

## Project #02 --- Part 02 (v1.3)

**Assignment:** Multi-tier PhotoApp with AWS EC2, S3 and RDS

**Submission:** via Gradescope (unlimited submissions)

**Policy:** individual work only, late work is accepted

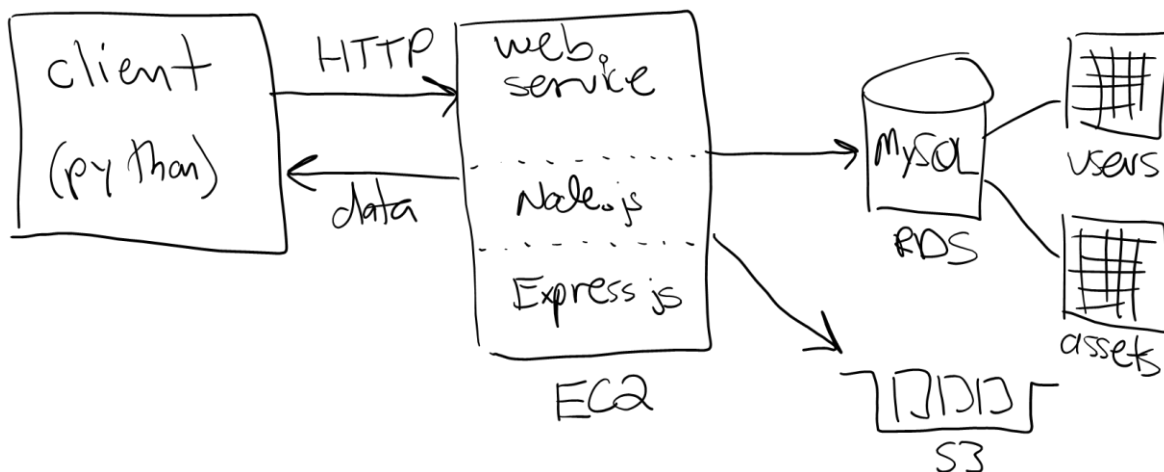
**Complete By:** Friday May 12 @ 11:59pm CST

Late submissions: you can submit up to 3 days late (Monday May 15), no penalty using late days.  
NO submissions accepted after Monday May 15.

**Pre-requisites:** Lecture 08 (Tues April 25). See Canvas for links to recording / PPT

### Overview

Here in project 02 we are building out the web service for our PhotoApp, and rewriting the client to interface with the web service API:



In part 02 we're going to complete the web service by adding two more functions to the API: `/user` and `/image/:userid`. Then we'll use AWS *Elastic Beanstalk* to help us provision an instance of EC2 to run our web service and expose our API to the outside world.

## Before we start...

Here in part 02 there are three main steps to perform:

1. Implement the API function **/user**, which uses the **PUT** verb to either insert a new user into the database, or if the user already exists (based on the email address) then the user's lastname, firstname, and bucketfolder in the database are updated.
2. Implement the API function **/image/:userid**, which uses the **POST** verb to upload an image to S3, and then store information about this asset in the database.
3. Use AWS **Elastic Beanstalk** to deploy your web service on an instance of EC2.

Gradescope will be used to collect your web service files, and your EC2 endpoint. We are not collecting your Python client, but you'll need to continue development for testing purposes.

### (1) /user

The **/user** function inserts a new user, or updates an existing user, in the database. The client will make a **PUT** request, passing the following data in the body of the request using JSON:

```
data = {
  "email":      "...",
  "lastname":   "...",
  "firstname":  "...",
  "bucketfolder": "..."
```

If the given email is not found in the **users** table, a new user is inserted and the web service returns the following response:

```
{
  "message": "inserted",
  "userid":  userid
}
```

Note that when you execute an SQL insert query, the MySQL library returns the auto-generated userid in **result.insertId** (capital I and lowercase d). If you want to double-check that the insert executed successfully, confirm that **result.affectedRows == 1**.

If the given email is found in the users table, the user's row in the database is updated with the provided data, and the web service returns the following response:

```
{
  "message": "updated",
  "userid":  userid
}
```

No changes are made to S3, even though the user's bucketfolder may have changed in the database. If an

error occurs, return the following response with a status code of 400:

```
{
  "message": "some sort of error message",
  "userid": -1
}
```

To get started, download the JS file “api\_user.js” from [dropbox](#). This file exports the function **put\_user**:

```
exports.put_user = async (req, res) => {
  console.log("call to /user...");

  try {
    .
    .
    .
  }
  catch (err) {
    res.status(400).json({
      "message": err.message,
      "userid": -1
    });
  }
}
```

You'll need to update your main “app.js” file to call this function when a PUT request is made for /user. First, add the following app.use( ) call to enable JSON deserialization of incoming JSON data (and support for large image files):

```
29 const express = require('express');
30 const app = express();
31 const config = require('./config.js');
32
33 const dbConnection = require('./database.js')
34 const { HeadBucketCommand, ListObjectsV2Command } = require('@aws-sdk/client-s3');
35 const { s3, s3_bucket_name, s3_region_name } = require('./aws.js');
36
37 app.use(express.json({ strict: false, limit: "50mb" }));
38
39 var startTime;
40 |
41 ~ app.listen(config.service_port, () => {
42   startTime = Date.now();
43   console.log('web service running...');
44   //
45   // Configure AWS to use our config file:
46   //
47   process.env.AWS_SHARED_CREDENTIALS_FILE = config.photoapp_config;
48 });
```



Now load the “api\_user.js” file (step 1) and associate a PUT request for /user to call put\_user (step 2):

```
61 //
62 // service functions:
63 //
64 var stats = require('./api_stats.js');
65 var users = require('./api_users.js');
66 var assets = require('./api_assets.js');
67 var bucket = require('./api_bucket.js');
68 var download = require('./api_download.js');
69
70 var user = require('./api_user.js'); 1
71
72 app.get('/stats', stats.get_stats); //app.get('/stats', (req, res) => {...});
73 app.get('/users', users.get_users); //app.get('/users', (req, res) => {...});
74 app.get('/assets', assets.get_assets); //app.get('/assets', (req, res) => {...});
75 app.get('/bucket', bucket.get_bucket); //app.get('/bucket?startafter=bucketkey', (req,
76 app.get('/download/:assetid', download.get_download); //app.get('/download/:assetid', (
77
78 app.put('/user', user.put_user); // app.put('/user', (req, res) => {...}); 2
79
```

The web service is now configured and runnable. This would be a good time to setup a testing infrastructure, and confirm that the web service returns an error when /user is called:

```
{
  "message": "TODO: /user",
  "userid": -1
}
```

How to test? You have at least two options. Option #1 is to extend your Python-based client with another command to update/insert a user. See [Lecture 08](#) (slide 13) for an example of how to code the client. Option #2 is to use a tool such as <https://postman.com> to generate a PUT request and supply test data; see lecture 08 for an example of this as well. [ *Note that we are **\*not\*** collecting the Python client here in part 02, so you are free to implement client-side testing however you want.* ]

**SUGGESTION:** when working with a new language and framework, it’s generally a good idea to write just a few lines of code, add some print debugging with `console.log( )`, run and test. Then write a few more lines, more `console.log( )`, run, test. Repeat. Your function needs to respond, so at the bottom of your try block have this:

```
console.log("/user done, sending response...");

res.json({
  "message": "so far so good",
  "userid": 123
});
```

Slowly evolve the function step by step until it’s working.

## (2) /image/:userid

The **/image/:userid** function uploads an image to S3 and updates the database accordingly. The client will make a POST request, passing the user id as part the path and the following data in the body of the request using JSON:

```
data = {
  "assetname": "...",
  "data":      "...",
}
```

The **assetname** element is the local filename, and **data** element is the image as a base64-encoded string. If the user id does not exist in the database, the web service returns normally (status code 200) with the following response:

```
{
  "message": "no such user...",
  "assetid": -1
}
```

If the user id exists, the image is uploaded to S3 and stored in the user's folder with a unique key; generate the key using UUID v4 as we have in the past. Also insert a new row into the **assets** table of the database, and respond with the auto-generated asset id as follows:

```
{
  "message": "success",
  "assetid": assetid
}
```

If an error occurs, return the following response with a status code of 400:

```
{
  "message": "some sort of error message",
  "assetid": -1
}
```

To get started, download the JS file "api\_image.js" from [dropbox](#). This file exports the function **post\_image**:

```
const uuid = require('uuid');

exports.post_image = async (req, res) => {
  console.log("call to /image...");

  try {
    .
    .
    .
  }
  catch (err) {
```

```

    res.status(400).json({
      "message": err.message,
      "assetid": -1
    });
  }
}

```

You'll need to update your main "app.js" file to call this function when a POST request is made for /image. Load "api\_image.js" file (step 1) and associate a POST request for /image/:userid to call post\_image (step 2):

```

61 //
62 // service functions:
63 //
64 var stats = require('./api_stats.js');
65 var users = require('./api_users.js');
66 var assets = require('./api_assets.js');
67 var bucket = require('./api_bucket.js');
68 var download = require('./api_download.js');
69 var user = require('./api_user.js');
70
71 var image = require('./api_image.js');
72
73 app.get('/stats', stats.get_stats); //app.get('/stats', (req, res) => {...});
74 app.get('/users', users.get_users); //app.get('/users', (req, res) => {...});
75 app.get('/assets', assets.get_assets); //app.get('/assets', (req, res) => {...});
76 app.get('/bucket', bucket.get_bucket); //app.get('/bucket?startafter=bucketkey', (req, res) => {...});
77 app.get('/download/:assetid', download.get_download); //app.get('/download/:assetid', (req, res) => {...});
78 app.put('/user', user.put_user); // app.put('/user', (req, res) => {...});
79
80 app.post('/image/:userid', image.post_image); // app.post('/image/:userid', (req, res) => {...});
81

```

The web service is now configured and runnable. This would be a good time to setup a testing infrastructure, and confirm that the web service returns an error when for example **/image/123** is called:

```

{
  "message": "TODO: /image",
  "assetid": -1
}

```

In this case I would recommend testing via your Python-based client, since you'll want to upload the file, and then download and display to confirm it was uploaded properly. You can also use the "users" and "assets" commands to ensure the database is being updated correctly.

**Some programming hints...** On the client, you need to pass the image as a base64-encoded string. This requires the following steps in Python:

1. *Open the file for binary read using `open(filename, 'rb')`, read the contents, and close the file. The result is an array of bytes we'll call `B`.*
2. *Encode as base64: `E = base64.b64encode(B)`*

3. Interestingly, *E* is an array of type *byte*, which JSON doesn't like. So convert *E* to a string *S* using *S = E.decode()*

*S* is your image as a base64-encoded string. Pass *S* as your data element to the web service. On the server, you'll receive the base64-encoded string and will need to decode so you can send the raw bytes to S3. Use the **Buffer** functionality in node.js to decode:

```
var S = req.body.data;
var bytes = Buffer.from(S, 'base64');
```

You'll send bytes to S3 using the **PutObjectCommand**; see this [reference](#) for how to call S3 using this command (scroll down and expand the example "*Upload an object to a bucket*"). Finally, remember to use UUID to generate a unique bucket key for the image:

```
// generate a unique name for the asset:
var name = uuid.v4();
```

Don't forget to prefix the key with the user's bucket folder and *"/*; you may assume the file extension is *".jpg"*.

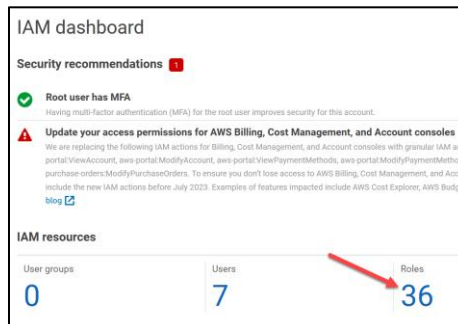
**SAME SUGGESTION**: build the web service function one small step at a time. Use `console.log()` to confirm your steps.

## (3) AWS IAM Roles

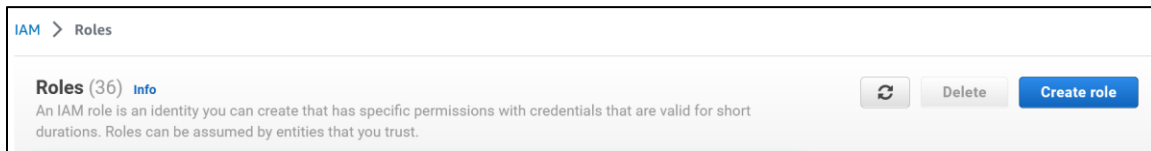
Congratulations, you have built a non-trivial web service using JavaScript, Node.js, and Express js! Well done. The programming is over, the last two steps are to configure AWS to run your web service using **EC2**, Amazon's *Elastic Compute* service. Since we are using a well-known framework, AWS makes it even easier by providing a service on top of EC2 called *Elastic Beanstalk* (EB) that makes setup even easier.

The first step is to setup two security roles in AWS IAM (identity access management), which are needed to control EB and EC2. Steps:

1. Login to AWS and open the Management Console. Search for "IAM".
2. You'll see 1 or more roles --- click on the number (36 in our case):



3. Click “Create role” to create a new role:



4. On the next screen, select “AWS service” and “EC2” as common use case:

A screenshot of the 'Select trusted entity' screen in the AWS IAM console. The title is 'Select trusted entity' with an 'Info' link. Under the 'Trusted entity type' section, there are five radio button options: 'AWS service' (selected, marked with a red circle containing '1'), 'AWS account', 'Web identity', 'SAML 2.0 federation', and 'Custom trust policy'. Each option has a brief description. Below this is the 'Use case' section with the instruction 'Allow an AWS service like EC2, Lambda, or others to perform actions in this account.' Under 'Common use cases', there are two radio button options: 'EC2' (selected, marked with a red circle containing '2') and 'Lambda'. Each option has a brief description.



5. The next screen is for adding additional permissions --- narrow down the search by entering "AWSElastic" in the search field and pressing ENTER. Then select the two roles shown and click "Next":

## Add permissions [Info](#)

### Permissions policies (Selected 2/848) [Info](#)

Choose one or more policies to attach to your new role.

Filter policies by property or policy name and press enter.

AWSElastic

X

Clear filters

-

Policy name

<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkWebTier</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkWorkerTier</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkMulticontainerDocker</div>
<input checked="" type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkEnhancedHealth</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkCustomPlatformforEC2Role</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkRoleWorkerTier</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkRoleSNS</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkRoleRDS</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkRoleECS</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkRoleCore</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkRoleCWL</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkReadOnly</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AdministratorAccess-AWSElasticBeanstalk</div>
<input checked="" type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticBeanstalkManagedUpdatesCustomerRolePolicy</div>
<input type="checkbox"/>	<div><div></div><div></div></div> <div>AWSElasticDisasterRecoveryRecoveryInstancePolicy</div>

Cancel

Previous

Next

6. On the next screen name the role “aws-elasticbeanstalk-service-role” and click “Create Role”:

## Name, review, and create

### Role details

Role name

Enter a meaningful name to identify this role.

Maximum 64 characters. Use alphanumeric and '+=,.\_@-' characters.

[Cancel](#)[Previous](#)[Create role](#)

7. After a few minutes the role should be created --- if anything goes wrong, you can delete and start over.
8. We need to repeat this process and create another role named “aws-elasticbeanstalk-ec2-role”. Repeat steps 2 – 6, except when you “Add permissions”, select these 3 policies instead:

IAM > Roles > aws-elasticbeanstalk-ec2-role

aws-elasticbeanstalk-ec2-role

### Summary

Creation date

April 05, 2023, 19:54 (UTC-05:00)

Last activity

✓ 26 minutes ago

PermissionsTrust relationshipsTagsAccess AdvisorRevoke

### Permissions policies (3) [Info](#)

You can attach up to 10 managed policies.

Filter policies by property or policy name and press enter.

	Policy name	Type
<input type="checkbox"/>	<input type="checkbox"/> AWSElasticBeanstalkWebTier	AWS manag...
<input type="checkbox"/>	<input type="checkbox"/> AWSElasticBeanstalkWorkerTier	AWS manag...
<input type="checkbox"/>	<input type="checkbox"/> AWSElasticBeanstalkMulticontainerDocker	AWS manag...

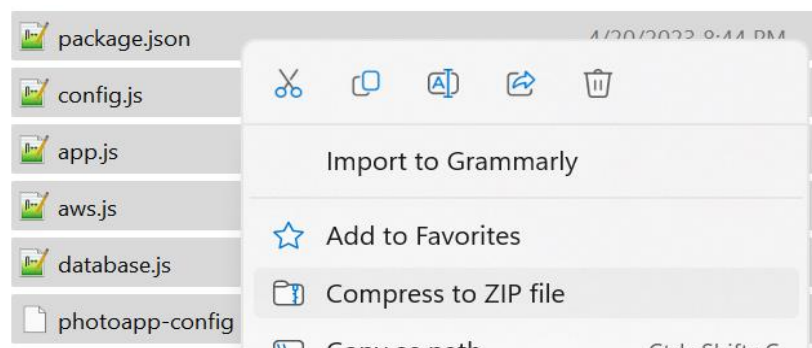
## (4) AWS Elastic Beanstalk

Now that the roles are created, here are the steps to get your web service up and running on EC2 using EB. When you're done, you'll have a public endpoint that allows your API to be called by anyone with an internet connection. Test by pointing your Python-based client at this endpoint.

1. Download your web service, in particular the following 12 files (if you are working on replit, you can download the files individually, or you can download as .zip and then extract the files). Either way, you need the following 12 files in a local folder on your laptop named "web-service":

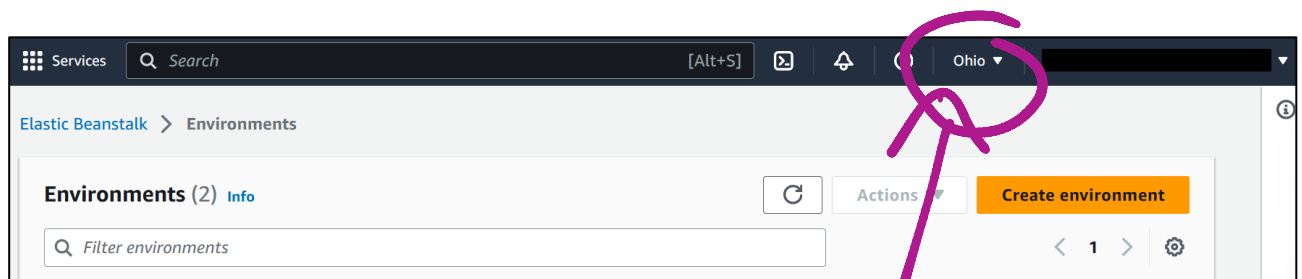
app.js	api_image.js	aws.js
api_asset.js	api_stats.js	config.js
api_bucket.js	api_user.js	database.js
api_download.js	api_users.js	photoapp-config

2. Download the file "package.json" from [dropbox](#) and save this in your "web-service" folder. This is a configuration file for Elastic Beanstalk for configuring Node.js and Express.js.
3. Select all the files in your "web-service" folder --- 13 files in total --- and create a compressed / archive file. The resulting file should have a .ZIP extension, and you should compress the files directly into the .ZIP:



For example, do *\*not\** compress the web-service folder --- the .ZIP needs to contain just the files, no folders. Normally you select all the files, right-click, and select compress. [ *On windows, it might also be select all the files, right-click, select "Send to", and then select "Compressed (zipped) folder" ]*

4. Login to AWS and open the Management Console. Search for "Elastic Beanstalk".
5. Use the drop-down to select your region --- it should be the same region that contains your database and bucket. When in doubt, select "Ohio" (us-east-2). You should see something like this:



6. Click “Create environment”. Select a “Web server environment” and press Select if you see a button (or maybe just scroll down, there may be differences):

Elastic Beanstalk > Create environment

### Select environment tier

Amazon Elastic Beanstalk has two types of environment tiers to support different types of web applications. Web servers are standard applications that listen for and then process HTTP requests, typically over port 80. Workers are specialized applications that have a background processing task that listens for messages on an Amazon SQS queue. Worker applications post those messages to your application by using HTTP.

☒ **Web server environment**  
Run a website, web application, or web API that serves HTTP requests.  
[Learn more](#)

☐ **Worker environment**  
Run a worker application that processes long-running workloads on demand or performs tasks on a schedule.  
[Learn more](#)

Cancel **Select**

7. Enter an application name. Like your bucket, this should be unique so use your name and some combination of “photoapp” and “web-service”:

### Application information [Info](#)

**Application name**

photoapp-nu-web-service

Maximum length of 100 characters.

8. Now configure the application as a managed platform running Node.js:

### Platform [Info](#)

**Platform type** 1

☒ **Managed platform**  
Platforms published and maintained by Amazon Elastic Beanstalk. [Learn more](#)

☐ **Custom platform**  
Platforms created and owned by you. This option is unavailable if you have no platforms.

**Platform** 2

Node.js

**Platform branch** 3

Node.js 18 running on 64bit Amazon Linux 2

**Platform version** 4

5.8.1 (Recommended)

9. Next, upload the .ZIP file containing your web service files:

**Application code** [Info](#)

☐ Sample application

☒ Existing version  
Application versions that you have uploaded.

☒ Upload your code  
Upload a source bundle from your computer or copy one from Amazon S3.

**Version label**  
Unique name for this version of your application code.

project02-server-v1

Source code origin. Maximum size 2 GB

☒ Local file

Upload application

File name: **web-service.zip**  
File must be less than 2GB max file size

☐ Public S3 URL

Fun fact: the .zip file is stored in S3.

10. Under “Configuration presets”, be sure to select “single instance” which is free tier eligible. My screen shows a Next, in which case click and continue:

**Presets** [Info](#)

Start from a preset that matches your use case or choose custom configuration to unset recommended values and use the service's default values.

**Configuration presets**

☒ Single instance (free tier eligible)

☐ Single instance (using spot instance)

☐ High availability

☐ High availability (using spot and on-demand instances)

☐ Custom configuration

11. Next is “Configure service access”. In steps 2 and 4 you’ll select the roles you created earlier --- skip step 3 (leave EC2 key pair blank). Click Next when you’re ready:

## Configure service access [Info](#)

### Service access

IAM roles, assumed by Elastic Beanstalk as a service role, and EC2 instance profiles allow Elastic Beanstalk to create and manage your environment. Both the IAM role and instance profile must be attached to IAM managed policies that contain the required permissions. [Learn more](#)

Service role

☐ Create and use new service role

☒ Use an existing service role

Existing service roles

Choose an existing IAM role for Elastic Beanstalk to assume as a service role. The existing IAM role must have the required IAM managed policies.

aws-elasticbeanstalk-service-role

EC2 key pair

Select an EC2 key pair to securely log in to your EC2 instances. [Learn more](#)

Choose a key pair

EC2 instance profile

Choose an IAM instance profile with managed policies that allow your EC2 instances to perform required operations.

aws-elasticbeanstalk-ec2-role

View permission details

Cancel

Skip to review

Previous

Next

<< continued on next page >>

Page 14 of 20

12. Setup networking, database, and tags... Under VPC, you should have one option to select --- this is the VPC that contains your MySQL database. Select this option because we want EC2 to run on the same VPC as the database. Note that your VPC will be named differently. If you have multiple choices, you should probably stop, look at your database under RDS, scroll to the right, and see which VPC is being used by your database --- you want to select that VPC here.

For the other options, you want to activate “Public IP address” (step 2) so your web service is accessible by the outside world. The “instance subnets” should be auto-populated based on the selected VPC --- select them all (step 3). Finally, do *\*not\** enable a database, we already have one!

### Set up networking, database, and tags - *optional* [Info](#)

#### Virtual Private Cloud (VPC)

VPC

Launch your environment in a custom VPC instead of the default VPC. You can create a VPC and subnets in the VPC management console. [Learn more](#)

vpc-0ccf3cdb378341daf | (172.30.0.0/16)

1

[Create custom VPC](#)

#### Instance settings

Choose a subnet in each AZ for the instances that run your application. To avoid exposing your instances to the Internet, run your instances in private subnets and load balancer in public subnets. To run your load balancer and instances in the same public subnets, assign public IP addresses to the instances. [Learn more](#)

##### Public IP address

Assign a public IP address to the Amazon EC2 instances in your environment.

☒ Activated

2

##### Instance subnets

Filter instance subnets

<input checked="" type="checkbox"/>	Availability Zone	Subnet	CIDR	Name
<input checked="" type="checkbox"/>	us-east-2b	subnet-095183698...	172.30.1.0/24	
<input checked="" type="checkbox"/>	us-east-2a	subnet-0ce250ecf2...	172.30.0.0/24	
<input checked="" type="checkbox"/>	us-east-2c	subnet-0e006a5e8...	172.30.2.0/24	

3

##### Database [Info](#)

Integrate an RDS SQL database with your environment. [Learn more](#)

☐ Enable database

4

**do not enable!**

Cancel

Skip to review

Previous

Next

13. Configure instance traffic and scaling... Leave alone and click Next...

### Configure instance traffic and scaling - *optional* [Info](#)

▼ Instances [Info](#)

Configure the Amazon EC2 instances that run your application.

Cancel

Skip to review

Previous

Next

14. Configure updates, monitoring, and logging... Switch to “basic” health reporting and “de-activate” platform updates:

### Configure updates, monitoring, and logging - *optional* [Info](#)

▼ Monitoring [Info](#)

#### Health reporting

Enhanced health reporting provides free real-time application and operating system monitoring of the instances and other resources in your environment. The **EnvironmentHealth** custom metric is provided free with enhanced health reporting. Additional charges apply for each custom metric. For more information, see [Amazon CloudWatch Pricing](#).

System

☒ Basic 1

☐ Enhanced

#### Health event streaming to CloudWatch Logs

Configure Elastic Beanstalk to stream environment health events to CloudWatch Logs. You can set the retention up to a maximum of ten years and configure Elastic Beanstalk to delete the logs when you terminate your environment.

Log streaming

☐ Activated (standard CloudWatch charges apply.)

Retention

7 ▼

Lifecycle

Keep logs after terminating environment ▼

▼ Managed platform updates [Info](#)

Activate managed platform updates to apply platform updates automatically during a weekly maintenance window that you choose. Your application stays available during the update process.

Managed updates

☐ Activated 2

Cancel

Skip to review

Previous

Next

Page 16 of 20



15. You should see the final review page. If everything looks good, scroll down and click Submit!

## Review [Info](#)

Step 1: Configure environment Edit

### Environment information

Environment tier	Application name
Web server environment	photoapp-nu-web-service
Environment name	Application code
Photoapp-nu-web-service-env	web-service.zip
Platform	
arn:aws:elasticbeanstalk:us-east-2::platform/Node.js 18 running on 64bit Amazon Linux 2/5.8.1	

Step 2: Configure service access Edit

### Service access [Info](#)

Configure the service role and EC2 instance profile that Elastic Beanstalk uses to manage your environment. Choose an EC2 key pair to securely log in to your EC2 instances.

Service role	EC2 instance profile
arn:aws:iam::444800541605:role/aws-elasticbeanstalk-service-role	aws-elasticbeanstalk-ec2-role

Step 3: Set up networking, database, and tags Edit

### Networking, database, and tags [Info](#)

Configure VPC settings, and subnets for your environment's EC2 instances and load balancer. Set up an Amazon RDS database that's integrated with your environment.

#### Network

Public IP address

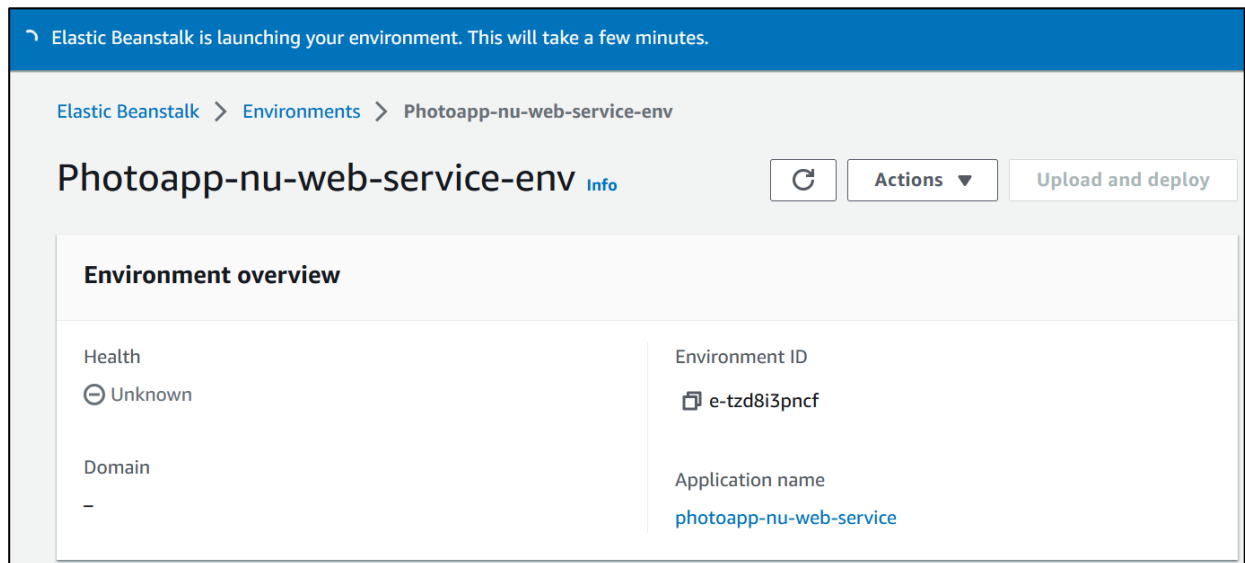
true

Cancel

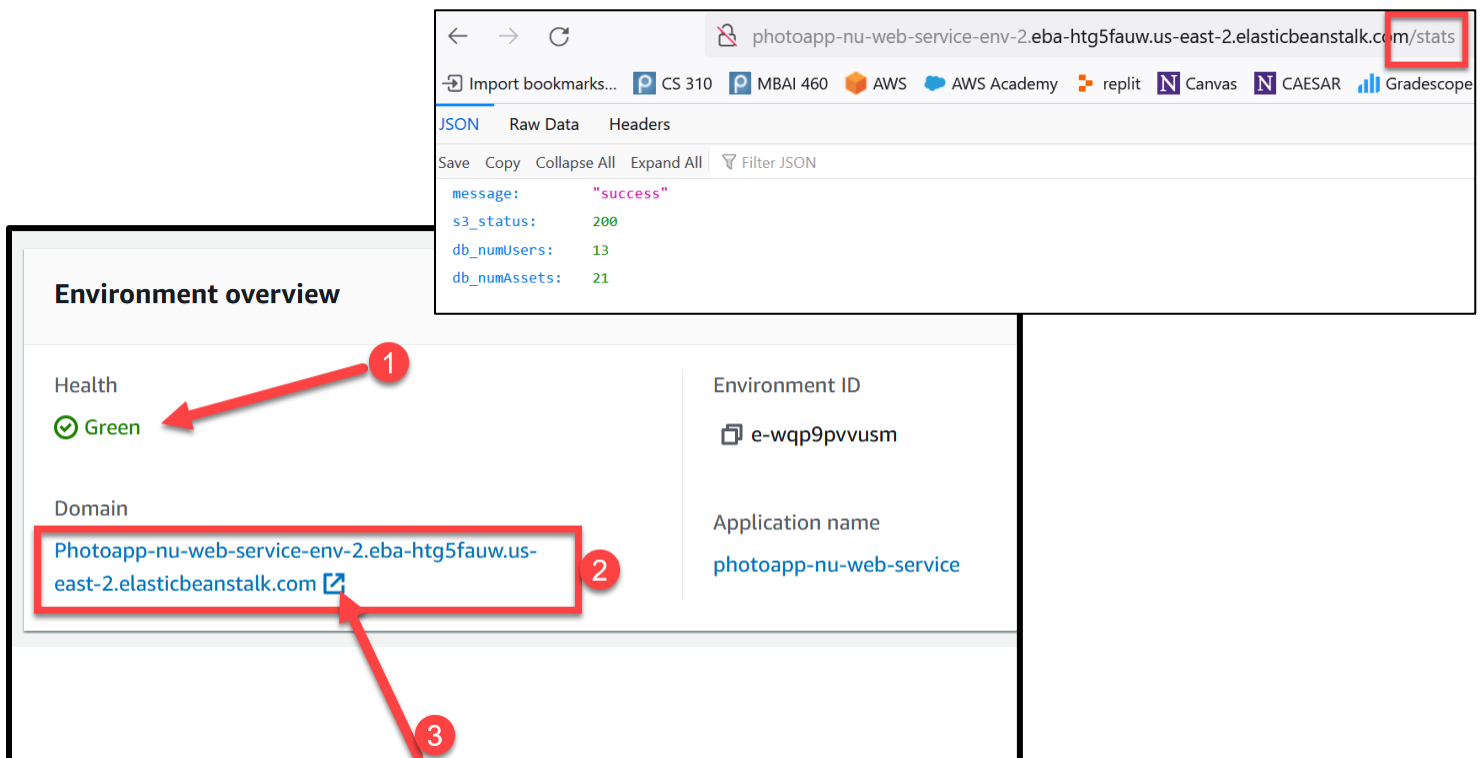
Previous

Submit

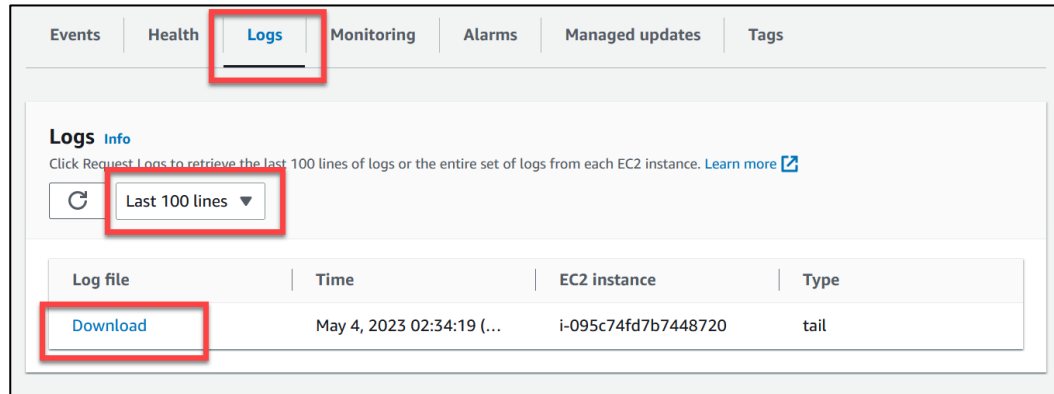
16. AWS will now configure EC2, load your code, and startup the web service. This will take a few minutes...



17. If all is well, you should get a "Green" health check (step 1), and your domain endpoint should be displayed (step 2). Click the little "pop-out" button (step 3) to open a browser window so you can test your GET functions like /stats and /users:



18. At this point, you'll want to test all the API functions using your Python-based client. Create another client config file, and paste your AWS EC2 endpoint into that config file. Run your client and test...
19. If your EC2 setup failed, the best course of action is to start over and see if you missed any steps... Another option is to look at the logs, which contain the output from EC2 but also the output from your `console.log()` statements:



20. Want to upload a new version of your code? View “Elastic Beanstalk” in the management console, select your environment, and you’ll see a button on the far-right for “Update and deploy”:



Follow the steps to upload a new version of your code...

21. How to stop the instance to save money? Turns out there’s no way to stop the instance that’s running --- if we stop it, AWS thinks it has crashed and will start a new instance within a few minutes (which is what we would normally want to happen). To “stop” and save money, I would click on the application (in the Application column) and terminate it. This will delete everything, but it’s easy enough to recreate. [ NOTE: there is a way to pause an instance, see this [post](#). ]
22. Congratulations, well done!

## Electronic Submission --- Part 02

Step 1 is to create a client-side config file named “ec2-client-config.txt” and paste your EC2 endpoint into this file for submission. The endpoint is the URL for your web service, and the config file should look like this:

```
[client]
webservice=http://photoapp-YOUR-NAME-web-service-env-UNIQUE-ID.elasticbeanstalk.com
```

Step 2 is to submit your files to Gradescope; note that we are only collecting the server-side code, and not the client. A few days before the due date, look for “Project 02 (server part 02)”. Submit the same files you deployed to AWS + “ec2-client-config.txt”. This should be a total of 13 files:

app.js	api_stats.js	database.js
api_asset.js	api_user.js	ec2-client-config.txt
api_bucket.js	api_users.js	photoapp-config
api_download.js	aws.js	
api_image.js	config.js	

If you want, add your “ec2-client-config.txt” file to the .ZIP file you uploaded to AWS, and submit the .ZIP to Gradescope. You have unlimited submissions, and the goal is a score of 100/100.