results` bucket and delete it from Glacier. Note: When restoring a Glacier object, a copy of the object exists only for a limited period (~24 hours), after which it's no longer available for download.

Since there is an indeterminate delay between your restoration request and the time that the object for download, you will need to run code in two places:

- After the user has upgraded their membership, you must initiate the archive restoration job for results files that were generated while the user was a Free user.

- Two separate scripts, `restore.py` and `thaw.py`, must be used to restore objects from Glacier to S3. `restore.py` initiates the restore job and `thaw.py` checks on the status of restore jobs and moves restored archives to the user's results bucket.

As with archival, there are many ways to accomplish these tasks, each with their own tradeoffs -- again, you must use message queues to decouple restoration from other GAS tasks, and <u>*you must describe the specifics of your approach and your rationale for choosing this approach*</u> in the notes you submit with your final project. Once again, since `restore.py` and `thaw.py` are not Flask applications you will need to use separate configuration files (`restore_init.ini` and `thaw_config.ini`) to store parameters such as S3 bucket names and SQS queue names.

MILESTONE CHECK: You should now have a fully working GAS application. DO NOT proceed to the next exercise unless you have everything above working correctly. From this point on we will enable the GAS to scale out and in based on load.

---

10. (13 points) Add a load balancer for the GAS web servers.

Our single web app server is fine for development and testing but not so for production, since we expect heavy demand for the GAS service! Before the advent of cloud computing, scaling out required substantial custom code, complex system configurations, and lots of manual “handholding”. As discussed in class, all cloud providers offer mechanisms for automating scale out (and scale in, when system load subsides). AWS has multiple services that work together for this purpose—we will use the following:

- The Auto Scaling service allows us to define standard configurations for EC2 instances and then launch multiple instances as needed, based on user-definable rules. More info is here: http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/WhatIsAutoScaling.html

- The Elastic Load Balancer (ELB) service allows HTTP(S) requests to be distributed among multiple instances in our Auto Scaling group. More info is here: http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elastic-load-balancing.html

- The CloudWatch service provides metrics that are used by Auto Scaling rules to determine when to launch (or terminate) instances, in response to variable load. More info is here: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatch.html

In this exercise you will create a load balancer and an auto scaling group, then you will attach the load balancer to your auto scaling group. But first we must prepare the environment for automated launching of new instances.

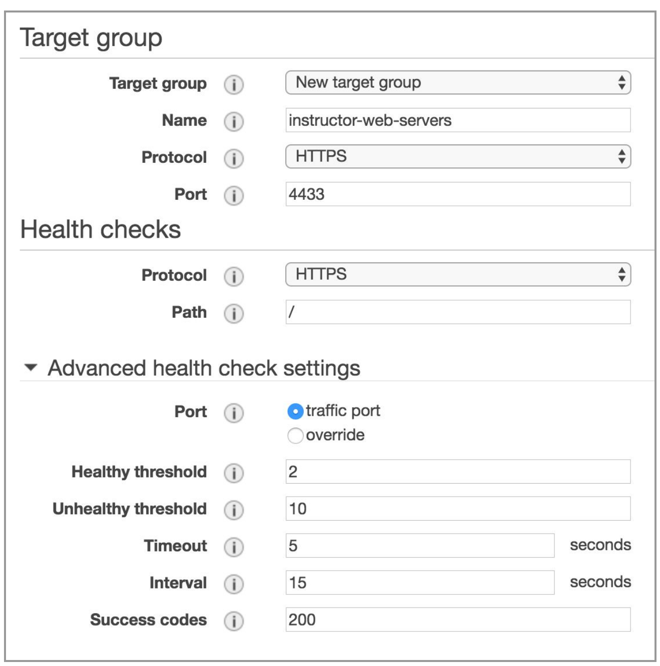
(a) (7 points) Preparing for automated launch: When the Auto Scaling service launches new instances, it uses a so-called “configuration group”. This configuration group defines all the usual EC2 parameters such as the instance type, security group, etc. Most importantly, the configuration group also includes <u>the user data that will run when new instances launch</u>. Up until now, after launching an instance we used SSH to access the instance and get everything else configured -- and then we started the server by running `./run_gas.sh`. All this manual intervention will clearly not work with Auto Scaling: new instances must configure themselves fully, and start running the GAS web server without any further input from us. In order for all of this to happen automatically, do the following:

- Upload your GAS code to S3. You can upload it to your home folder in the `mpcs-students` bucket. Before uploading, create a single ZIP file (e.g. in the sample below I call this file `gas_web_server.zip`) that contains all the required source code such as *.py, templates/*, static/*, etc.

- Create the user data file that will be used by new instances launched via Auto Scaling. Name this file `user_data_web_server.txt`. This user data file must contain commands to:
    - Download your GAS source code ZIP file from S3.
    - Unzip your source code ZIP file.
    - Change file ownership to `ubuntu:ubuntu` for your app files, e.g. `chown -R ubuntu:ubuntu /home/ubuntu/gas/...`
    - Copy the SSL certificate files from S3 to your instance (just as you did in Ex. 1)
    - Run the web server app. This must be the last command in `user_data_web_server.txt` and should look something like: `sudo -u ubuntu /home/ubuntu/gas/web/run_gas.sh &`

(b) (2 points) Create a load balancer.

- On the EC2 console, go to "Load Balancers" and click "Create Load Balancer".
- Select "Application Load Balancer" and click "Create".
- Enter a load balancer name (prefixed by your AWS username, e.g. `instructor-elb`).
- Under Load Balancer Protocol select "HTTPS (Secure HTTP)"; port will change to 443.
- Select all the availability zones and click "Next: Configure Security Settings"
- Select "Choose a certificate from AWS Certificate Manager (ACM)" and confirm that the "*.ucmpcs.org" certificate is selected. Click "Next: Configure Security Groups"
- Select "Create a new security group". Name your security group **the same as your ELB above**, and ensure that it allows only HTTPS traffic on port 443 from anywhere.
- Click "Next: Configure Routing". Here you are telling the ELB about the type of application that it will be serving. In this case the ELB will sit "in front of" our web servers.



- Create a new target group called <CNetID>-web-servers.
- Set the protocol to HTTPS and the port to 4433. Recall that our web server listens on port 4433, so the ELB will be routing standard SSL port (443) to port 4433 behind the scenes.
- Under “Health Checks”, set the protocol to HTTPS and the path to “/”.
- Under “Advanced health check settings” change “Healthy threshold” to 2, “Unhealthy Threshold” to 10, and “Interval” to 15 seconds. Note: These settings are *not appropriate* for production use, but they make things more convenient during development and testing.
- Click “Next” Register Targets” and then click “Next:Review”, followed by “Create”.

After a short interval your ELB will enter the “active” state and is ready for use.

(c) (4 points) Create an Auto Scaling Group and Launch Configuration.
Auto Scaling groups encapsulate the rules that determine how our application scales out (when demand increases) and in (when demand drops). Launch configurations provide a means to save your standard EC2 instance launch procedure so that it can be used by the auto scaler when launching new instances.

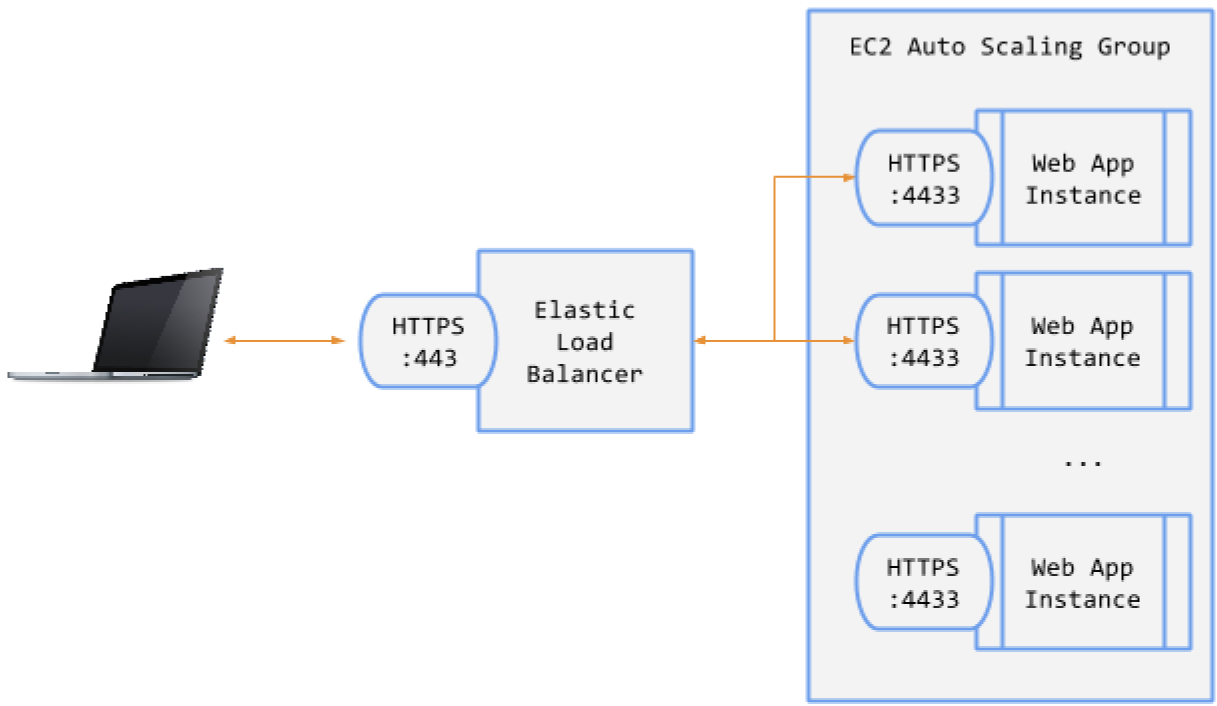
- On the EC2 console, go to “Auto Scaling Groups” and click “Create Auto Scaling Group”.
- Click “Create Launch Configuration”.
- Click “My AMIs” tab and select our class Base AMI.
- Select t2.nano as the instance type and click “Next: Configure details”.
- Name your launch configuration <CNetID>-web-launch-config.
- Under “IAM Role” select your instance profile (instance_profile_<CNetID>)
- Expand “Advanced Details” and enter the User data; <u>this is where you use the file created in part (a) of this exercise above</u>. Select “As file” and locate the user_data_web_server.txt file on your local machine. **Make sure your user data file is formatted as pure text, with no spurious characters. Hidden/non-text characters are the most common reason for user data not working.**
- Click “Next: Add Storage”; accept the defaults and click “Next: Configure Security Group”.
- Select “Create a new security group” name it <CNetID>-web-auto-scaling. By default, new security groups allow SSH traffic on port 22; our instance will not accept SSH traffic and will only allow HTTP traffic on port 4433 and Postgres (RDS) database traffic on port 5432.
- Change the security group rules so that they look like the example below:

Type	Protocol	Port Range	Source		
Custom TCP Rule	TCP	4433	Anywhere	0.0.0.0/0	⊗
Custom TCP Rule	TCP	5432	Anywhere	0.0.0.0/0	⊗

- Click “Review”, check the details, and click “Create launch configuration”.
- Select your EC2 key and click “Create launch configuration”.
- Name your Auto Scaling group <CNetID>-web-auto-scaler.
- Set “Group size” to start with 2 instances. We never want a single point of failure!

- Click in the “Subnet” field and select any one of the four subnets listed; this is necessary because none of our instances will have public IP addresses so they must be in a VPC subnet in order to be reachable.
- Expand “Advanced Details” and check “Receive traffic from Elastic Load Balancer(s)”. Under “Target Groups” select the name of the target group you created in part (b) above.
- Click “Next: Configure scaling policies”; leave the default “Keep this group at its initial size” selected for now and click through the next two pages until you get to the tags page.
- Add a tag with called “Name” with the value “`<CNetID>-gas-web`”. This is how all the instances in your auto scaling group will be tagged. *<u>Note: It’s particularly important that you tag your instances properly to avoid confusion, because there will be hundreds of instances running</u>*.
- Click “Review” and Click “Create Auto Scaling group” to finalize.

This is what your environment will look like now:



(d) Test your load-balanced configuration. It can take quite a while for the instances to launch and be configured, and for the load balancer to put them into service. Once they’re ready you should be able to go to the public DNS name of the load balancer and see the GAS home page. The URL will look something like this: `https://<CNetID>-elb-208692995.us-east-1.elb.amazonaws.com`.

Note: *You must specify HTTPS* -- and it’s no longer necessary to specify a port because the load balancer is listening on the default HTTPS port, 443.

Troubleshooting the configuration if your load balancer is not responding:

- Check that at least one instance is marked as “InService”. If both instances are out of service, manually remove them from the ELB and add them back after a few minutes.
- Manually launch a test instance to check that your user data is being executed correctly. You should be able to access the GAS home page on the instance without doing any additional manual configuration.
- Modify your launch configuration’s security group to allow SSH traffic on port 22; this way you can SSH into the instance(s) and see what’s going on.

Alternative approach to the above: It’s very likely that you will need to change your user data multiple times before it works as expected. You can do this via the AWS console but it is quite cumbersome:

1. Create a copy of the launch configuration
2. Edit the user data in the new launch configuration
3. Associate the new launch configuration with your auto scaling group
4. Delete the old launch configuration

A more efficient way is to use the AWS CLI:

`$ aws autoscaling create-launch-configuration --launch-configuration-name <CNetID>-web-launch-config-NEW --user-data ...` (*add path to user data file + all other parameters like security groups here*)
`$ aws autoscaling update-auto-scaling-group --auto-scaling-group-name <CNetID>-auto-scaler --launch-configuration-name <CNetID>-web-launch-config-NEW`
`$ aws autoscaling delete-launch-configuration --launch-configuration-name <CNetID>-web-launch-config`

The CLI approach takes a bit of time to set up but will usually save you time in the long run, especially since you have to do the same thing for the annotator instances below.

VERY IMPORTANT NOTE
When you have confirmed that your load balancer is working, run the `mpcsdomain` command to associate your personal subdomain (`<username>.ucmpcs.org`) with your load balancer:

```
$ ./mpcsdomain update --subdomain <CNetID> --elb <your_ELB_name>
```

11. (8 points) Add auto scaling rules to the web server farm.

Our Auto Scaling group is currently fixed in size at 2 instances. After extensive profiling and experimentation you've discovered that web app response times rise sharply when we're processing more than 200 requests per minute (yes, this just an arbitrary metric). Your profiling exercise also showed that loads of 100 requests per minute or lower can easily be handled without additional instances. We will now create scaling policies that add and remove instances as load changes.

(a) (2 points) Create a CloudWatch alarm on your load balancer that is triggered when the sum of successful responses exceeds 200 for one minute. Successful responses are those that return any of the HTTP 2XX status codes. (We're ignoring HTTP error codes (4XX and 5XX) just for simplicity). CloudWatch alarms are managed on the "Monitoring" tab in the load balancer console.
Note: Make sure you **do not** send a notification when the alarm is triggered, although in a live application you would definitely want to notify someone and/or log all alarms.

Really Important Note: If you created multiple ELBs above (maybe you were experimenting, or made mistakes the first time around) ***make sure you select the CloudWatch metric for your current/active ELB****. Unfortunately, AWS "helpfully" retains metrics for deleted resources for an extended period so you may see multiple instances of a metric. If you select an outdated metric your auto scaler will not work and you will struggle to discover the reason :-)*.

(b) (2 points) Create a policy that scales out your web server farm. Your policy must add one instance when the above alarm is triggered (i.e. when the ELB receives more than 200 HTTP 2XX responses in a minute). Policies are managed on the "Scaling Policies" tab of the Auto Scaling Groups console (click "Create a simple scaling policy" to see the correct inputs). Make sure you set the "And then wait" parameter to 300 seconds - this specifies how long the auto scaler waits before considering adding another instance to address the alarm.

(c) (2 points) Create another CloudWatch alarm on your load balancer that is triggered when the target response time is below 10ms for at least one minute.

(d) (2 points) Create a policy that *scales in* your web server farm. Your policy must remove one instance when the alarm in 11(c) above is triggered.

12. (19 points) Add scaling rules to the annotator.

So far we've focused on the front end of the GAS service. Now we turn our attention to automating and scaling the back end, i.e. the annotator. Scaling the annotator server farm configuration is somewhat simpler than that of the web server farm, since there is no load balancer involved (and hence no target groups); the auto scaler will simply add and remove instances based on the number of messages arriving in the job requests queue.

(a) (7 points) *Preparing the annotator instances for automated launch*: This is similar to the automated deployment of the web server instances, except you will need to deploy the Anntools files (as found at https://github.com/mpcs-cc/anntools), as well as your `annotator.py`, `run.py` and the file you use to store configuration parameters. As before:

- Create a ZIP archive containing the annotator files (call it `gas_annotator.zip`) and put it in your home folder on S3.
- Copy the web server user data to a new file, `user_data_annotator.txt`, and modify it so that it downloads and unzips the annotator ZIP file. Make sure you change file ownership after copying/moving since the user data commands run as root, e.g. `chown -R ubuntu:ubuntu ...`
- Add a command to your user data that runs the annotator script. Note that, unlike the web server, we don't have a shell script to start the annotator so just doing `python annotator.py` in your user data will not work. The user data commands run in a separate shell context and cannot use the Python virtualenv that we need. You will need to create a wrapper script (e.g. run_ann.sh) that sources the virtualenv and then runs annotator.py. Then, in user data launch this wrapper script: `sudo -u ubuntu /<path>/run_ann.sh &`

(b) (2 points) Create a launch configuration, `<CNetID>-ann-launch-config`, for your annotator instances. You can use the same instructions used for the web server launch configuration, except that the annotator <u>does not require</u> <u>any ports</u> open in its security group since it only calls on other AWS services (SNS, DynamoBD, etc.).

(c) (2 points) Create an auto scaling group for the annotator called `<CNetID>-ann-auto-scaler`. You can use the same instructions used for the web server in Exercise 10 above, except that you will not associate an Elastic Load Balancer with this auto scaling group. As previously mentioned, <u>*you must add a "Name" tag to your auto scaler instances*</u> with the value: "`<CNetID>-gas-ann`".

(d) (2 points) Create a CloudWatch alarm to <u>*scale out*</u> the annotator server farm. Since this alarm is not based on EC2 metrics we will need to create it via the CloudWatch console.

- On the CloudWatch console, select "Alarms" and click "Create Alarm".
- Click on "SQS Metrics" and locate your job requests queue.
- Select the "NumberOfMessagesSent" metric and click "Next".
- Set the parameters so that the alarm triggers when more than 50 messages are sent to the job request queue in 10 minutes.

- Under “Actions”, delete the default notification.
- Click “Create Alarm”.

(e) (2 points) Create another CloudWatch alarm to *scale in* the annotator server farm when the number of messages arriving in the queue is below 5 for 10 minutes.

(f) (2 points) Create an Auto Scaling policy to *scale out* the annotator server farm. It must add one annotator instance to the farm when the message queue alarm in 12(d) above is triggered.

(g) (2 points) Create an Auto Scaling policy to *scale in* the annotator server farm when load subsides. It must remove one annotator instance from the farm when the message queue alarm in 12(e) above is triggered.

That completes the required work under our reduced scope this quarter. If you complete everything up to this point you are eligible to get 100% on the assignment.

If you complete any exercises beyond this point you will get extra credit.

OPTIONAL: 13. (10 points) Test under load using Locust.

Now that we have our scaling policies set up let's see them in action. After all, a key reason we're building cloud apps is to reap the benefits of elasticity, so we should know how our app will behave under varying load conditions. We use a simple load testing tool called Locust. This is not an all-singing, all-dancing tool that you might encounter on the job, but it's good enough to give you a taste for automated testing. We will use it to run a crude experiment on our load balancer.

The tool simulates a number of concurrent users accessing the GAS. Each user submits a number of requests to the GAS with a specified frequency, thus creating an arbitrarily high load - a swarm of locusts descending on your ELB!

(a) Update your web server auto scaling configuration. Recall that we set a fixed limit of 2 instances when we created the auto scaling group; our auto scaler will not respond to alarms unless we change this. Go to the Auto Scaling console, select your group; under the "Details" tab, **set "Max" to 10 instances and "Desired" to 2 instances**.

(b) Configure and run a Locust server on a separate EC2 instance and then access it via its browser console. I recommend you spend some time reading the Locust docs so you have a better sense for how the pieces fit together.

1. Launch a new EC2 instance to run the locust server, then SSH into the instance and install the locust package:
   ```
   workon mpcs
   pip install --upgrade pip locust
   ```

2. Create a file called `locustfile.py` that contains the following (https://gist.github.com/vasv/5f07bd6f17df3ce12fe7a05a257869ad):

   ```
   from locust import HttpUser, between, task

   class UserBehavior(HttpUser):
     wait_time = between(1, 3)

     def on_start(self):
       pass

     @task(1)
     def index(self):
       self.client.get("/")
   ```

Note that you can test against pages other than the home page (“/”), but **make sure you do NOT hit any pages that require authentication** otherwise Locust will effectively generate a DDoS attack against Globus Auth and you will be shut down. This is very serious!

3. When you’re ready to test, run: `locust --host=https://<CNetID>.ucmpcs.org`

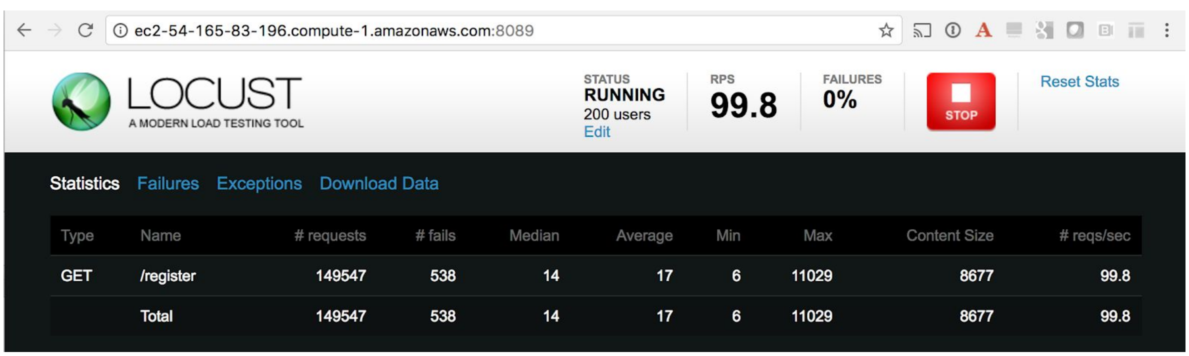
```
ubuntu@ip-172-31-17-43:~$ locust --host=https://instructor.ucmpcs.org
[2017-05-20 10:40:52,487] ip-172-31-17-43/INFO/locust.main: Starting web monitor at *:8089
[2017-05-20 10:40:52,487] ip-172-31-17-43/INFO/locust.main: Starting Locust 0.7.5
```

4. In your browser, navigate to the Locust console; your Locust server is listening on port 8089, e.g. `http://<CNetID>-locust.ucmpcs.org:8089/`. You should see a form asking for swarm size and rate.

(c) Run load test(s). Start with a small swarm, e.g. 100 users @ 5 users/second. Your server will show how the swarm is growing and confirm when all your locusts have hatched.

```
ubuntu@ip-172-31-17-43:~$ clear
ubuntu@ip-172-31-17-43:~$ locust --host=https://instructor.ucmpcs.org
[2017-05-20 10:40:52,487] ip-172-31-17-43/INFO/locust.main: Starting web monitor at *:8089
[2017-05-20 10:40:52,487] ip-172-31-17-43/INFO/locust.main: Starting Locust 0.7.5
[2017-05-20 10:42:56,655] ip-172-31-17-43/INFO/locust.runners: Hatching and swarming 100 clients at the rate 5 clients/s...
[2017-05-20 10:43:16,728] ip-172-31-17-43/INFO/locust.runners: All locusts hatched: WebsiteUser: 100
[2017-05-20 10:43:16,728] ip-172-31-17-43/INFO/locust.runners: Resetting stats
```

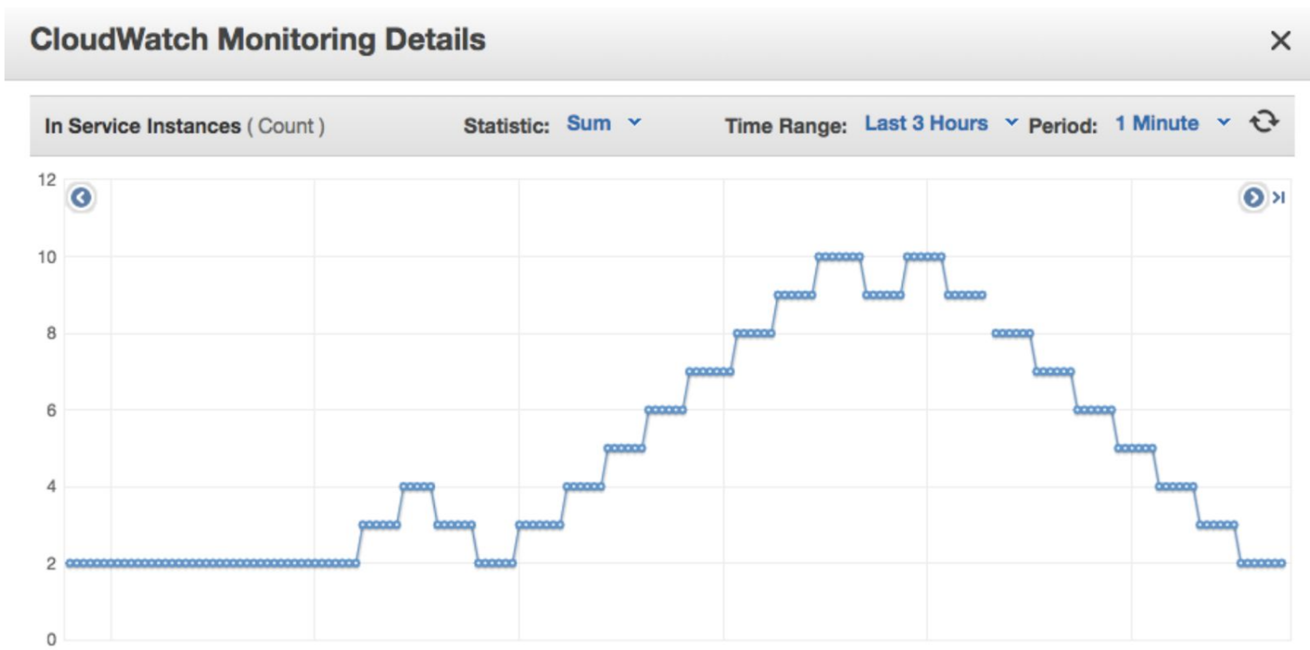
Step up to a larger test load (maybe 200-300 users, at ~20-30/sec.) On the web console you will see the swarm in action tests run. Mine looks something like this:



(d) (5 points) Monitor your auto scaling group and note when instances are added. Briefly describe what you observe, and more importantly, try to explain why things are happening when they do. I’m looking for insight, not just a replay of events.

(e) (5 points) After observing scale out behavior, stop your test by hitting the red STOP button on the Locust web console. Monitor your auto scaling group and briefly describe what happens when your scale in policy goes into effect. Again, insight into cause/effect is required.

Capture screenshots of the Locust web console showing your tests and include them in your writeup. Also capture some screenshots of the AWS CloudWatch console that shows how the instances in your auto scaling group grew and shrank at various points. This is best done by viewing the CloudWatch instance count graph at 1 minute intervals; it should look similar to this:



A general word of advice: be patient during load testing. Bear in mind that many things are happening asynchronously across your infrastructure so you may not see results when you expect to. This is why I want you to describe/explain your observations.

<u>VERY IMPORTANT: When you're done load testing please terminate your Locust instance and make sure your auto scaling group only has two instances running. We get very close to the limit on the number of concurrent EC2 instances and your fellow students may be prevented from running their tests if you leave your instances running.</u>

OPTIONAL: 14. (10 points) Load test the annotator farm.

We can't use Locust to test the annotator farm since our instances are not exposed to the Internet but we set the alarm thresholds purposely low in Exercise 12 so we can use a simple script to simulate load.

Write a script, `ann_load.py`, that submits an arbitrary number of messages to the request queue (simulating job submissions). You can use hardcoded, repetitive test data, e.g. a fixed user ID. Run your script and observe how the auto scaler behaves; include observations and, again, speculate on the cause/effect of the observed behavior in your project write-up (as in 13(d/e) above).

Final Deliverables

You are required to submit the following:

1. A fully-functional GAS **accessible at https://<CNetID>.ucmpcs.org**. Your environment must include three types of instances:
    a. Two instances running the web application
    b. Two instances running the annotator
    c. One instance running the utility scripts, each in a separate tmux sessions

    I cannot stress enough how important it is to have your app actually running. We will not have the time to start up your instances and configure the ELB to see if your code actually works so you will be heavily penalized (-30 points) if your app is not reachable at <CNetID>.ucmpcs.org. If you do not get to the point where your ELB is working, please note this at the top of your README.md file and give us the DNS of an instance that we can access directly via port 4433.

2. **Complete source code** committed to GitHub that includes:
    a. Python code
    b. Web app templates
    c. Configuration file(s)
    d. AWS EC2 user data files
    e. Any other files necessary for your GAS to run

    Specifically, your code must be organized as follows:
    - `/web`: your updated `views.py` and `config.py` files; I expect that only these two files have changed relative to the gas-framework repo that you started with. If you modified other files from the gas-framework include them here and provide a description of what you changed (and why) in README.md
    - `/web/templates:` all HTML templates
    - `/ann:` `annotator.py`, `run.py`, `ann_config.ini`, and all other AnnTools files
    - `/util: util_config.ini, helpers.py`, `ann_load.py`
    - `/util/archive: archive.py, archive_config.ini`, other archive related code
    - `/util/notify: notify.py, notify_config.ini`, other notify related code
    - `/util/restore: restore.py, restore_config.ini`, other restore related code
    - `/util/thaw: thaw.py, thaw_config.ini`, other thaw related code
    - `/aws`: `user_data_web_server.txt`, `user_data_annotator.txt`

    Due to the massive grading task that this project represents it's very important that you adhere strictly to the above structure. ***<u>If you use a different directory structure you will be penalized 20 points</u>***.

Reminders:

- As always, include error trapping and handling code for all critical actions.
- Do not hardcode anything that doesn't need to be hardcoded; credential hardcoding will result in -20 points. For the utility scripts you must modify the relevant config file to avoid hardcoding global variables for things like the DynamoDB table name, SQS queue names, and SNS topic names.
- Please clean up your code (remove testing `print()` statements, unused functions from old homework, etc.).
- Add comments, especially for new code, e.g. for job listings and file downloads.
- List all references as inline comments. Failure to cite references where we expect you to have consulted outside sources *will result in -5 points per occurrence*.

3. Your repo must include a `README.md` file with:
    a. Description of your archive process.
    b. Description of your restore process.
    c. Anything else you'd like to share with the grader. Include notes here if you didn't manage to complete all the exercises and wish to describe what issues you encountered that prevented you from doing so.
    d. If you completed the optional exercises, include descriptions of what you observed and why.If you prefer, you may include the auto scaling description and screenshots in a PDF file.

NOTE: Under no circumstances should you submit descriptions or notes in any other format like MS Word, RTF, etc. These files will be ignored by graders and you will forfeit any points related to those questions.

Instructor Grading Rubric

This is the flow we will use to test your GAS:

1. Login and run multiple annotations.
2. Check that we receive notification emails when annotations are complete.
3. View a list of annotation jobs.
4. Click on jobs in the list and view the job details.
5. Click on a link to download the input/results files
6. Click on a link to view the log file (should work for all users right after results are received).
7. Revisit the list of annotation jobs, after allowing about 10 minutes to pass. Click on a link to download the results file (this should not work; we should be asked to upgrade, by displaying an upgrade link instead of a file link); we should <u>still be able to view the log file</u>.
8. Run another few annotations.
9. View the user's profile (account); click the link to upgrade to a Premium user.
10. Provide the credit card info and upgrade to a Premium user.
11. View annotations run in step 8; all results should still be available since we just upgraded.

12. View the jobs list again after allowing thawed results to be moved out of Glacier.
13. Click on a link to download the results files, and view the log file (this should all work now since it's a Premium user).

We will check the following in the AWS environment:

- Uniquely identified input, output, log files are stored in the correct S3 buckets.
- Correctly updated items are stored in the database for each job.
- Archived Free user files are stored in Glacier, and Glacier IDs are in the database.
- You have correctly configured notification topics and subscribed the corresponding message queues to their topic.
- You have a running load balancer with, at most, two instances in service.
- You have two Auto Scaling groups: one for the web app and another for the annotator, each with two instances.
- Your auto scalers work correctly, i.e. new instances are launched and become operational automatically in the event of a failure; to test this **we will terminate all -web and -ann instances, and expect your entire environment to automatically be restored - this is a critical aspect of using cloud infrastructure**.
- All your instances are properly name-tagged by your auto scalers during launch.
- You have a separate instance running the various utility scripts.

We may use Da Locusts to test the scaling policies of your web auto scaling group. We expect that it will scale up and down under load without exceeding the maximum of 10 instances or going below the minimum of 2 instances. We may also test the annotator auto scaling group.

FINAL IMPORTANT NOTE: At this stage of the class there is no excuse for not having the basic annotation flow working (i.e. we should be able to select/upload a file and view the results after a few seconds/minutes). If your GAS does not successfully complete annotations and return results we are unable to grade the rest of the project; in this case we will deduct 50 points and you will most likely end up with a failing grade on the project.

P.S. Based on past experience, most students will need about 40-60 hours to complete the project (some students have spent more than 80 hours). With this in mind, PLEASE START EARLY—every year I stress the importance of starting the project early, and every year there are a few students that ask for extensions because they thought they could get it all done over the last weekend. Don't be one of those students—**ALL REQUESTS FOR EXTENSIONS WILL BE DENIED.** Good luck!