MPCS 51083 (Spring 2017) - Capstone Project

<u>if you are graduating this quarter, project is due Wednesday, May 31 at 5:30pm</u>

<u>if you are not graduating, project is due Tuesday, June 6 at 5:30pm</u>

Total points: 200 + 5 bonus

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 (but feature limited) 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 add functionality for scaling the application and will integrate with an external cloud service for additional functionality.

Key Functions

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

- **Register for an account to use the service** -- All users of the service must be registered. Two classes of users will be supported: Free and Premium. Premium users will have access to additional functionality, beyond that available to Free users.

- **Convert 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 the test credit card accounts 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 to the user to convert to a Premium user.

- **Receive notifications when annotation jobs finish** -- The GAS will send users an email when their annotation request has completed, including a link where they can go to view/download the result/log files.

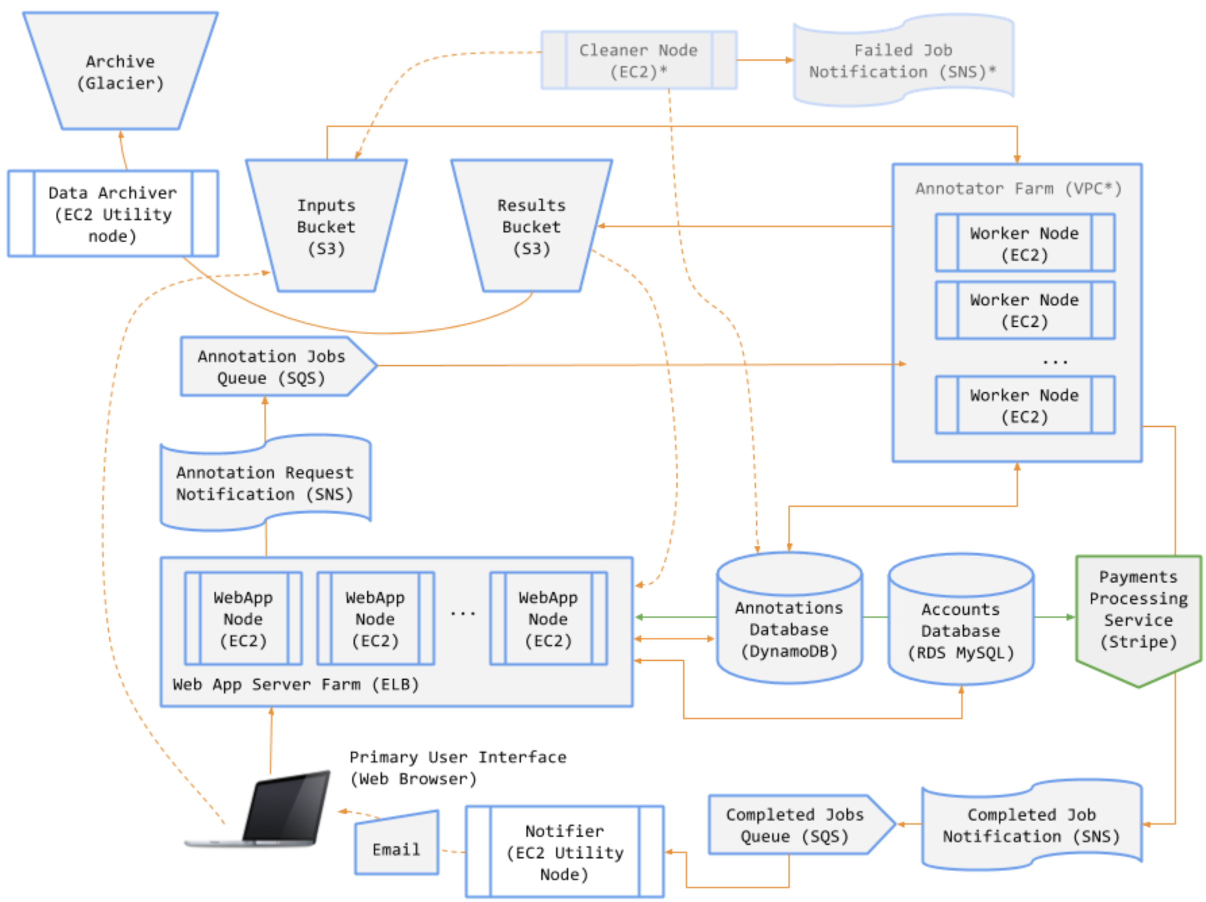
- **Browse jobs and download annotation results** -- The GAS will store annotation results for later retrieval. All users may view a list of jobs (completed and running). Free users may download results up to 30 minutes after their job has completed; thereafter their results will be archived and only available to them if they convert to a Premium user. Premium users will have all their results 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 cheap, highly-durable store for archiving the data of Free users.
- A relational database for user account information, job metadata and job logs.
- A service that runs the AnnTools software for annotation.
- A (front end) web application for users to interact with the GAS.
- A set of message queues and notification topics for managing various system activities.

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



* Components are not implemented in the Capstone Project. May be added for bonus points.

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. For these reasons 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 certain thresholds, the GAS will terminate the excess web servers.

2. On the back end, the annotator service will be delivered by multiple servers 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, some components will be extended and others will be added as follows:

- Create a multi-threaded and secure environment for running our web server.
- Add user registration, authentication, and authorization mechanisms.
- Add mechanisms for handling completion and notification of annotation jobs.
- Enable comprehensive listing of a user’s annotation jobs.
- Add ability to download results and view the annotation log file.
- Integrate credit card payment functionality for premium users.
- Archive Free users’ data (to Amazon 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 using Locust.

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

Preparation

Launching Instances

Please adhere to the guidelines below as you set up your environment for this project:

- You **must** use this command to launch your instances:

```
aws ec2 create-tags --resources `aws ec2 run-instances --image-id
ami-e3b1c7f5 --count 1 --instance-type t2.micro --iam-instance-profile
Name="instance_profile_<username>" --key-name <your_key_name>
--security-groups mpcs --user-data
https://s3.amazonaws.com/mpcs-resources/mpcs_capstone_user_data.txt
--query 'Instances[0].InstanceId' --output text` --tags
'Key=Name,Value=<your_username>-capstone-web'
```

Note the highlighted parameters:

- We're using a new AMI (which is configured to simplify automation)
- The command sends user data to initialize your instance. You will need to automate certain actions later in the project when experimenting with scaling, so it's worthwhile spending the time to understand how this works now. You can view the user data here: https://s3.amazonaws.com/mpcs-resources/mpcs_capstone_user_data.txt.

- **IMPORTANT UPDATE:** I added the argument `--iam-instance-profile Name="instance_profile_<username>"`. It associates an IAM role with the instance that allows applications running on the instance to get temporary AWS credentials dynamically, i.e. **you no longer need to manually add your AWS access key and secret to the instance**. As I mentioned in class, this is a best practice for security. Make sure you change `instance_profile_<username>` before running the command, e.g. `instance_profile_instructor`. These benefits come at the cost of some changes to your code:

    - When creating an AWS resource with boto3 you may need to specify the AWS region. This is because you no longer run `aws configure` on the instance, and hence it has no knowledge of the default region to use for your boto3 commands. For example, to connect to Dynamo DB you would write: `dynamo = boto3.resource('dynamodb', region_name='us-east-1')`; previously this was just: `dynamo = boto3.resource('dynamodb')`. Better yet, instead of hardcoding the region name, you can use the Bottle config: `region_name=request.app.config['mpcs.aws.app_region']`.

    - When using temporary AWS credentials, some operations—in particular, signing requests/creating signed URLs—require that you use a session token, *in addition* to the AWS access key and secret. Rather than have you figure all this out on your own, I decided to add my modified S3 upload code to the gas-framework repo. There are a few minor changes to what I shared previously:
        - The S3 policy must now include the access token; this takes the form: `...{"x-amz-security-token": aws_session_token}, ...`
        - We can get the token from the session object thus: `aws_session_token = str(session.get_credentials().token)`
        - The form in `upload.tpl` must also include the token as a field (recall that the form must have exactly the same fields as the policy: `<input type="hidden" name="x-amz-security-token" value={{aws_session_token}} />`

    Again, all these changes have already been made in `mpcs_app.py` for your coding enjoyment!

- Ensure that you use the **`mpcs`** security group when launching instances (as in the command above). *Do not use your own group or the one automatically generated by EC2* when launching an instance.

- When accessing a new instance at an existing DNS name (e.g. **`<username>.ucmpcs.org`**) via SSH, you may get a host identification error. Run this command to remove the old DNS name from your authorized hosts file: **`ssh-keygen -R <username>.ucmpcs.org`**

Using the Enhanced Bottle Web Framework

We want the GAS to look somewhat appealing so we are going to use an enhanced version of the Bottle framework for this project. *Note: You must use this updated framework because it contains required code for registering new users and managing authentication/authorization.*

Get the framework: **`git clone https://github.com/mpcs-cc/gas-framework.git mpcs`**

(It's also downloaded and unzipped automatically as part of the user data passed to the instance when launching instances with the command above)

The enhanced framework provides the following:

- Styled web pages using the [Bootstrap CSS library](). You don't need to change anything related to styling, but if you want to experiment with a different look and feel, have at it!

- A modular template structure so every page in your application shares a header, footer, and other HTML elements. In order to use this structure you need to be aware of the following:

    - Your template (**`.tpl`**) files must reside in the **`views/`** directory, off the root directory of your application.

    - You no longer need to include header and footer HTML in your template. Your templates will now look like this:

```
%include('views/header.tpl')

<div class="container">
    <div class="page-header">
        <h2>My Page</h2>
    </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>
    </div>
</div> <!-- container -->

%rebase('templates/base', title='GAS - My Page')
```

Important Note: The `%rebase` statement **must** be included as the last line in your template. This tells Bottle to replace a placeholder in the base template with the content of your template.

See the `home.tpl` and `login.tpl` for two examples of templates that you can repurpose for your use.

- The framework uses the Bottle `Config` class to set and access various configuration values throughout your application. You can add and change configuration key/value pairs in `mpcs.conf` using standard .ini file format. Then you can access these in your code using dotted notation, e.g. `request.app.config ['mpcs.aws.s3.inputs_bucket']` will return the value “`gas-inputs`” -- note that all config values are returned as strings.

- If needed, the framework provides a logger that can be used to log events to a file and to the console. To use the logger, add this line to your source file: `from mpcs_utils import log`. Then you may write to the log like so: `log.info("A log message...")`.

- Runtime environment configuration parameters are stored in OS environment variables and read into the Bottle config at startup. These include important parameters like the location of static resources and the port that the application listens on. To change these parameters, edit the file `run_gas.sh`. The default file looks like this:

```
export MPCS_DEBUG=True
export MPCS_APP_HOST=0.0.0.0
export MPCS_APP_PORT=4433
export MPCS_STATIC_ROOT=./static/
export MPCS_TEMPLATES_ROOT=views/
export MPCS_LOGS_ROOT=./log/
export MPCS_LOG_FILE=mpcs.log
[[ -d ./log ]] || mkdir $MPCS_LOGS_ROOT
if [ ! -e $MPCS_LOGS_ROOT/$MPCS_LOG_FILE ]; then
    touch $MPCS_LOGS_ROOT/$MPCS_LOG_FILE;
```

```
fi
python web_server.py
```

When using the enhanced framework, in order to start our web server we just do:

```
./run_gas.sh
```

- **web_server.py** imports **mpcs_app.py** which is where your front end source code will be added. **mpcs_app.py** currently has code for just the home page and the registration/login forms.

IMPORTANT NOTES

- Your existing code for uploading files, sending notifications to SNS, etc. must be moved into/added to the **mpcs_app.py** file (see Ex. 2 below).

- Your existing **upload.tpl** template (and any other templates you created previously) must be converted to use the modular structure. Essentially, this means removing any header and footer HTML, and adding the **%include** and **%rebase** statements at the top and bottom of the file.

The enhanced framework also implements some rudimentary security. From this point on our web server will respond only to HTTPS requests. We require HTTPS for things like registering and authenticating users. Many web sites/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*. We will also substitute the default Bottle WSGI server with the CherryPy server which is a well-tested, multi-threaded WSGI server suitable for production use.

Browse **web_server.py** and see how it uses the CherryPy (cheroot) WSGI server. The standard Bottle 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 8888.

Copy the ucmpcs.org SSL certificate and private key to your instance; put these files in the same directory as **web_server.py**. If you use the command listed above to launch your instance, these files will be automatically added as part of the user data passed to the instance. If you launched an instance using the console or other command you can download and unzip the SSL files from: https://s3.amazonaws.com/mpcs-resources/mpcs_ssl_files.zip.

Exercises

1. (7 points) Configure the enhanced framework on your web server instance.

We will use the Cork Python module to implement this authentication and authorization functionality. Cork is a very simple, lightweight module that uses a relational database (among other storage types) to store user profile information. The database is accessed via SQLAlchemy, a widely used Python module for interacting with SQL databases. Cork also makes use of the Beaker Python module to store session information (e.g. cookies that contain the username). Cork, SQLAlchemy, and Beaker are already installed on your instance when you use our class AMI. I decided to spare you the misery of working with all these modules individually and so I've already integrated them into the framework. All you need to do is setup and initialize the database and configure your web app to use them. Credentials for 1(a), 1(b) and 1(c) are available in this file: https://s3.amazonaws.com/mpcs-resources/mpcs_conf_rds_ses.zip.

(a) Initialize the user database. Cork uses three tables for authentication and authorization: **`register`**, **`users`**, and **`roles`**. When you execute the SQL script below, it will create these tables and populate the roles table with some default values.

- Edit **`mpcs_initialize_user_db.sql`** and set the name of the database in three places:

  ```
  DROP DATABASE IF EXISTS <username>_accounts;
  CREATE DATABASE <username>_accounts;
  USE <username>_accounts;
  ```

  On each of the above lines, replace **`<username>`** with your AWS username (CNetID). For example, my database would be called **`instructor_accounts`**.

- Now run the command below to initialize your database. This should be run from your EC2 instance (you *can* run it on your laptop, but you must have the MySQL client tools installed):

  ```
  mysql --host=ucmpcs.catcuq1wrjmn.us-east-1.rds.amazonaws.com --port=3306
  --user=<admin_user> --password < mpcs_initialize_user_db.sql
  ```

You only need to run this once; after the database is initialized, all new web server instances that you launch will simply access it.

(b) Edit **`mpcs.conf`** and update the database URL, **`mpcs.auth.db_url`**. Change the username/password and the database name, e.g. **`instructor_accounts`**.

(c) Configure your application to use Amazon Simple Email Service (SES) for sending email notifications (e.g. user registration confirmation, annotation job completion). In **`mpcs.conf`**:

- Change **`mpcs.auth.email_sender`** to: **`<your_aws_username>@ucmpcs.org`**.
- Update the username/password for **`mpcs.auth.smtp_url`**

(d) Modify the **views/registration_email.tpl** template to include your personal subdomain so that registration confirmations go to the correct URL.

Now that you've configured the framework you can use the **auth** object:

- You can access an authenticated user's attributes as follows (you must include: **from mpcs_utils import auth** in your source file):
    - **auth.current_user.username** - returns the user's GAS username
    - **auth.current_user.role** - returns the user's role
    - **auth.current_user.description** - returns the user's full name
    - **auth.current_user.email_addr** - returns the user's email address

- Cork is initialized with the following roles (these are the only valid role names for the GAS):
    - **free_user**: has access to pages/functions that require authentication, e.g. the /analysis page.
    - **premium_user**: has access to pages only available to Premium users
    - **admin_user**: has access to admin pages/functions (not used in this project)
    - **super_user**: has access to all system pages/functions (not used in this project)

- **Make sure you pass auth=auth when calling template() to render a template;** then you can access the logged-in user's profile information in the template as: **{{auth.current_user.<user-attribute>}}**.

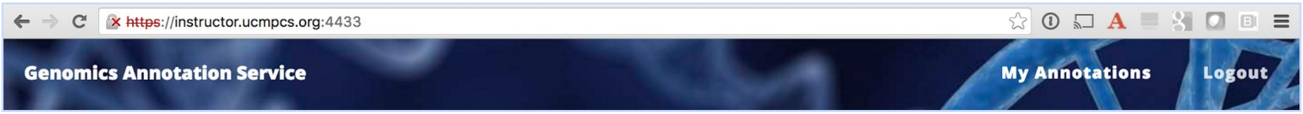
- If you want to access variables stored in the session, use: **auth._beaker_session.get('<session-variable-name>')**

- You can restrict access to a route/function in the GAS by requiring that a user be authenticated. Make the first line in your route: **auth.require(fail_redirect='/login')**. This redirects unauthenticated users to the "/login" page to authenticate.

- You can also specify that a user be redirected back to the requested page after authenticating, by specifying a redirect URL in the authentication check: **auth.require(fail_redirect='/login?redirect_url=' + request.url)**.

- If you want to restrict access to a function/page only to a specific role, use the following: **auth.require(role=<role-name>, fail_redirect='/login')**. See above for valid role names.

VERY IMPORTANT - Before continuing, start your web server (using **run_gas.sh**) and test your new setup by going to **https://<username>.ucmpcs.org:4433**. You should see the GAS home page (you can ignore any SSL certificate security warnings).



You should be able to:
- Register a new user account and receive a confirmation email.
- Click on the link in the confirmation email to activate your new account.
- Login with your new account and see the “My Annotations” options on the header nav menu.

If you cannot access the web app and complete these tasks 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. (12 points + 5 bonus) 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 **`mpcs_app.py`**.

(b) (4 points) Expose new functionality via the GAS menu system, e.g. add an option to list annotation jobs and to submit a new job (see the **`base.tpl`** file on how to do this; basically it’s just entries in a <ul> list).

(c) (6 points) Up until now we’ve hardwired a username for the annotation request. Modify your code so that it uses the **`auth`** object to get the authenticated user’s username 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 and register/login**.

BONUS POINTS: For 5 bonus points, you can elect to implement a check for file size when a Free user is requesting an annotation. You can implement this as in two ways:

- *Using Javascript in the upload form*
- *Using an Amazon policy condition*

*Use **150KB** as the input file size limit for Free users. This lines up with the five sample input VCF files provided (the two free files are both below this limit).*

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 HW6, create an SNS topic for results notification. The topic name should include your AWS username, e.g. **`instructor_job_results`**.

(b) (1 point) As you did in HW6, 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 HW6).

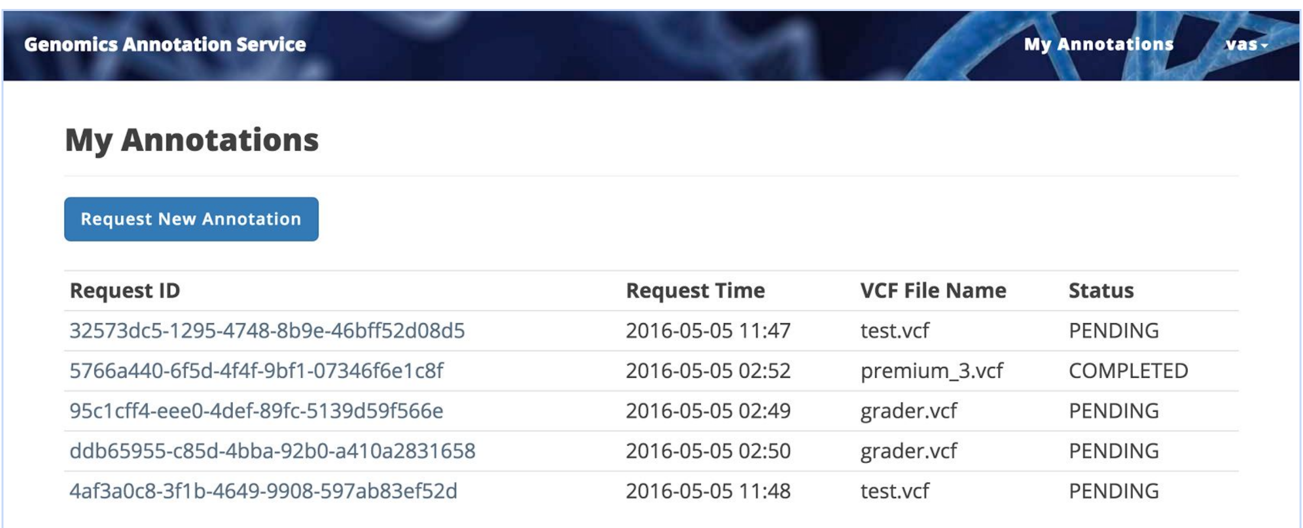
(d) (3 points) Modify **`run.py`** so that it publishes a notification to this topic when the job is complete, i.e. after the database has been updated and the results/log files have been copied to S3.

(e) (10 points) Write a new Python script called **`results_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 **`job_runner.py`** in HW6, except that it sends an email message instead of launching an annotation job. In order to send email, use boto to access the Amazon SES service; docs are here: http://boto3.readthedocs.io/en/latest/reference/services/ses.html#ses.

The notification email must include the job ID, linked to the page that displays the job details (see Exercise 5 below). Run the **`results_notify.py`** script on a separate instance and tag this instance as **`<username>-Utilities`** since we will run other scripts here also.

4. (6 points) Display a list of all jobs for a given user.

Create a page that displays a list of all the jobs submitted by a given user; 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

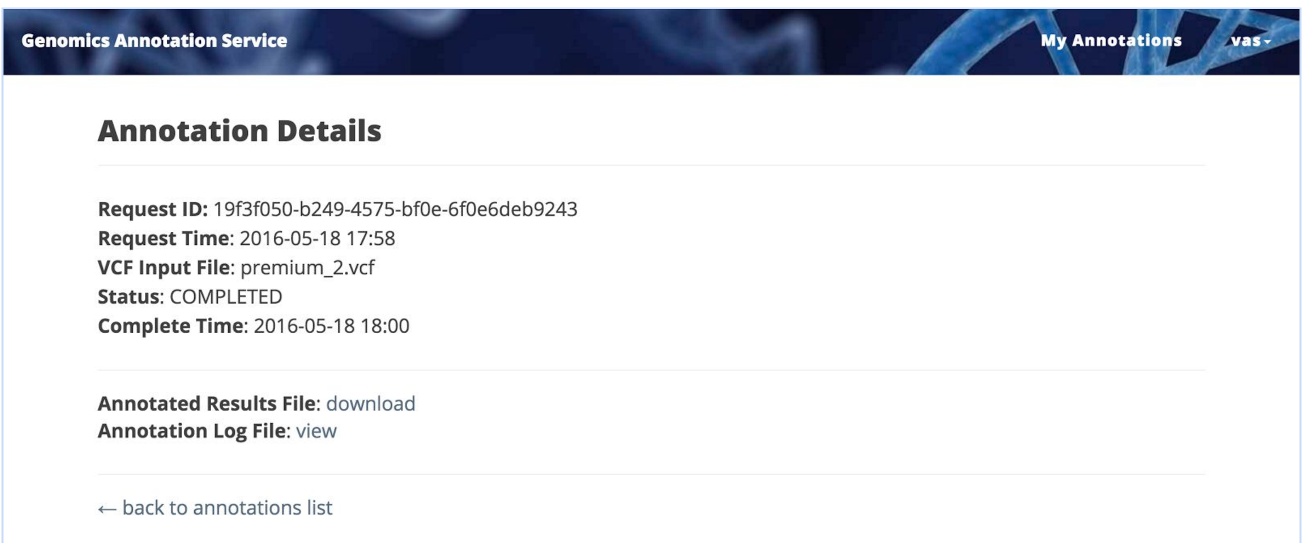
The page should display a table that contains four columns: the job ID, the date/time submitted, the input file name and the job status. The job ID should be hyperlinked to a page that shows all the details of the job (see next exercise). Make sure you only return jobs for the currently authenticated user. *Note that you must use **query** functions (**not scans**) when retrieving items from DynamoDB.*

5. (10 points) Display job details.

Create a page that displays the details of a given job. The page should be rendered in response to a request like: `https://<my_gas_url>:4433/annotations/<job-uuid>`. It should list these attributes (as 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> (this must be a hyperlink that retrieves the input file)
> **Complete Time**: <date/time when job completed> (hide from display if job is not complete)
> **Results filename**: <results file name> (this must be a hyperlink that downloads the results file; see Ex. 6 below; hide from display if job is not complete)
> **Log filename**:<log file name> (this must be a hyperlink that displays the log file, see Ex. 6 below; hide from display if job is not complete)

The page should look something like this:



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”.

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

When the user clicks one of the filename hyperlinks 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 signed download URL, using a similar process to that for your input file upload).
- If the user clicks on the log file link, display the log file in the browser (hint: you can read the S3 object into a string and just return that in a simple template).

7. (20 points) Archive Free user data to Glacier.

Our policy for the GAS is that Free users can only download their data for up to 30 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 that periodically checks the completion times of jobs belonging to Free users and moves their results files from the **`gas-results`** bucket to Glacier.

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 can use any approach you like, but *you must describe the approach used and your reasons(s) for choosing this approach* in the notes you submit with your final project.

Important: Glacier does not provide a nice interface for searching/downloading your archived objects. For this reason you **must** capture the object's Glacier ID and persist it to the Dynamo DB item for the annotation job. You will need to do this for only the results file (not the log file), e.g.:

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

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. (35 points) Enable Free users to upgrade to Premium via Stripe payments.

The objective of this exercise is to demonstrate what is involved in integrating with a third-party SaaS system. 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! We will integrate the Stripe service (www.stripe.com) to manage all subscription and billing functions for the GAS. 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).

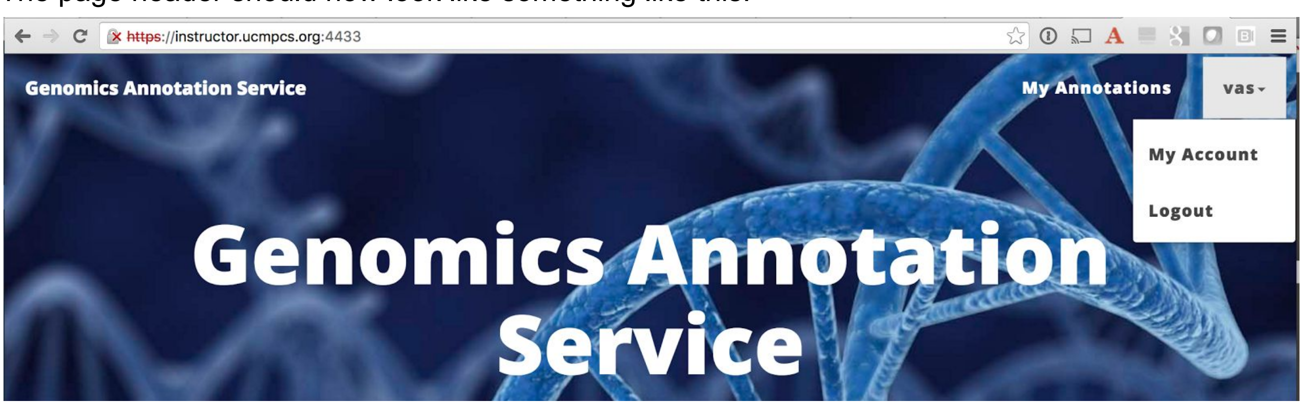
(a) Add a basic user profile page. All new users registering for the GAS are Free users by default and have the option to convert to a Premium user by clicking on a link in their profile page.

- **(3 points)** Create a `profile.tpl` template that displays the user's full name, username, and current subscription level. You can use the user's role to indicate the subscription level (by default this will be "free_user"). The template must also include a link to "Upgrade to Premium" that takes the user to the `/subscribe` page. This link must only be displayed for Free users; if the user's role is "premium_user", do not display the upgrade link.

- **(3 points)** Currently, the system displays a top-level menu option like "Logout: <username>", when a user is authenticated. Modify the navigation menu in `base.tpl` so that, instead of just a "Logout" link, the system displays the authenticated user's name which, when clicked, opens up a dropdown menu with two options: "My Account" and "Logout". The first option must take the user to `/profile` which displays the information above, and the second option routes to `/logout` (as it does now). This is how you create a dropdown menu using Bootstrap classes:

```
<li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown">Username<b
class="caret"></b></a>
    <ul class="dropdown-menu">
        <li><a href="#">My Account</a></li>
        <li><a href="{{get_url('logout')}}">Logout</a></li>
    </ul>
</li>
```

The page header should now look like something like this:



(b) (1 point) Go to www.stripe.com and sign up for a Stripe account.

(c) (1 point) Create a Premium plan in Stripe (https://dashboard.stripe.com/test/plans). Your plan should charge users $999.99 weekly. Make a note of your plan's ID (e.g. **`premium_plan`**) since you will use this in your subscription code below.

(d) (1 point) Locate the **test** API keys for your Stripe account (https://dashboard.stripe.com/account/apikeys). Update the section in the **`mpcs.conf`** file for your public and secret keys (make sure you use the test keys, not the production ones):

```
[mpcs.stripe]
public_key = pk_test_c9X....
secret_key = sk_test_d7A....
```

(e) (1 point) Add the Javascript code required by Stripe. Add the code block below to the bottom of the **`scripts.tpl`** file in the section marked **`<!-- ADD STRIPE JS CODE BELOW -->`**. *Note: Substitute your own Stripe API key in the highlighted line*.

```
<!-- Stripe code -->
<!-- This includes the main Stripe bindings -->
<script type="text/javascript" src="https://js.stripe.com/v2/"></script>

<script type="text/javascript">
// This identifies our web app in the createToken call below
Stripe.setPublishableKey('<your_stripe_pk_test_key_here');

// This function adds the Stripe token to the form as a hidden field so we
// can access it in our server code. If there's an error in the payment
// details it displays the errors in the "payment-errors" class
var stripeResponseHandler = function(status, response) {
  var $form = $('#subscribe_form');
  if (response.error) {
    // Show the errors on the form
    $form.find('.payment-errors').text(response.error.message);
    $form.find('#bill-me').prop('disabled', false);
    $form.find('.payment-errors').html('<div class="alert alert-error
alert-block"><button type="button" class="close"
data-dismiss="alert">&times;</button><h4>We could not complete your
request.</h4>' + response.error.message + '</div>');
  } else {
    // Get the Stipe token
    var token = response.id;
    // Insert the token into the form so it gets submitted to the server
    $form.append($('<input type="hidden" name="stripe_token" />').val(token));
    // Re-submit the form to our server
```

```
    $form.get(0).submit();
  }
};

// This function intercepts the form submission, calls Stripe to get a
// token and then calls the stripeResponseHandler() function to complete
// the submission.
jQuery(function($) {
  $('#subscribe_form').submit(function(e) {
    var $form = $(this);
    // Disable the submit button to prevent repeated clicks
    $form.find('#bill-me').prop('disabled', true);
    Stripe.createToken($form, stripeResponseHandler);
    // Prevent the form from submitting with the default action
    return false;
  });
});
</script>
```

(f) (8 points) Create a template called `subscribe.tpl` that will be used to get the user's credit card information. The template must include a `<form>` that captures the following information:

- Name on credit card
- Credit card number
- Credit card verification code
- Credit card expiration month
- Credit card expiration year

The Stripe Javascript code included above provides us with some magic for this 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. We denote these fields using an attribute that starts with `data-stripe="<some-Stripe-field-name>"`. For your form you must include the following data-stripe attributes in the respective input fields:

- `data-stripe="name"` for the name on the credit card
- `data-stripe="number"` for the CC number
- `data-stripe="cvc"` for the CC verification code
- `data-stripe="exp-month"` for the CC expiration month
- `data-stripe="exp-year"` for the CC expiration year

So, for example, your Credit Card Number field should look something like this:

```
<input class="form-control input-lg required" type="text" size="20"
data-stripe="number" />
```

Very Important Note: Do not include `name="...."` or any other ID attributes in your subscription form fields. This will prevent the Stripe code from working correctly.

Additional important requirements:

- Your form tag **must** have the id and name attributes set as follows: **`id="subscribe_form" name="subscribe_submit"`**, e.g. something like:

```
<form role="form" action="/subscribe" method="post" id="subscribe_form"
name="subscribe_submit">
```

- The submit button on your form **must** have the id attribute set as: **`id="bill-me"`**, e.g. something like:

```
<input id="bill-me" class="btn btn-lg btn-primary" type="submit"
value="Subscribe>
```

Note: If you use other values for the form and submit button IDs, you will need to update the Stripe Javascript code to use them.

(g) (12 points) Add a Bottle route in **`mpcs_app.py`** that processes the submitted billing information, e.g. **`@route("/subscribe", method="POST")`**. This must perform the following actions:

- Extract the Stripe token from the submitted form. This is a special-purpose field that is automagically created by the Stripe Javascript and is a one-time use token for accessing the Stripe API securely. The field is called **`"stripe_token"`**.

- Create a new Customer in Stripe using the the token, your Stripe API key and the data entered by the user on the form. Note: the credit card data are all stored in serialized, encrypted form in **`stripe_token`**, so you don't need to extract any other field values from the form. When you call the Stripe API it will create a new Stripe customer, populating the required credit card information from the token, setting the customer's subscription plan ID (use your own plan ID here), and adding optional information such as the user's email and GAS username (both of which we retrieve from own own **`auth`** object). If successful, this call will return a Stripe **`Customer`** object that contains lots of info, including the customer's ID.

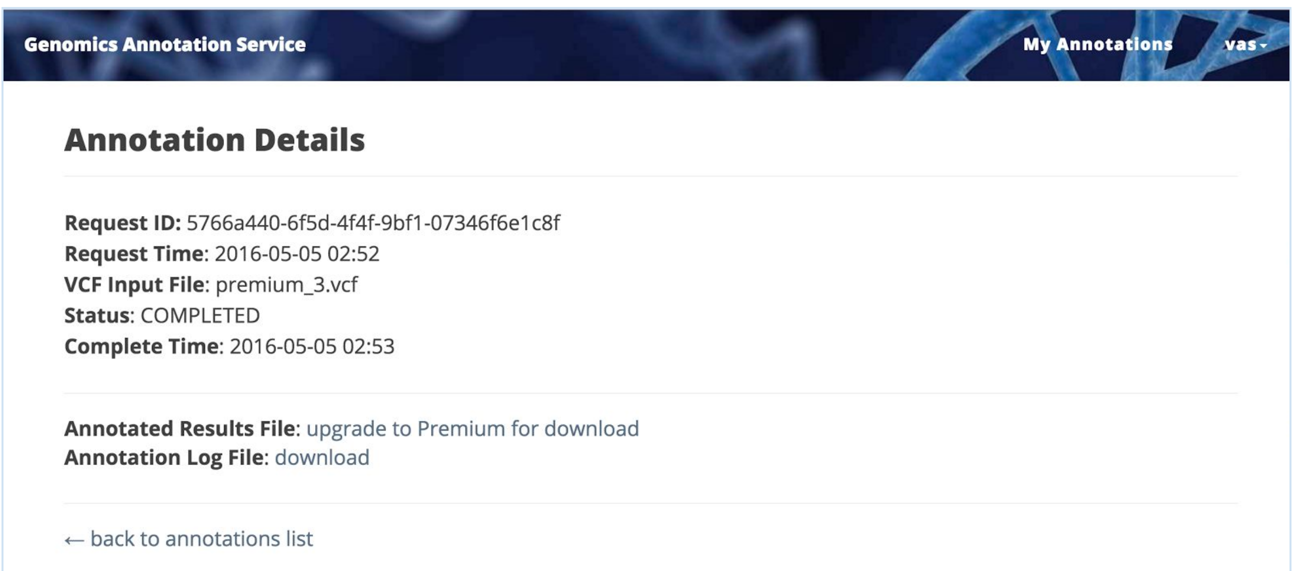
- Update the user's profile in our user database. We need to change the user's role to "premium_user" to indicate that this is a paying subscriber:

```
auth.current_user.update(role="premium_user")
```

Note: In a real application we would also store the Stripe customer ID in our users database so we can retrieve the user's billing records, if necessary. However, for the purposes of this project we're ignoring this.

(h) (5 points) 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 30 minutes after completion. (Yes, this is a ridiculously restrictive policy but we're using such a short interval so that we can test the functionality in the limited time we have available for the project).

With this in mind, modify the 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 30 minutes have passed since the annotation job completed. If so, display the link to upgrade; if not, display the link to download the results file. Note: The log file should always be available for viewing, irrespective of the user's Free or Premium account status.

9. (20 points) Restore data for users that upgrade to Premium.

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

1. Initiate a Glacier archive retrieval job (see: http://boto3.readthedocs.io/en/latest/reference/services/glacier.html#Glacier.Client.initiate_job). This asks Glacier to "thaw" the object and make it available for download.
2. After the object is restored from Glacier, download the object to the user's directory in the `gas-results` bucket. Note that when restoring a Glacier object, a copy of the object exists

only for a limited period (typically around 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 any results files that were generated while the user was Free user. This will require modifications to your code that processes a subscription request, after the user is created in Stripe.
- A separate script must check on the status of restored(ing) archives and move those archives are completely restored to the user's results bucket.

Once again, there are many ways to accomplish these tasks, each with their own tradeoffs -- you can use any approach you like, but *you must describe the approach used and your reasons(s) for choosing this approach* in the notes you submit with your final project.

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. (12 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) (6 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 the user data that will run when new instances launch. 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:

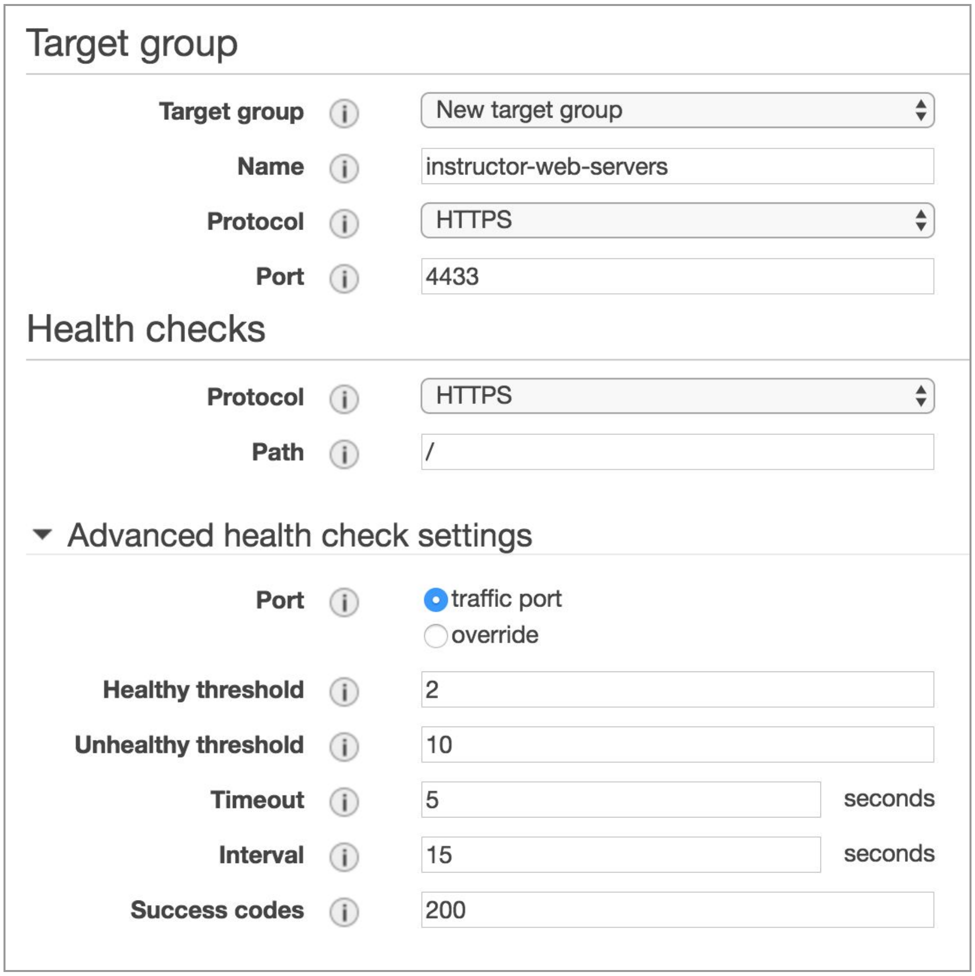
- Upload your GAS code to S3. You can upload it to your home folder in the **`mpcs-students`** bucket. Before uploading, create a single ZIP file (e.g. in the sample below I call this file **`gas_web_server.zip`**) that contains all the required source code such as *.py, views/*, static/*, etc.

- Create the user data file that will be used by new instances launched via Auto Scaling. Name this file **`auto_scaling_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/mpcs/...`**,
    - Run the web server app. This must be the last command in **`auto_scaling_user_data_web_server.txt`** and should look something like: **`sudo -u ubuntu /home/ubuntu/mpcs/run_gas.sh &`**

Feel free to start with the user data that was provided for launching capstone project instances in the Preparation section above, and modify/extend as needed.

(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 "Continue".
- Enter a load balancer name (prefixed by your AWS username, e.g. **`instructor-elb`**).
- Under Load Balancer Protocol select "HTTP (Secure HTTP)"; port will change to 443.
- Select all the availability zones and click "Next: Configure Security Settings"
- Select "Choose an existing 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 `<username>-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.

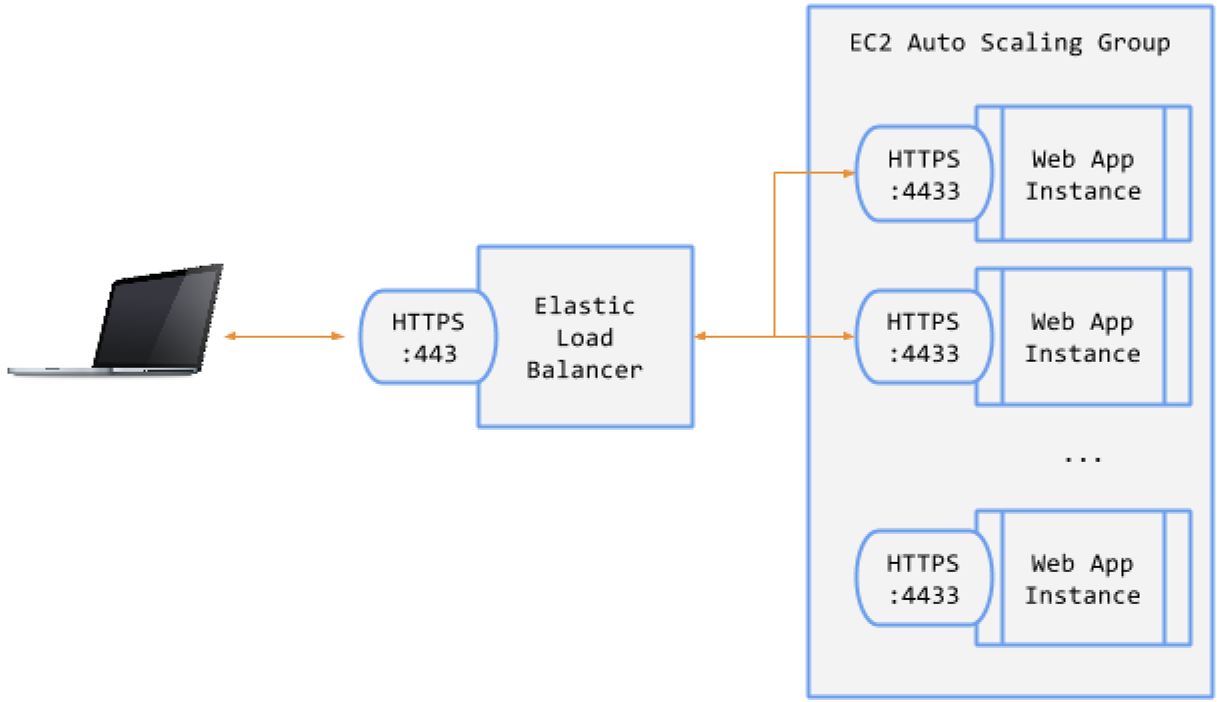
Auto Scaling groups encapsulate the rules that determine how our application scales out (when demand increases) and in (when demand drops). Launch configurations provide a means to save your standard EC2 instance launch procedure so that it can used by the auto scaler when launching new instances.

- 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 the “MPCS Capstone 05/2017 - ami-e3b1c7f5” AMI.
- Select t2.micro as the instance type and click “Next: Configure details”.
- Name your launch configuration **<username>-launch-config**.
- Under “IAM Role” select your instance profile, e.g. **instance_profile_instructor**
- 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 **auto_scaling_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 **<username>-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 MySQL (RDS) database traffic on port 3306.
- 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	3306	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 **<username>-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 “**<username>-capstone-webserver**”. 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://instructor-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.

<u>Troubleshooting the configuration if your load balancer is not responding</u>:

- 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.

VERY IMPORTANT NOTE

When you have confirmed that your load balancer is working, let me know so that I can associate your personal subdomain (**`<username>.ucmpcs.org`**) with your load balancer. This way you can access your GAS using a more convenient URL (and it will be required for load testing later).

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 is a very unrealistic 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: You do not need to 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 first time around) ***make sure you select the CloudWatch metric for your current/active ELB****. 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. Make sure you set the "Instances need" parameter to 120 seconds - this specifies how long the auto scaler waits before considering adding a another instance to address the alarm.

(c) (2 points) Create another CloudWatch alarm on your load balancer that is triggered when the average latency 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 alarm in 2(c) above is triggered.

12. (18 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; the auto scaler will simply add and remove instances based on the number of messages arriving in the job requests queue.

(a) (6 points) *Preparing the annotator instances for automated launch*: This is similar to the automated deployment of the web server instances, but you only need to deploy the AnnTools files,

job_runner.py, run.py -- the SSL certificate files are not required since the annotator instances do not serve HTTPS requests. As before:

- Create a ZIP archive containing the annotator files (call it **gas_annotator.zip**) and put it in your home folder on S3; don't forget to make it public.
- Copy **auto_scaling_user_data_web_server.txt** to a new file, **auto_scaling_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 since the user data commands run as root, e.g. **chown -R ubuntu:ubuntu /home/ubuntu/mpcs/...**
- Add a command to your user data that runs the annotator script. This must be the last command in the file and should look something like:
  sudo -u ubuntu python /home/ubuntu/anntools/job_runner.py &

(b) (2 points) Create a launch configuration for annotator instances. You can use the same instructions used for the web server launch configuration, except that the annotator does not require any ports 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. 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, *you must use a tag for your auto scaled instances* called "Name" with the value "**<username>-capstone-annotator**", e.g. **instructor-capstone-annotator**.

(d) (2 points) Create a CloudWatch alarm to *scale out* 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 "NumberOfMessagesReceived" metric and click "Next".~~
- Select the "NumberOfMessagesSent" metric and click "Next".
- Set the parameters so that the alarm triggers when more than 10 messages are sent to the job request queue ~~in one minute~~ in five minutes.
- Under "Actions", delete the default notification and click "Create Alarm".

(e) (2 points) Create another CloudWatch alarm to *scale in* the annotator server farm when the number of messages received is below 5 for 2 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 11(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 11(e) above is triggered.

13. (18 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 will use a simple load testing tool called Locust. This is not an all-singing, all-dancing testing 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) (1 point) 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) (5 points) 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 an EC2 instance using this AMI **`ami-5199dc47`** (and use the mpcs security group).
2. SSH into the instance and edit **`locustfile.py`**. You can use the default file as is, but I encourage you to experiment with testing different routes - by default Locust will only test against our registration page (**`/register`**).
3. When you're ready to test, run locust: **`locust --host=https://<username>.ucmpcs.org`**

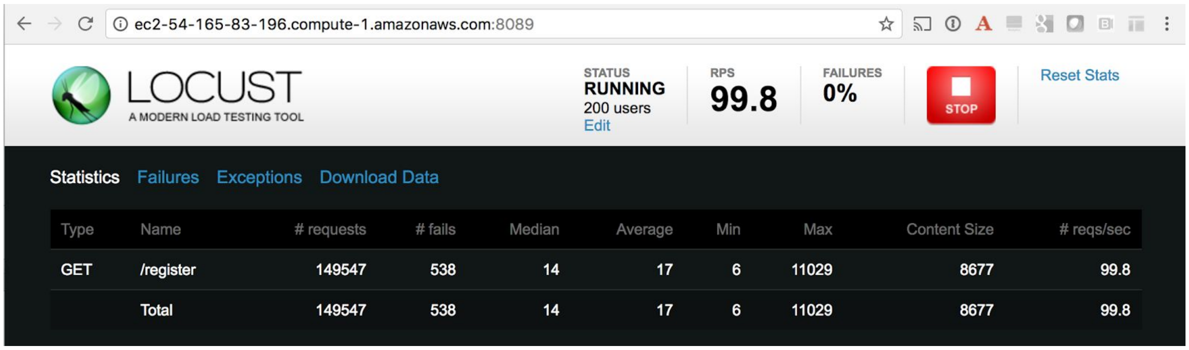
```
ubuntu@ip-172-31-17-43:~$ locust --host=https://instructor.ucmpcs.org
[2017-05-20 10:40:52,487] ip-172-31-17-43/INFO/locust.main: Starting web monitor at *:8089
[2017-05-20 10:40:52,487] ip-172-31-17-43/INFO/locust.main: Starting Locust 0.7.5
```

4. In your browser, navigate to the Locust console; your Locust server is listening on port 8089, e.g. **`http://ec2-54-165-83-196.compute-1.amazonaws.com: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 (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) (6 points) Monitor your auto scaling group and note when instances are added. Briefly describe what you observe, and explain why you think instances launch (or not) when they do.

(e) (6 points) After observing scale out behavior, stop your test by hitting the red STOP button the web console. Monitor your auto scaling group and briefly describe what happens when your scale in policy goes into effect.

(f) Capture screenshots of the Locust web console showing your tests and include them in your writeup. Also capture some screenshots of the AWS console that show how the instances in your auto scaling group grew and shrank at various points.

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.

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 are really close to the limit on the number of concurrent EC2 instances and your fellow students may be prevented from running their tests.

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 Python script that submits an arbitrary number of messages to the request queue (simulating job submissions). You can use hardcoded, repetitive test data, i.e. same input file on S3, fixed username, etc. Run your script and see how the auto scaler behaves.

Final Deliverables

On completion of the project you're expected to submit the following:

1. A **fully-functional GAS** accessible at https://<your-username>.ucmpcs.org. Your environment will include three types of instances:
    a. Two instances running the web application
    b. Two instances running the annotator
    c. (at least) One instance running the utility scripts

    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 if your app is not reachable at your subdomain. If you do not get to the point where your ELB is working, please note this at the top of your write up and give us the DNS of an instance that we can access directly via port 4433.

2. **Full source code** committed to GitHub. Include:
    a. All Python code (including framework code that you did not change)
    b. All templates (including the ones provided with the framework)
    c. Configuration file(s)
    d. AWS EC2 user data files
    e. Any other files necessary for your GAS to run

    Reminders:
    - Include basic error trapping and handling code for critical actions.
    - Do not hardcode anything that doesn't need to be hardcoded; credential hardcoding will result in -10 points. For the utility scripts you may include hardcoded global variables at the top of the script for things like SQS queue names (but if you want higher quality code, use environment variables :-).
    - Please clean up your code (remove unused functions from old homework, etc.).
    - Add comments, especially for new code, e.g. job listings, file downloads.

    List all **references**, ideally as inline comments. Failure to cite references where we expect you to have consulted outside sources *will result in -5 points per occurrence*.

3. Write-up and screenshots from the load testing exercise.

4. Include a **short write-up** (as README.md or PDF) that explains any aspect of your work you wish to elaborate on, e.g. the approach used to handle archiving of results files for Free users. In particular, I encourage you to submit notes if you don't manage to complete all the exercises and wish to describe what issues you encountered that prevented you from doing so.

Instructor Grading Rubric

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

1. Register a new user.
2. Get email confirmation notice; click to confirm new user registration.
3. Login using the new username.
4. Run one or more annotations.
5. Receive notification email when annotation(s) is(are) complete.
6. View a list of annotation jobs.
7. Click on a job in the list and view the details.
8. Click on a link to download the input/results files, and view the log file (this should work for all Free users right after results are received).
9. Revisit the list of annotation jobs, after allowing about one hour to pass. Click on a link to download the results file (this should not work; I should be asked to upgrade, ideally just by displaying an upgrade link instead of a file link); I should still be able to view the log file.
10. View the user's profile; click the link to upgrade to a Premium user.
11. Provide the credit card info and upgrade to a Premium user.
12. View the jobs list again after ~6 hours, to allow 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 two notification topics (one for job requests, one for job completions) and subscribed the corresponding message queues to their topic.
- You have a running load balancer with at least two instances in service.
- You have two Auto Scaling groups, each with two instances: one for the web app and another for the annotator.
- All your instances are properly name-tagged *by your auto scalers* during launch.
- You have separate instance(s) running the results notifier and the archive 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 will use manual loading to test the annotator auto scaling group if you have not provided a script as part of exercise #14.