# 2019 -CS352 - Assignment 2: SelfieLessActs on AWS

| Member Name | USN | Section | #hrs effort |
| --- | --- | --- | --- |
| Maanvi Nunna | 01FB16ECS187 | D | 35 |
| Meghana Ganesh | 01FB16ECS205 | D | 12 |
| Nandagopal NV | 01FB16ECS221 | D | 35 |
| P Abhishikta Sai | 01FB16ECS242 | D | 25 |

Date of Evaluation : 16/02/2019 Evaluator : Srinivas Shekar

1. **AWS username on which demo was shown**

Maanvi Nunna - [maanvi7062@gmail.com](mailto:maanvi7062@gmail.com)

1. **How is statelessness implemented in your API Design**

We stuck to the spec for the APIs. This was mostly designed in such a way as to ensure that the server need not do anything to maintain state. Every single time we make an API call, all the information needed by the server is sent, so that the server doesn’t have to keep track of previous transactions. For example, the client is expected to specify the username for creating acts. To maintain state of the user, without resorting to a server-side overhead, we made use of cookies for the time-being. This is obviously not the right way of doing it, but to make use of sessions, we would have had to use some server. In contrast, our method is purely uses the client. We understand the risks associated to such an approach and only implemented this as a way to showcase the interaction of our UI with the REST APIs built as part of this assignment.

We had to add in a few extra APIs.

Particularly:

1. GetAllActs - GET

-Serves all acts in reverse chronological order, irrespective of which category they belong to.

-Very similar to the Get-all-acts-in-category API as in the spec.

2. checkPassword - POST

- Retrieves the password for the given username and checks it against the password typed in by the user. This aims at performing authorization in a very rudimentary manner.

- Sends back the necessary status codes in case the username doesn’t exist or when the password is incorrect and when all the credentials match the database entry, status code informing success is returned.

1. **Summarize your learning as part of this assignment**

We learnt how to use FLASK to serve REST API calls. We used MySQL to store our database. We also made use of Python FLASK’s mysql module to interact with the DBMS. Initially, after writing a couple of APIs and testing them, we realized that the database isn’t automatically refreshed after each consequent API request. After a little search we found out that we need to commit the state of the database using conn.commit() following the execution of queries that make a change to the database, and hence our problem got fixed and the changes made through the API calls were seamlessly reflected in our database.

Next, we also learnt that MySQL does not allow a connection to be open for too long, and also has a timeout property, i.e, if it’s idle for longer than the timeout, it terminates the connection and when it unexpectedly closes connection, the next query in the same API is not serviced. We stumbled on this when we tried to see how fast our server was able to service requests. When we clicked too fast, it started throwing an error due to the aforementioned property of MySQL. After searching on this issue, we found the solution of establishing and closing the connection to the MySQL database in every API call and the problem was fixed.

Since our front-end includes only JavaScript, the way we talk to the backend is by making AJAX requests. When making an AJAX request, we have to mention the type of data being sent using the GET or POST or any other method. Especially in one API where you can upvote an act, the request is just one number that is the ACTID in an array. For this, we had to change the content-type and datatype as FLASK was unable to identify the data being sent, and printed it as a NoneType object.

One other problem we had was handling images of different formats. Due to the nature of the spec of UploadAct API, we couldn’t specify the extension of the target file. This meant that the API has no regard for the format of the image as it only accepts base64 encoded strings, which don’t preserve information on the actual format of the image. After some quick testing, we realised that the decoding from a base64 string to an actual image depends on the target format and it simply won’t work if we attempt to convert a jpeg image to a base 64 string and decode it into a png. It results in a corrupt image. Since we were told to strictly adhere to the spec, we had no choice but to add a restriction on the format of the image. For this purpose, we chose png as the most suitable format because unlike jpeg, png supports transparency and is less susceptible to jpeg’s rather harsh compression.

We originally took it for granted that the EC2 instance already has the required inbound rules on ‘HTTP’. We quickly realised that we’d have to explicitly add a security rule that allows inbound requests on port 5000(which is where our flask service was listening).

**Any other observations/challenges/comments**

Found this assignment extremely time consuming. The APIs themselves were easy enough to program once we understood an example API. However, integrating it with the UI, setting it up on the VM, ensuring proper service, handling exceptions on the UI side proved to be quite time consuming. We also noticed considerable lag when using the web page to make API calls. This seems to be because of network delays but we aren’t certain. The APIs perform quite well on our local machines.

Another challenge we faced was that by default, CORS(Cross-Origin-Resource-Sharing) is not allowed and flask will straight up deny requests from other webpages. We needed to figure this out and we eventually fixed it by including some headers.