CS 310
Scalable Software Architectures
Fall 2023

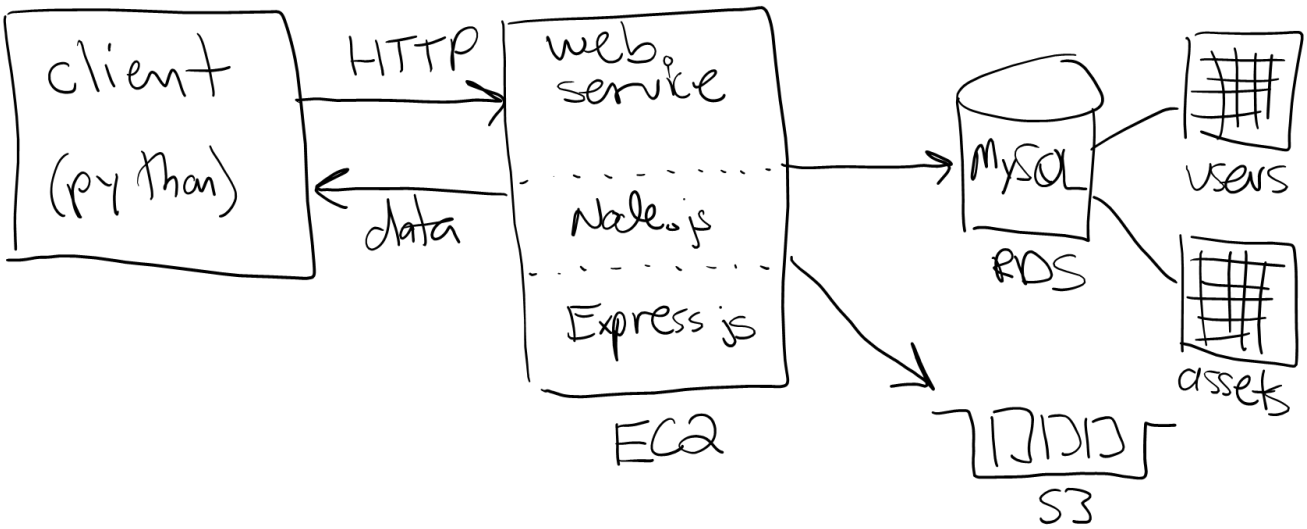
Project #02 --- Part 01 (v1.0)

Assignment: Multi-tier PhotoApp with AWS EC2, S3 and RDS
Submission: via Gradescope (unlimited submissions)
Policy: individual work only, late work is accepted
Complete By: Part 01 by Friday October 20th @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Sunday 10/22 @ 11:59pm

Pre-requisites: Lectures 05, 06, and 07. See Canvas for lecture recordings and links to PPT.

Overview

In Project 01 we built a client-server (two-tier) PhotoApp where the client-side Python code interacted directly with AWS, in particular the RDS and S3 services. Here in Project 02 we're going to inject a web service tier between the Python-based client and the AWS services:



As we've discussed in class, the web service will be written in JavaScript using Node.js and the Express framework. The client is still Python-based, rewritten to interact with the web service instead of AWS directly. The database and S3 bucket remain unchanged from Project 01.

Before we start…

We're going to approach Project 02 much like Project 01 --- in steps. Here are the major steps:

1. *Build the web service, running either in replit or on your local machine. This way it's easy to run, test and debug. You will test using a web browser as the client, not the Python-based client.*
2. *Build the Python-based client, providing a better way to test. You'll also be able to confirm images are downloaded properly by displaying.*
3. *Add another feature (image upload). Update web service, test. Update python client, test.*
4. *Package and deploy using AWS Elastic Beanstalk / EC2, making it available to the world.*

Steps 1 and 2 are the focus on this handout (part 01). Steps 3 and 4 are the focus of part 02. Please note that parts 01 and 02 will have *different* due dates. Part 01 (this handout) has a due date of Friday 10/20 (max late submission of Sunday 10/22). Part 02 (future handout) will have a due date of Friday 10/27 (max late submission of Sunday 10/29).

Getting started (server-side)

For the server-side web service, a total of 10 files are being provided so you have a framework in which to work. You'll find these files on replit under "**Project 02 (server)**", or in this dropbox folder. The files:

app.js	api_stats.js	database.js
api_asset.js	api_users.js	photoapp-config.ini
api_bucket.js	aws.js	
api_download.js	config.js	

The web service's main file is "**app.js**" (in class it was "index.js", but has been renamed based on AWS preferences). This file starts listening on the proper port, and registers the web service functions (API) we are defining:

```
/stats
/users
/assets
/bucket?startafter=bucketkey
/download/:assetid
```

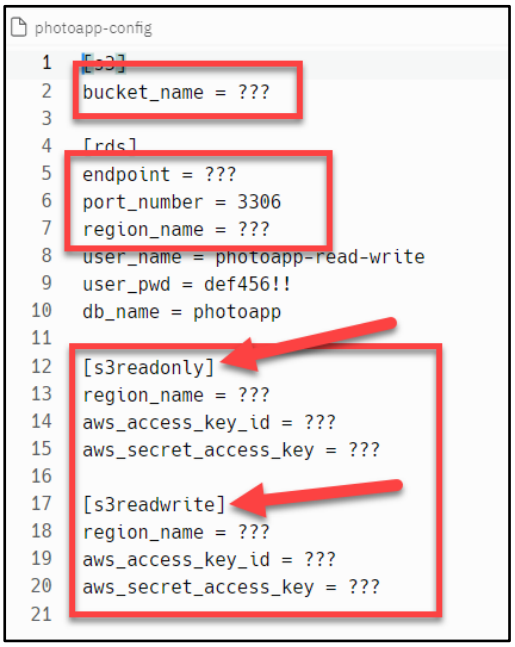
We talked extensively about **/stats** in class, and this handler is completely implemented in "**api_stats.js**". For modularity, each web service function is defined in a separate .js file. You'll want to review the code in "api_stats.js" (and the lecture notes if necessary) before implementing the other functions in the API.

The remaining .js files --- aws.js, config.js, and database.js --- create the necessary S3 and DB objects for talking to S3 and MySQL, respectively. Review but do not modify these files. Finally, we have a config file much like we did in Project 01: **photoapp-config.ini**. Continue reading before doing anything…

Configuring the server

The handout is going to assume you are working in **replit**. You are free to work outside of replit, but in that case you'll need to setup an environment running Node.js and Express js, then download the code from dropbox and get it running. If you run into problems, you're on your own --- for our own sanity the course staff is only supporting replit. That said, it should be straightforward e.g. to get Node.js up and running in VS code.

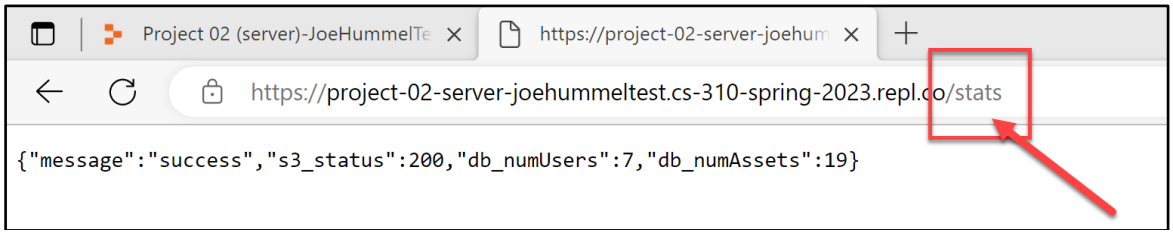
Open the team project "Project 02 (server)". The first step is to update "photoapp-config.ini" file based on your configuration from Project 01. You need to replace each ??? with your AWS values --- bucket name, RDS endpoint, etc. Feel free to copy-paste or upload your photoapp-config.ini file from Project 01 ---------->



Once you have edited **photoapp-config.ini**, test as follows. First, make sure your RDS service (database) is running. Now, back in replit, stop the service if it's running. Then run. The app will read the config file and replit will open two windows on the right-side --- you should see a client-side browser and a console window:

The upper window acts like a client and browses to the home directory "/" --- this will show the status of the app, how long it's been running, and the state of the database connection. You can refresh this window at any time. More importantly is the "baseurl" shown --- that's the endpoint for your web service. Copy this URL and paste it into a new browser tab (or click the little "pop-out" button denoted in the screenshot). Append **/stats**

to the URL, and this will call the /stats API function in the web service. You should see something similar to the following (but not identical since your bucket and database will differ from ours):



If you see something like the above, then all is well with your configuration and you can continue with the next section. If adding /stats doesn't work, then either (a) your configuration is wrong, or (b) something is wrong with your internet connection.

Web service API

The goal of your web service (here in Part 01) is to support the following five API functions. Here are the paths (also called routes):

/stats
/users
/assets
/bucket?startafter=bucketkey
/download/:assetid

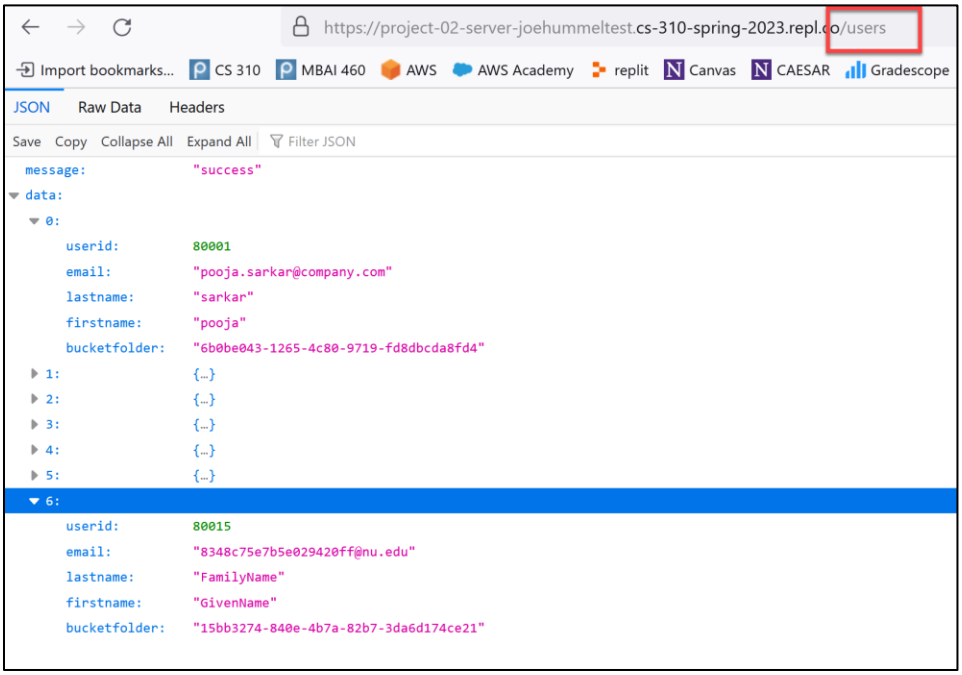
Each API function is defined by a corresponding .js file. For example, /users is implemented in the file "api_users.js". Here's the contents of that file, which defines the handler to throw an error until implemented. Note that the object representing the database is named "dbConnection" (the code presented during lecture used "photoapp"):

```
//
// app.get('/users', async (req, res) => {...});
//
// Return all the users from the database:
//
const dbConnection = require('./database.js')

exports.get_users = async (req, res) => {

  console.log("call to /users...");

  try {

    throw new Error("TODO: /users");
```

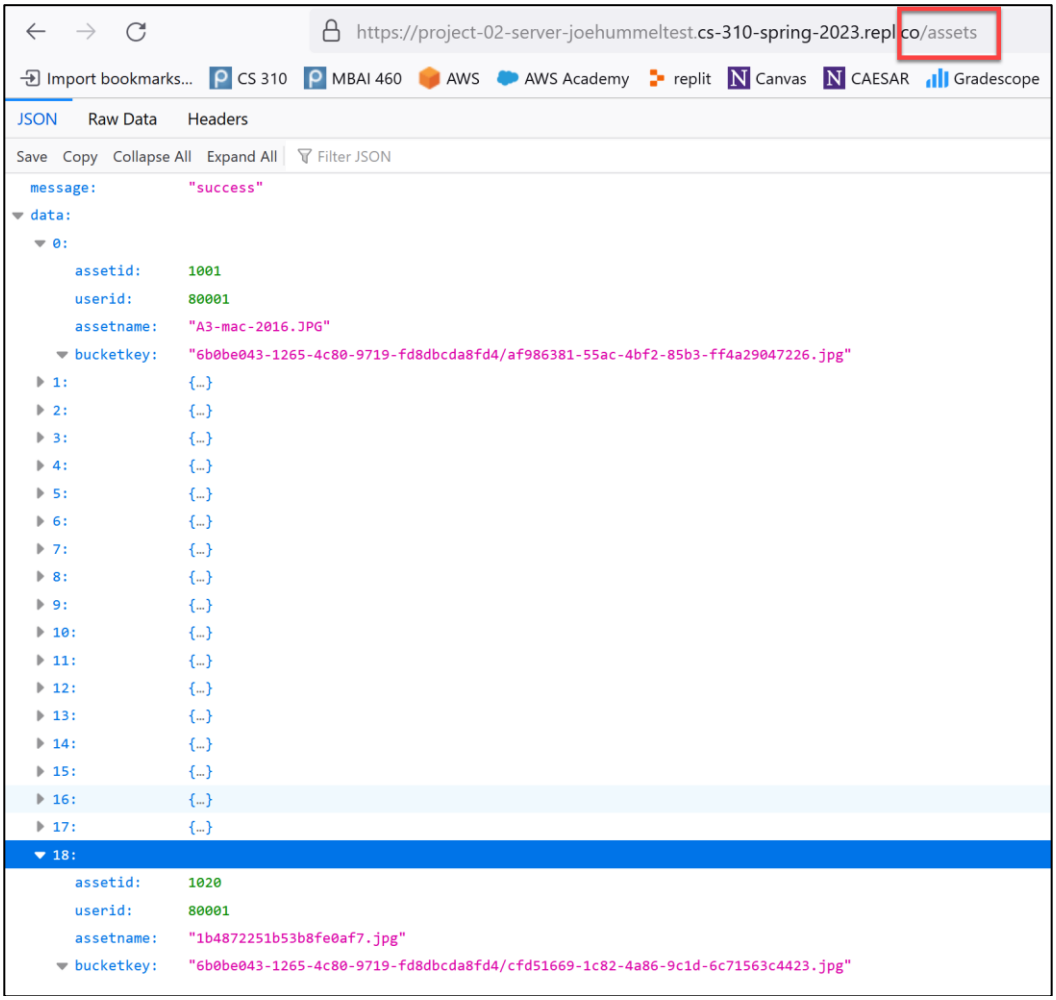
```
    //
    // TODO: remember we did an example similar to this in class with
    // movielens database (lecture 05 on Thursday 04-13)
    //
    // MySQL in JS:
    //   https://expressjs.com/en/guide/database-integration.html#mysql
    //   https://github.com/mysqljs/mysql
    //

  }//try
  catch (err) {
    res.status(400).json({
      "message": err.message,
      "data": []
    });
  }//catch

}//get
```

Notice the "TODO" comment contains links to one or more online resources to help you implement the function; each .js file has similar links to supporting resources. The recommended order of implementation is as follows: /users, /assets, /bucket, and /download.

The first two functions --- **/users** and **/assets** --- are single calls to MySQL, selecting all columns for all users or all columns for all assets. Retrieve the users in ascending order by userid; retrieve the assets in ascending order by assetid. Test your code using the client browser tab, appending /users or /assets to your web service's baseurl. Your output should look similar to the screenshots on the next page. Note that the screenshots were made using **Firefox**, which defaults to a more readable display of the response. Here's a possible response for **/users**:

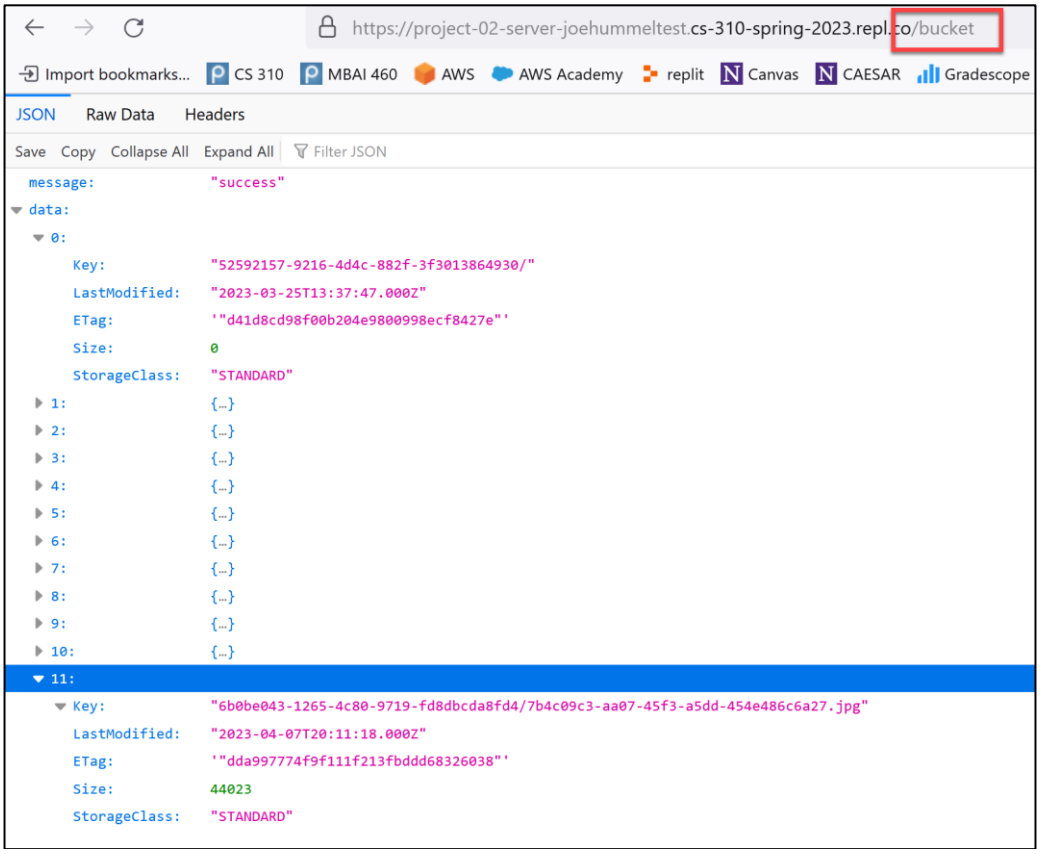


And here's a possible response for **/assets**:

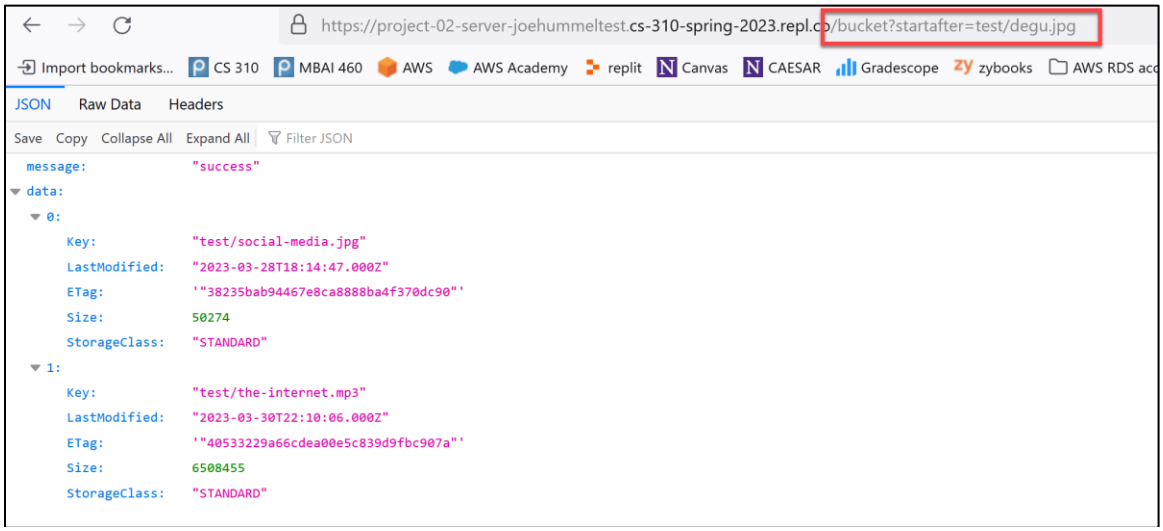


<< continued on next page >>

The **/bucket** API function returns information about each asset in the bucket: Key, LastModified date, etc. However, instead of returning all data in one call, data is returned a page at a time where a page is defined as at most 12 assets. This implies that the path /bucket returns information about the first 12 assets in the bucket (the assets are always returned in alphabetical order by key):

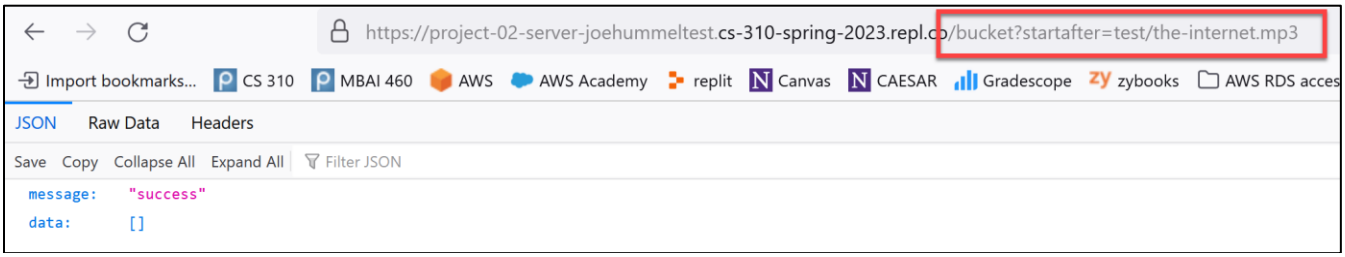


The client can retrieve the next page of data by using a query parameter **?startafter=bucketkey**, where bucketkey is the last key in the previous page. Recall that lecture 06 discussed how to retrieve query parameters in JavaScript. Here's an example URL:



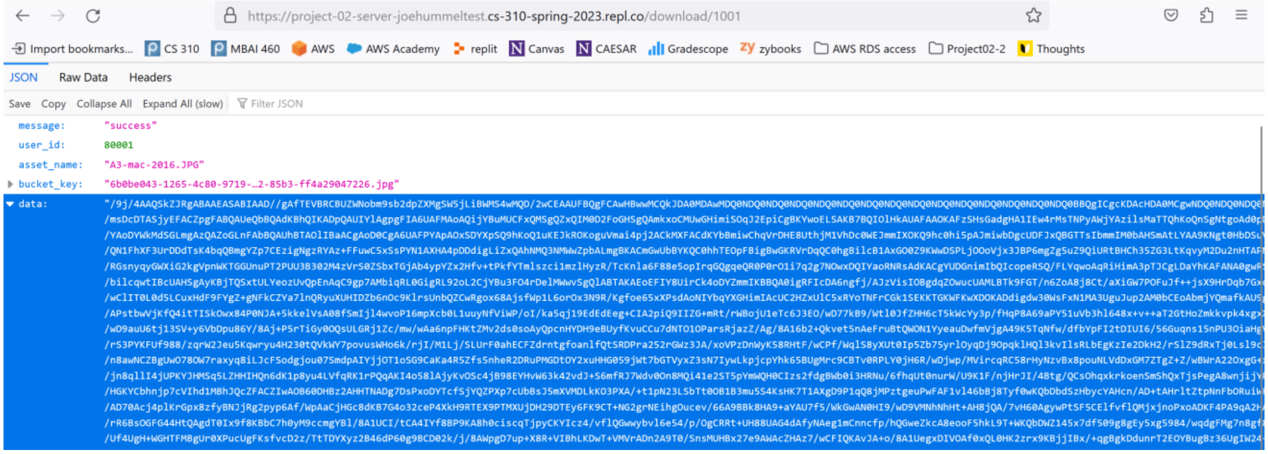
The /bucket API function is implemented using S3's **ListObjectsV2** command; see the TODO section in "api_bucket.js" for resources on how to use this command. You control the page size via the *MaxKeys* parameter within the input object (S3 commands take an input object that controls how the command behaves); the "startafter" functionality is specified by providing a *StartAfter* parameter as well. Note that S3 may respond with fewer than 12 assets; check the *KeyCount* in the response to see how many assets were actually returned. The data itself is returned in the response *Contents*.

NOTE: if KeyCount is 0 then Contents does not exist, so you'll need to handle that as a special case to properly return an empty list to the client if the last page is empty (i.e. the client happens to pass the last key in the bucket as the value to "startafter"):



The last API function is **/download/:assetid**, which uses the assetid to lookup the asset's bucket key in the database, and then calls S3 to download this asset and then send it to the client as a base64-encoded string. Recall that this path syntax implies assetid is a parameter to the API function, and passed as part of the path. For example, to download asset id 1001, the client appends **/download/1001** to the baseurl as shown below:

The image data is contained in the **data** element of the response --- in the screenshot above you only see the first 50 characters or so because the element is not expanded (Firefox allows the elements to be expanded or contracted). Expanding data in this case would reveal a string with over 100,000 characters (and this is a small image):

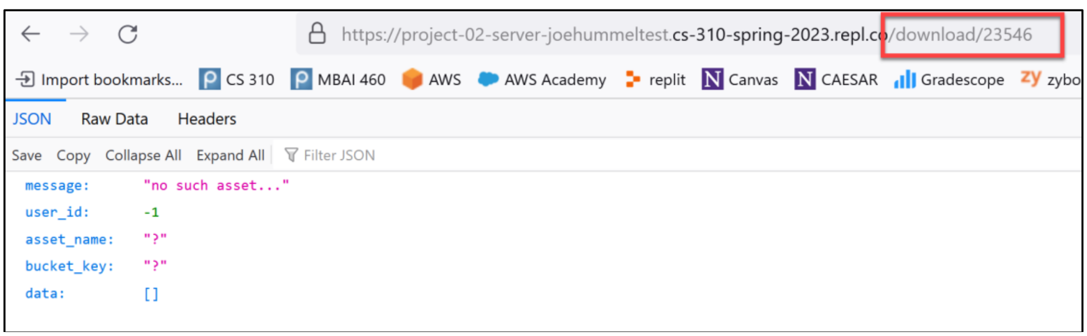


When the parameter is part of the path it's known as a URL parameter; how to retrieve a URL parameter is discussed in Lecture 06. To download an asset from S3, use the **GetObject** command, see the TODO comment in "api_download.js" for documentation and example code. After you await for the result from s3.send( ), the next step is to transform the Body of the result into a base64-encoded string so it can be sent to the client. This is also an asynchronous process, so you have await for it to finish:

```
var datastr = await result.Body.transformToString("base64");
```

Welcome to asynchronous programming! [*BTW, you might be wondering… Why can't we send the raw bytes? This is one of the impacts of building our service as a web service. The HTTP protocol for request/response limits the range of characters that can be sent, so raw binary data cannot be transmitted over HTTP. Base64 limits the character set to 64 chars, which is safe. The tradeoff is the resulting string is longer.*]

What happens if the asset id is invalid? Your response should look as follows (with a status code of 400):



There's no way (right now) to test and make sure the image is properly downloaded from S3 and encoded. That's one reason we need a Python-based client (next step) so we can decode and display the image and see if it's correct.

At this point your web service should be implemented and working. Now it's time to focus on the client-side.

Getting started (client-side)

For the Python-based client, only 2 files are being provided. You can find these files on replit under "**Project 02 (client)**", or in this dropbox folder. Here are the files:

main.py photoapp-client-config.ini

Edit "photoapp-client-config.ini" to contain the baseurl for **your** web service; right now this is the URL to your replit server (something like **https://project-02-server-...-fall-2023.repl.co/**). When you're done, your .ini file should look like this:

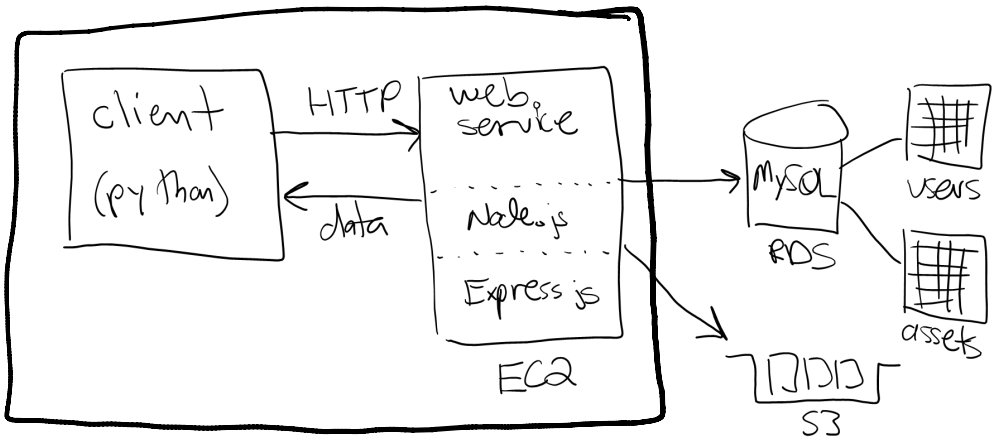
```
[client]
webservice=https://project-02-…-fall-2023.repl.co
```

There are a total of 6 commands to implement, very similar to project 01. Those commands are the following:

```
** Welcome to PhotoApp v2 **

What config file to use for this session?
Press ENTER to use default (photoapp-config),
otherwise enter name of config file>


>> Enter a command:
   0 => end
   1 => stats
   2 => users
   3 => assets
   4 => download
   5 => download and display
   6 => bucket contents
```

1. Get stats
2. Get list of users
3. Get list of assets
4. Download
5. Download and display
6. Get list of bucket contents, displaying one page at a time

Commands 1 and 2 are implemented for you, and follow the approach we discussed in Lecture 05. Your job is to implement the remaining 4 commands via **web service calls** --- your Python-based client cannot communicate with AWS S3 or RDS directly. In other words, your python client cannot execute SQL, and cannot access the S3 bucket directly using boto3.



<< continued on next page >>

Python-based client

Command 1: Get stats

Command 1 calls the web service API function **/stats** and displays the response. This command is already implemented in the provided "main.py" file.

```
1
bucket status: success
# of users: 7
# of assets: 19
```

Command 2: List users

Command 2 calls the web service API function **/users** and displays information about each user returned in the response. This command is already implemented in the provided "main.py" file.

```
2
80001
  pooja.sarkar@company.com
  sarkar , pooja
  6b0be043-1265-4c80-9719-fd8dbcda8fd4
80002
  e_ricci@email.com
  ricci , emanuele
  ab099de4-ea33-4237-8c78-5584dc591231
80003
  li_chen@domain.com
  chen , li
  52592157-9216-4d4c-882f-3f3013864930
80012
  ecf16af39d6120b463f9@nu.edu
  FamilyName , GivenName
  4fdd4244-7ae9-4fc8-a0eb-e6eebf28901e
```

Take a minute to review the implementation, in particular notice we're converting the response data into Python objects using the jsons module:

```
class User:
  userid: int  # these must match columns in DB
  email: str
  lastname: str
  firstname: str
  bucketfolder: str

.
.
.

users = []
for row in body["data"]:
  user = jsons.load(row, User)
  users.append(user)
```

This is commonly done in client-side programming involving databases, and is known as ORM --- "Object-Relational Mapping". This allows the client-side code to be written in a more natural object-oriented style:

```
for user in users:
  print(user.userid)
  print(" ", user.email)
  print(" ", user.lastname, ",", user.firstname)
  print(" ", user.bucketfolder)
```

Command 3: List assets

```
3
1001
  80001
  A3-mac-2016.JPG
  6b0be043-1265-4c80-9719-fd8dbcda8fd4/af986381-55ac-4bf2-85b3-ff4a29047226.jpg
1002
  80001
  A3-verve-Megan-on-bow.jpg
  6b0be043-1265-4c80-9719-fd8dbcda8fd4/62ec70d5-ba0c-4822-9017-3864b1d1fb47.jpg
1003
  80001
```

Command 3 calls the web service API function **/assets** and displays information about each asset returned in the response. Your implementation must use try-except to handle errors that might occur, and handle status codes != 200 in a similar manner to the provided commands. You are encouraged to use a similar ORM approach as used in command 2.

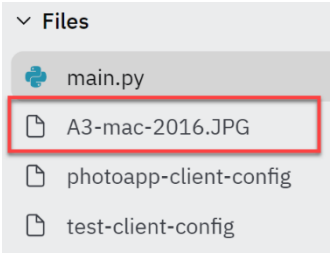
Command 4: Download

Command 4 inputs an asset id from the user, and then passes this id to the API function **/download**. Based on the response, the output is either "No such asset…"

```
4
Enter asset id>
101
No such asset...
```

or the returned base64-encoded string is decoded and written to the file system as a binary file. The name of the file should be the **asset name** returned in the response, in this case "A3-mac-2016.JPG":

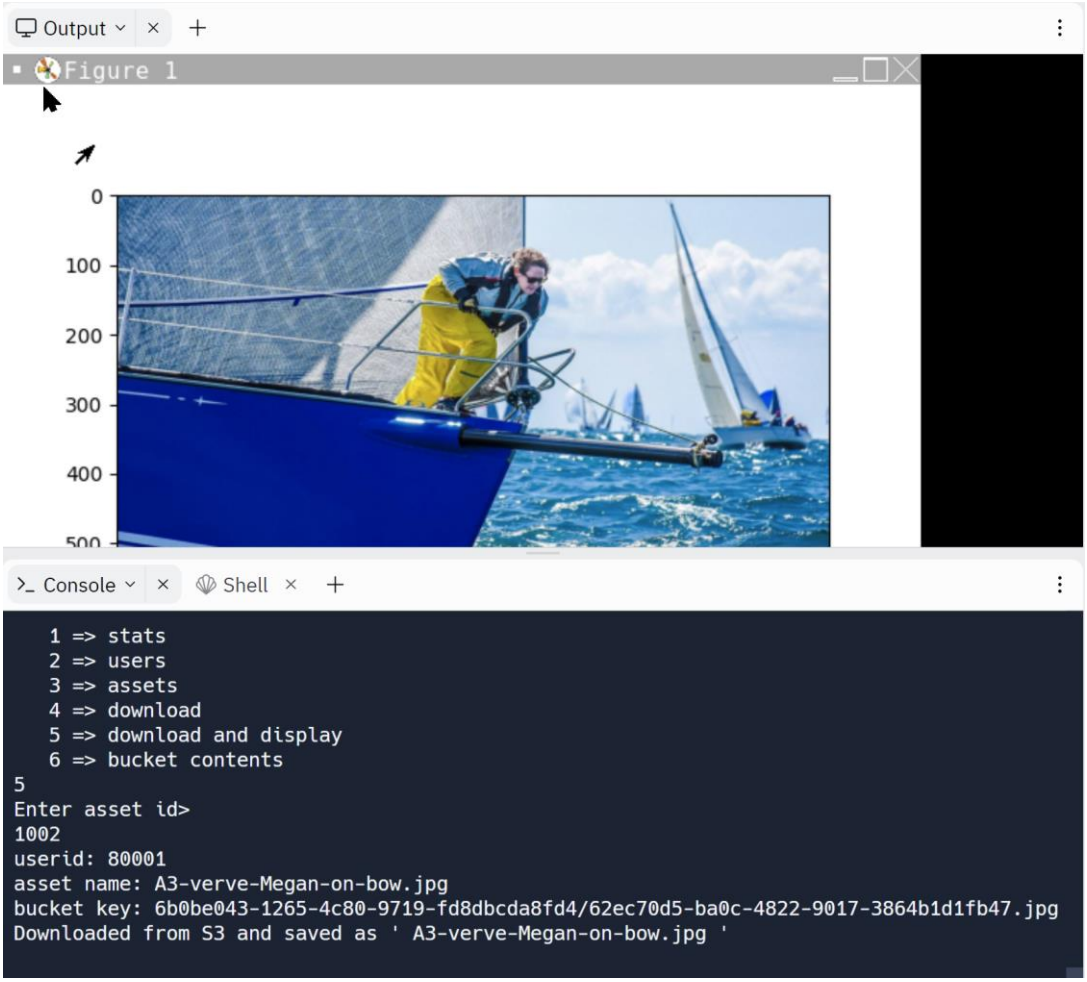
```
4
Enter asset id>
1001
userid: 80001
asset name: A3-mac-2016.JPG
bucket key: 6b0be043-1265-4c80-9719-fd8dbcda8fd4/af986381-55ac-4bf2-85b3-ff4a29047226.jpg
Downloaded from S3 and saved as ' A3-mac-2016.JPG '
```



Your implementation must use try-except to handle errors that might occur, and handle status codes != 200 in a similar manner to the provided commands. To decode the data use the base64.b64decode( ) function. And when you write the decoded bytes to the file, it must be a binary file, not a text file:

```
outfile = open(assetname, "wb")  # wb => write binary
```

Command 5: Download and display



Same as command 4, except display the image like we did in Project 01 using **matplotlib**.

<< continued on next page >>

Command 6: List bucket contents one page at a time

Command 6 calls the web service API function **/bucket** and displays information about each bucket asset returned in the response. Recall that the /bucket function returns information in pages of size 12, so the client needs to call the web service each time the user asks for a page:

```
6
52592157-9216-4d4c-882f-3f3013864930/
  2023-03-25T13:37:47.000Z
  0
52592157-9216-4d4c-882f-3f3013864930/195e06c8-6005-4006-97f2-1c7c19b3414b.jpg
  2023-03-25T13:58:12.000Z
  43534
52592157-9216-4d4c-882f-3f3013864930/564055dc-f109-4df8-bdf1-27a629d4a0c6.jpg
  2023-03-25T13:57:13.000Z
  5438
52592157-9216-4d4c-882f-3f3013864930/a68cab2e-7d23-4af3-bbb1-f067befc6712.jpg
  2023-03-25T13:56:36.000Z
  60222
```

.
.
.

```
6b0be043-1265-4c80-9719-fd8dbcda8fd4/701521d6-c273-43dc-ad7f-bcc1e4d76bcd.jpg
  2023-04-11T22:26:52.000Z
  44023
6b0be043-1265-4c80-9719-fd8dbcda8fd4/7b4c09c3-aa07-45f3-a5dd-454e486c6a27.jpg
  2023-04-07T20:11:18.000Z
  44023
another page? [y/n]
```

If the user inputs **y**, then the client calls the web service to request the next page, displays these to the user, and prompts again. Do not assume the web service will return exactly 12 assets --- it could be any number from 0..12. If the user inputs anything else, break out of your paging loop and the command ends.

```
another page? [y/n]
n

>> Enter a command:
   0 => end
   1 => stats
   2 => users
   3 => assets
   4 => download
   5 => download and display
   6 => bucket contents
```

Special case: if the web service returns 0 assets, do not prompt the user for another page --- automatically break out of your paging loop. Your implementation must use try-except to handle errors that might occur, and handle **status** codes != 200 in a similar manner to the provided commands.

Electronic Submission --- Part 01

Programs will be collected and auto-graded using Gradescope. A few days before the due date, two Gradescope assignments will open named

Project 02 (server part01) **Project 02 (client part01)**

Make sure your database is running in RDS, and then for "Project 02 (server part01)" submit all your server files:

app.js
api_asset.js
api_bucket.js
api_download.js
api_stats.js
api_users.js
aws.js
config.js
database.js
photoapp-config.ini

You have unlimited submissions, and the goal is a score of 100/100.

For the client, it doesn't matter if your database is running because we plan to test your client against our web service. For "Project 02 (client part01)", submit just one file:

main.py

You have unlimited submissions, and the goal is a score of 100/100.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your "Submission History". This must be done before the due date.

This is a 300-level CS class, so we expect you'll comment your code, write functions, use error handling, etc. At the very least, be sure to put your name at the top of the "app.js" and "main.py" files in the header comments. We reserve the right to manually review your submissions for the required elements (e.g. the client-side Python code should be using try-except and check response status codes). You may lose points if these required elements are not present.

Part 02?

This is NOT the end of Project 02. Part 02 will involve additions to the web service and client, as well as details on how to deploy your web service to AWS EC2 using *Elastic Beanstalk*. The due date for Part 02 is one week after the due date for Part 01, which means Part 02 will be due Friday October 27th @ 11:59pm.

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's

eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity website. With regards to CS 310, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.