



Building A 101 Web App

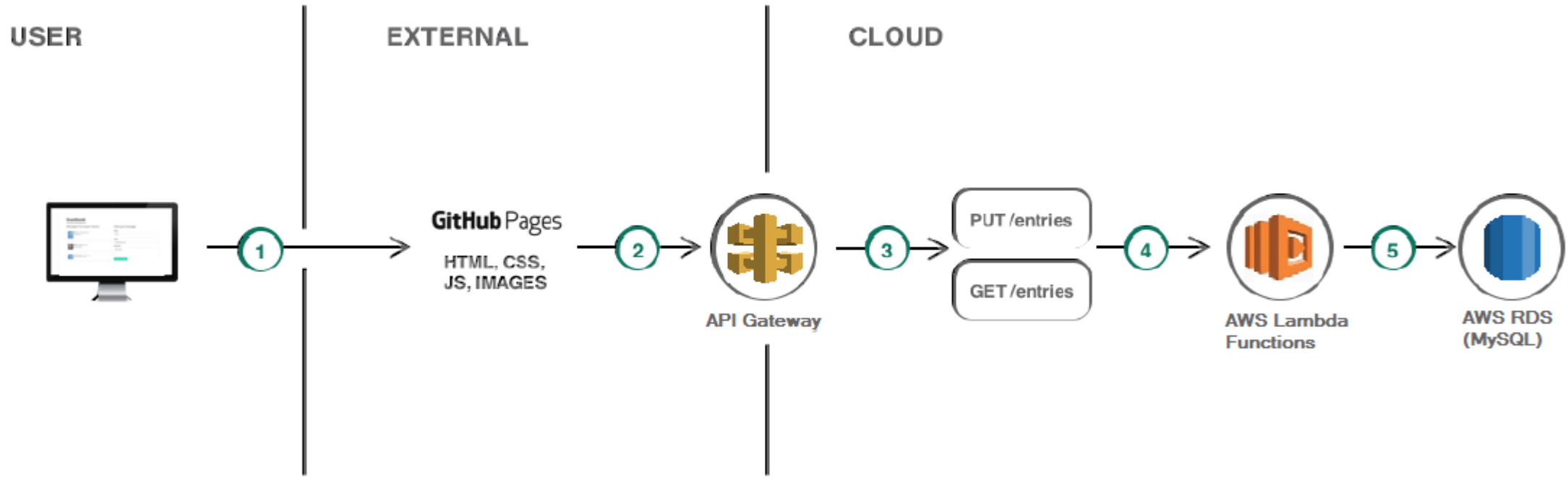
THE SERVERLESS WAY

Runcy Oommen
<https://runcy.me>

What is Serverless?

“Serverless computing is a cloud computing execution model in which the cloud provider dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application.” (via Wikipedia)

Sample scenario: Serverless Web Application



What is “Serverless”?

- A way to host and run your code without having to *think* about servers.
- The ability to expose a single function, run in on a shared server, and pay only for the milliseconds it takes to execute.
- A command line tool that simplifies building and deploying serverless functions (serverless framework).



github.com/serverless/serverless

A Few Good Resources

- **AWS Info page on serverless**

<https://aws.amazon.com/serverless/>

- **Serverless Architectures**

<https://martinfowler.com/articles/serverless.html>

- **Lambda + Serverless**

<https://www.youtube.com/watch?v=71cd5XerKss>

SHUT UP

SHOW ME DEMO

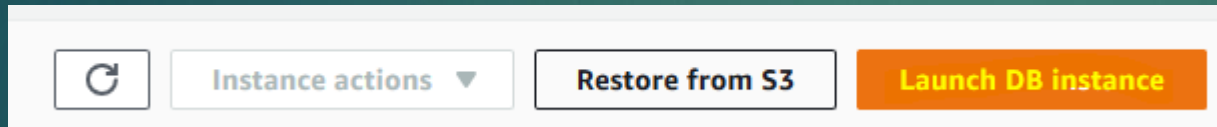


Pre-requisites

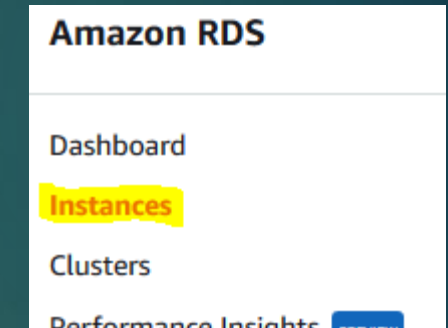
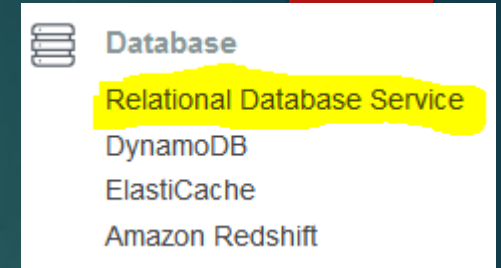
- AWS Free Tier
- Source Code For Cloning
<https://github.com/roommen/serverless101>
- Basic knowledge of Python, HTML, JS, CSS
- A good IDE like Visual Studio Code

Let's start up our DB

- Login to AWS Console
- Select “Relational Database Service” from Database
- Click “Instances” from the left-menu
- Click on “Launch DB Instance”

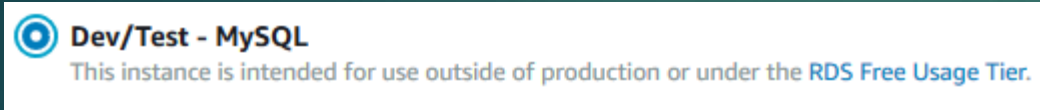


- Select “MySQL” as the engine and click “Next”



MySQL Setup

- In next screen, choose “Dev/Test – MySQL”



- Instance Specifications

DB engine
MySQL Community Edition

License model [info](#)
general-public-license ▼

DB engine version [info](#)
mysql 5.6.37 ▼

DB instance class [info](#)
db.t2.micro — 1 vCPU, 1 GiB RAM ▼

Multi-AZ deployment [info](#)
☐ Create replica in different zone
Creates a replica in a different Availability Zone (AZ) to provide data redundancy, eliminate I/O freezes, and minimize latency spikes during system backups.
☒ No

Storage type [info](#)
General Purpose (SSD) ▼

Allocated storage
20 GB
(Minimum: 20 GB, Maximum: 16384 GB) Higher allocated storage [may improve](#) IOPS performance.

MySQL Advanced Settings - Network & Security

Network & Security

Virtual Private Cloud (VPC) [info](#)

VPC defines the virtual networking environment for this DB instance.

Default VPC (vpc-59541030) ▼



Only VPCs with a corresponding DB subnet group are listed.

Subnet group [info](#)

DB subnet group that defines which subnets and IP ranges the DB instance can use in the VPC you selected.

default ▼

Public accessibility [info](#)

☒ Yes

EC2 instances and devices outside of the VPC hosting the DB instance will connect to the DB instances. You must also select one or more VPC security groups that specify which EC2 instances and devices can connect to the DB instance.

☐ No

DB instance will not have a public IP address assigned. No EC2 instance or devices outside of the VPC will be able to connect.

Availability zone [info](#)

No preference ▼

VPC security groups

Security groups have rules authorizing connections from all the EC2 instances and devices that need to access the DB instance.

☒ Create new VPC security group

☐ Choose existing VPC security groups

Keep everything as
the default setting

MySQL Advanced Settings – Database options

Database options

Database name

webapp

Note: if no database name is specified then no initial MySQL database will be created on the DB Instance.

Database port

TCP/IP port the DB instance will use for application connections.

3306

DB parameter group [info](#)

default.mysql5.6

Option group [info](#)

default:mysql-5-6

☐ Copy tags to snapshots

IAM DB authentication [info](#)

☐ Enable IAM DB authentication

Manage your database user credentials through AWS IAM users and roles.

☒ Disable


Provide an appropriate DB name

MySQL Advanced Settings – Encryption & Backup


Encryption

Encryption

- ☐ Enable Encryption
Select to encrypt the given instance. Master key ids and aliases appear in the list after they have been created using the Key Management Service(KMS) console. [Learn More](#).
- ☒ Disable Encryption

 The selected engine or DB instance class does not support storage encryption.

Backup

 Please note that automated backups are currently supported for InnoDB storage engine only. If you are using MyISAM, refer to detail [here](#).

Backup retention period [info](#)

Select the number of days that Amazon RDS should retain automatic backups of this DB instance.

7 days ▼

Backup window [info](#)

- ☐ Select window
- ☒ No preference

Leave everything as default

MySQL Advanced Settings – Monitoring, Log, Maintenance

Monitoring

Enhanced monitoring

☐ Enable enhanced monitoring
Enhanced monitoring metrics are useful when you want to see how different processes or threads use the CPU.

☒ Disable enhanced monitoring

Log exports

Select the log types to publish to Amazon CloudWatch Logs

☐ Audit log

☐ Error log

☐ General log

☐ Slow query log

IAM role
The following service-linked role is used for publishing logs to CloudWatch Logs.

RDS Service Linked Role

Maintenance

Auto minor version upgrade [info](#)

☒ Enable auto minor version upgrade
Enables automatic upgrades to new minor versions as they are released. The automatic upgrades occur during the maintenance window for the DB instance.

☐ Disable auto minor version upgrade

Maintenance window [info](#)
Select the period in which you want pending modifications or patches applied to the DB instance by Amazon RDS.

☐ Select window

☒ No preference


Cancel

Previous

Launch DB instance

- Leave everything as default
- Click “Launch DB Instance”

MySQL Getting Initialized

 **Your DB instance is being created.**
Note: Your instance may take a few minutes to launch.

Connecting to your DB instance

Once Amazon RDS finishes provisioning your DB instance, you can use a SQL client application or utility to connect to the instance.
[Learn about connecting to your DB instance](#)

[All DB instances](#) [View DB instance details](#)

- It may take sometime for DB to be initialized and available depending on your AZ/Region

MySQL Endpoint

- Once the DB creation is successful, you should have something like this:

RDS > Instances > lambda101

lambda101

Instance actions ▼

Summary

Engine MySQL 5.6.37	DB instance class info db.t2.micro	DB instance status available	Pending maintenance none
------------------------	---	---------------------------------	-----------------------------

Connect

Endpoint lambda101.██████████.ap-south-1.rds.amazonaws.com	Port 3306	Publicly accessible Yes
---	--------------	----------------------------

Security group rules (2)

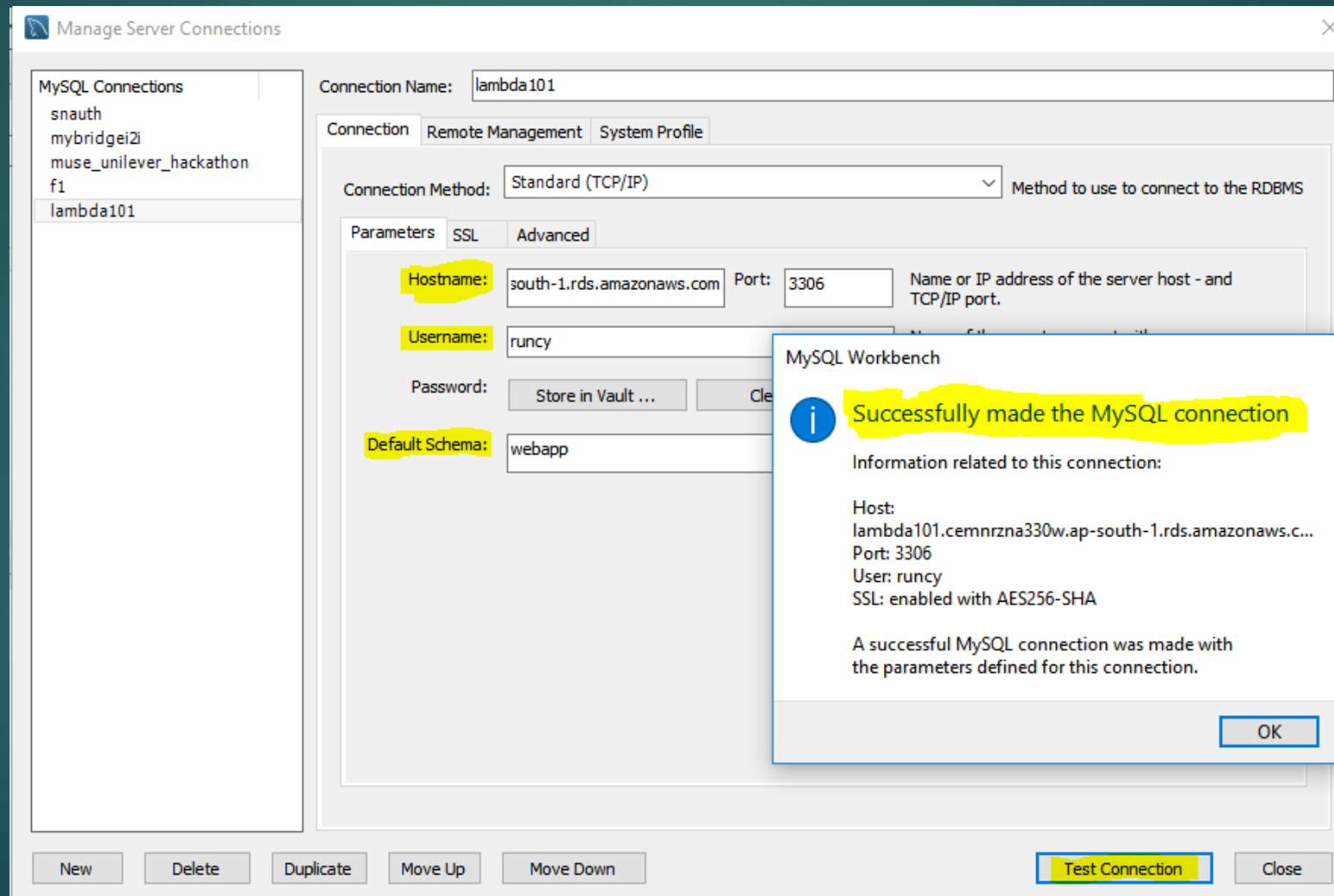
Filter security group rules

Security group	Type	Rule
rds-launch-wizard-1 (sg-c4c766af)	CIDR/IP - Inbound	115.160.251.210/32
rds-launch-wizard-1 (sg-c4c766af)	CIDR/IP - Outbound	0.0.0.0/0

Make sure you've the right inbound and outbound rules associated with the security group

Test the connection

Use a software like MySQL Workbench to test connection, view table details, run queries etc..



Creating Users table

- Go to the cloned “serverless101” repository
- Navigate to the “db” folder
- Edit the ‘Create_Users.py’ file with the DB info you created earlier

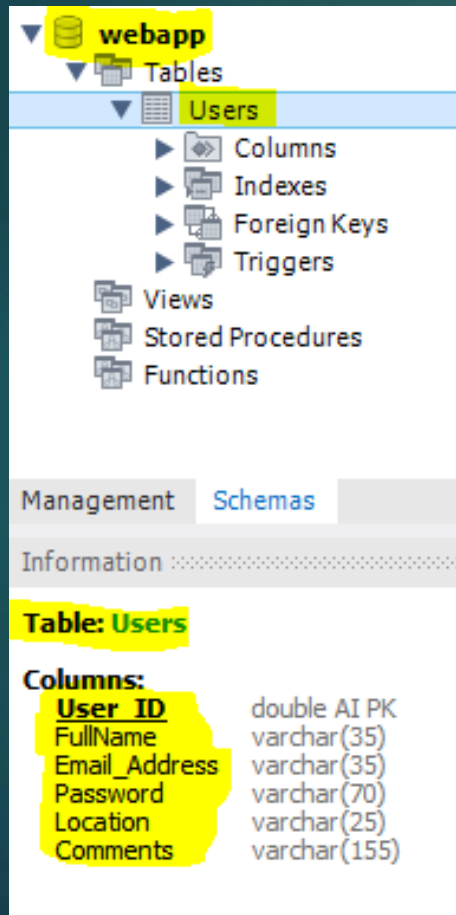
Edit the details in Create_Users.py

```
def create_users():
    ....connection, cursor = None, None
    ....try:
    ....    ....#Database Connection Parameters -- Replace this with your DB endpoint
    ....    ....lambda101_cnx_str = {'host': 'lambda101. [REDACTED] .ap-south-1.rds.amazonaws.com',
    ....    ....    'username': 'runcy',
    ....    ....    'password': '[REDACTED]',
    ....    ....    'db': 'webapp'}
    ....    ....connection = mysql.connector.connect(host=lambda101_cnx_str['host'], user=lambda101_cnx_str['username'],
    ....    ....    password=lambda101_cnx_str['password'], database=lambda101_cnx_str
    ....    ....cursor = connection.cursor()
    ....    ....cursor.execute('CREATE TABLE Users('
    ....    ....    'User_ID DOUBLE NOT NULL AUTO_INCREMENT PRIMARY KEY,'
    ....    ....    'FullName VARCHAR(35) NOT NULL,'
    ....    ....    'Email_Address VARCHAR(35) NOT NULL,'
    ....    ....    'Password VARCHAR(70) NOT NULL,'
    ....    ....    'Location VARCHAR(25) NOT NULL,'
    ....    ....    'Comments VARCHAR(155) NOT NULL)'
    ....    ....    ';')
    ....    ....print("Table Users created successfully.")
    ....except mysql.connector.Error as err:
    ....    ....print(err)
    ....finally:
    ....    ....if connection:
    ....    ....    ....connection.close()
    ....    ....if cursor:
    ....    ....    ....cursor.close()

if __name__ == '__main__':
    ....create_users()
```

Run the Create_Users.py file

```
runcy@RUNCYOOMMEN-PC:/mnt/f/serverless101/db$ python3 Create_Users.py  
Table Users created successfully.
```



- Go to MySQL Workbench
- Verify the Users table got created successfully

AWS Lambda with Python - Steps

- In this web app example, we have:
 - User Registration – handled by `serverless/register_login.py`
 - User Login – handled by `serverless/login.py`
 - User Display – handled by `serverless/users.py`
- Edit each of these `.py` files with DB connection parameters as created earlier
- For Python to be enabled as AWS Lambda function, we need to zip all our source code and dependencies – we use *mysql.connector* as a dependency in each of these files

AWS Lambda with Python – Extract dependencies

- Create a temp folder called *register* and copy *register_login.py* to it
- Do a pip install of the *mysql-connector* under that folder
(Use specific version 2.1.4 – I was getting an error for the latest one)

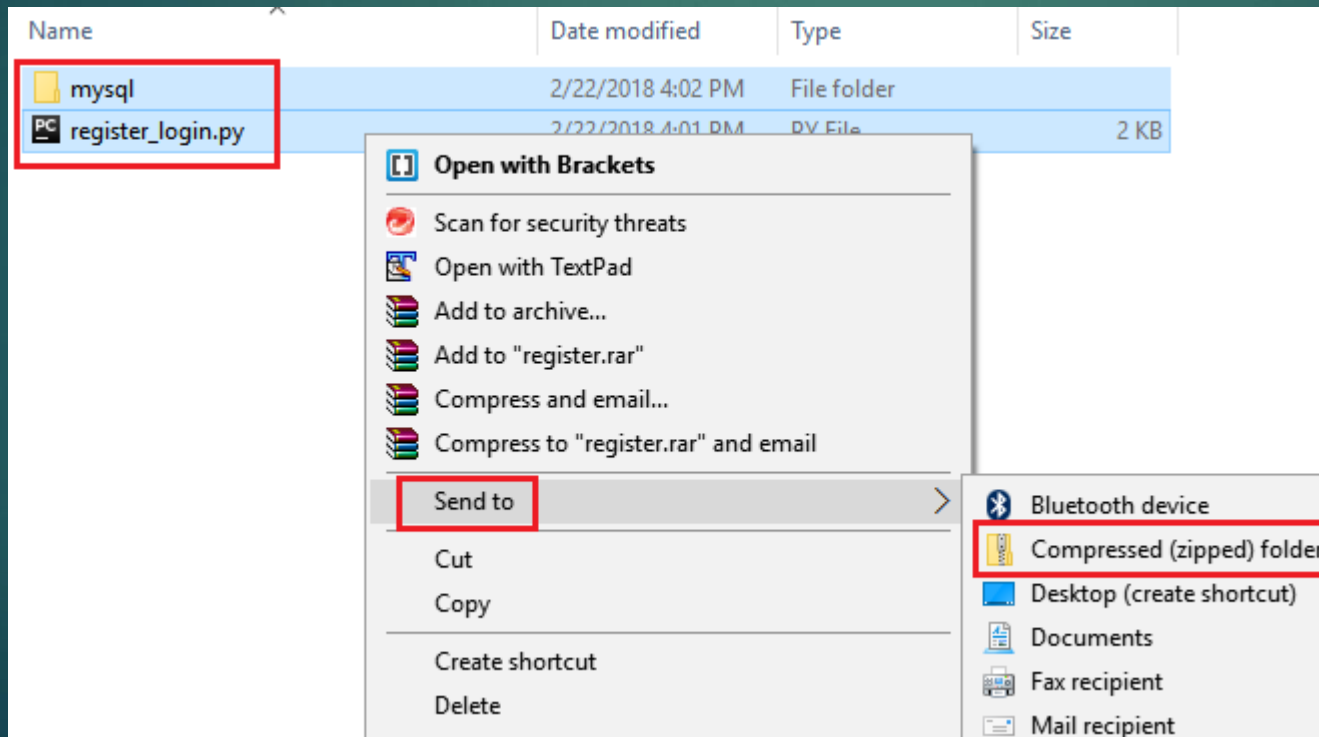
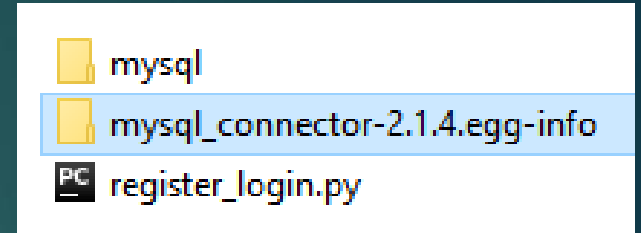
```
runcy@RUNCYOOMMEN-PC:/mnt/f/serverless101/serverless$ mkdir register
runcy@RUNCYOOMMEN-PC:/mnt/f/serverless101/serverless$ cp register_login.py register
runcy@RUNCYOOMMEN-PC:/mnt/f/serverless101/serverless$ cd register/
runcy@RUNCYOOMMEN-PC:/mnt/f/serverless101/serverless/register$ pip3 install mysql-connector==2.1.4 --target .
Downloading/unpacking mysql-connector==2.1.4
  Downloading mysql-connector-2.1.4.zip (355kB): 355kB downloaded
  Running setup.py (path:/tmp/pip_build_runcy/mysql-connector/setup.py) egg_info for package mysql-connector

    warning: no files found matching 'README.txt'
Installing collected packages: mysql-connector
  Running setup.py install for mysql-connector
    Not Installing C Extension

    warning: no files found matching 'README.txt'
Successfully installed mysql-connector
Cleaning up...
```

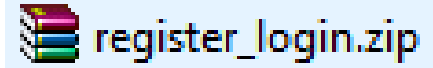
AWS Lambda with Python – Zip ‘em up

- Under the register folder, You might see a folder *mysql_connector-2.1.4.egg-info* which can be deleted if you want to
- Select the rest (*register_login.py* **file** and the *mysql* **folder**) and extract it to a zip file by right-clicking on it



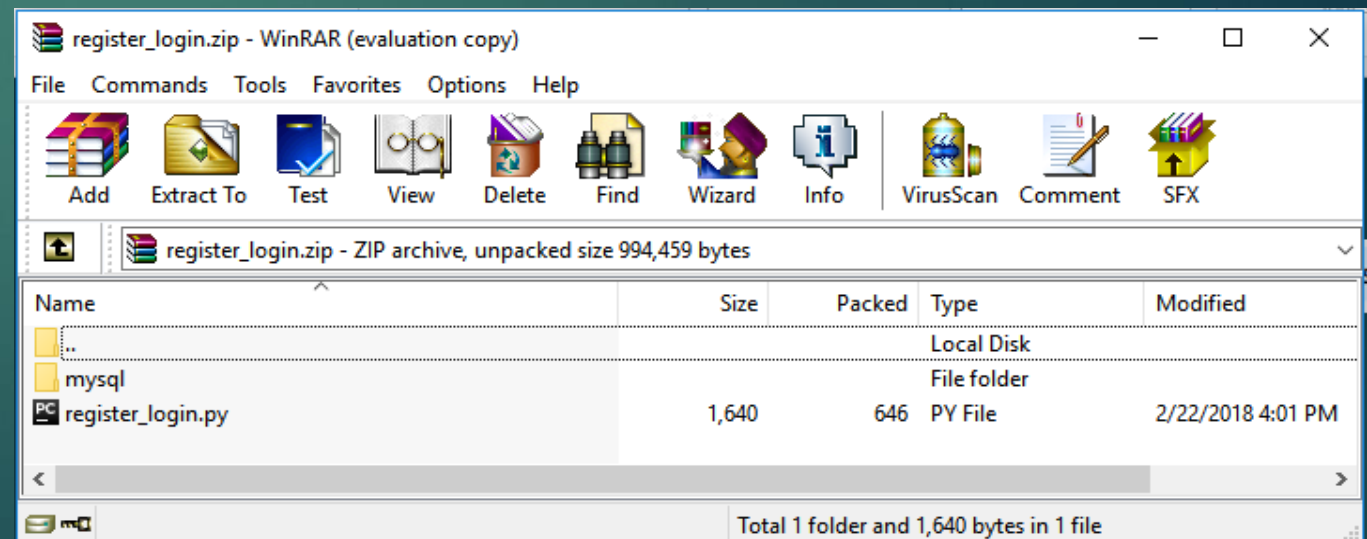
AWS Lambda with Python – Zip details

- You should now have a register_login.zip file created
- Verify the contents of this zip file and ensure that the contents look identical to screenshot below



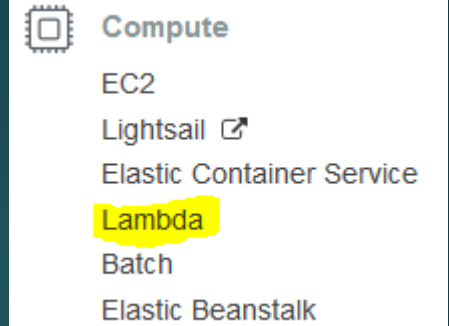
NB: The *register_login.py* file and *mysql* folder should be visible as is and not under another folder inside the zip file. Otherwise there will be problems while creating the lambda functions (later steps)

Repeat this process for *login.py* and *users.py* to create *login.zip* and *users.zip*



Let's create the Lambda functions

- Login to AWS Console
- Select “Lambda” from Compute
- Click “Create function”
- Select “Author from scratch”



Create function

Author from scratch

Start with a simple "hello world" example.



User Registration - Lambda function creation

Author from scratch [Info](#)

Name*

lambda101_register_login

Runtime*

Python 3.6

Role*

Defines the permissions of your function. Note that new roles may not be available for a few minutes after creation. [Learn more](#) about Lambda execution roles.

Choose an existing role

Existing role*

You may use an existing role with this function. Note that the role must be assumable by Lambda and must have Cloudwatch Logs permissions.

lambda_basic_execution

Cancel

Create function

User Registration - Lambda function code

- In the next screen, upload the zip file created earlier (*register_login.zip*) and change the Handler info to *register_login.lambda_handler*
- The format of the Handler should be *<python_filename>.lambda_handler*

Function code [Info](#)

Code entry type: Upload a .ZIP file ▼

Runtime: Python 3.6 ▼

Handler [Info](#): register_login.lambda_handler

Function package*

📁 Upload register_login.zip (335.3 kB)

For files larger than 10 MB, consider uploading via S3.

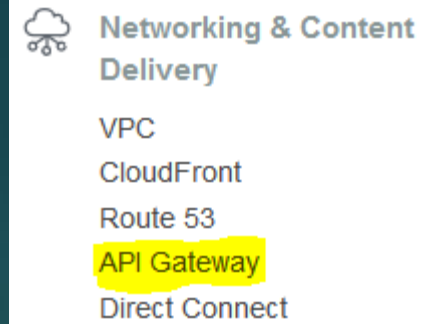
- Once done, click “Save”

Save

- Do this for each of the remaining zip files to create three lambda functions for user registration, login and view details

Integration with API Gateway

- Login to AWS Console
- Select “API Gateway” from Networking & Content Delivery
- Click “Create API”
- Choose “New API”, provide name and other details



Create new API

In Amazon API Gateway, an API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

☒ New API ☐ Clone from existing API ☐ Import from Swagger ☐ Example API

Settings

Choose a friendly name and description for your API.

API name*

Description

Endpoint Type

* Required

Create API

API Gateway – Create Resource (register-login)

- In the next screen, choose “Create Resource” from Actions and provide appropriate details

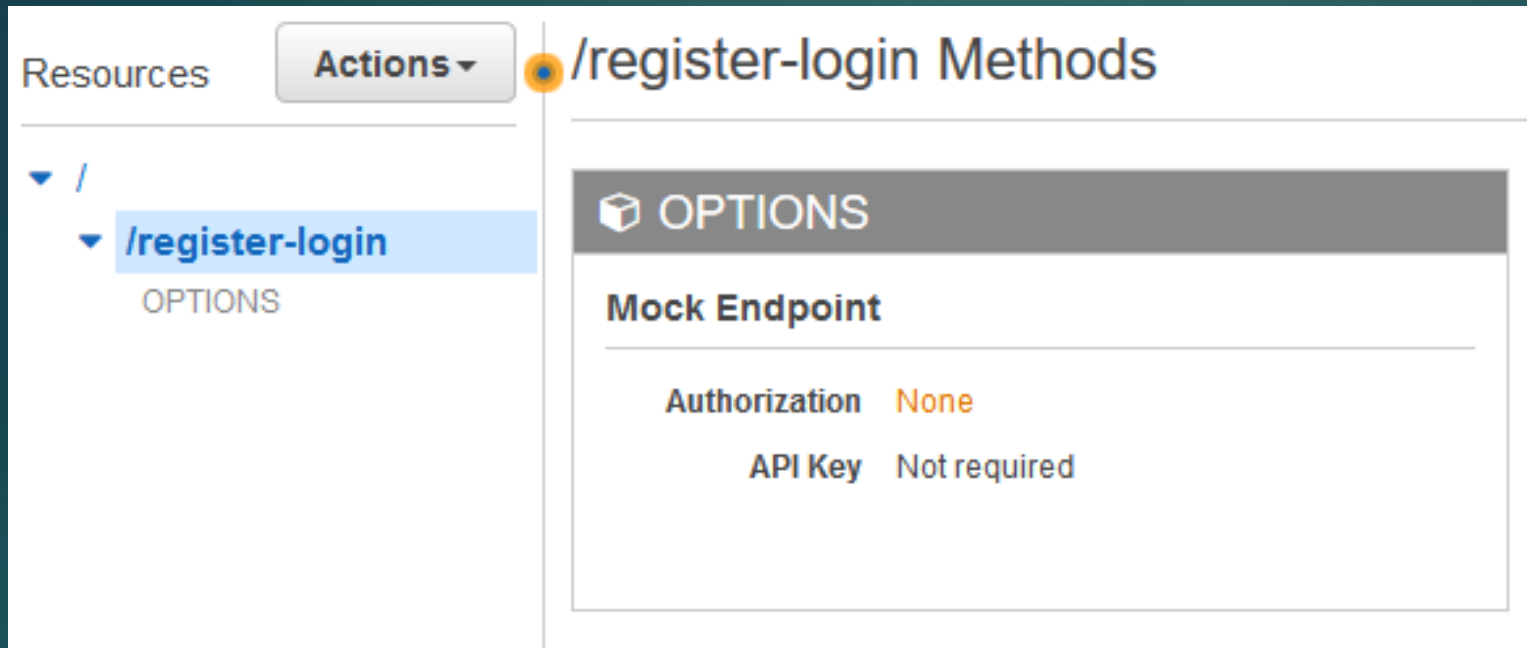
The screenshot shows the AWS API Gateway console interface for creating a new child resource. The left sidebar shows the navigation menu with 'APIs' selected, and 'lambda101demo' is highlighted under the 'APIs' section. The 'Resources' section for 'lambda101demo' shows a single resource with the path '/'. The 'Actions' dropdown menu is open, and 'Create Resource' is selected. The main form area is titled 'New Child Resource' and contains the following fields and options:

- Resource Name***: register_login
- Resource Path***: / register-login
- Enable API Gateway CORS**: ☒

Below the form fields, there is a note: "You can add path parameters using brackets. For example, the resource path {username} represents a path parameter called 'username'. Configuring /{proxy+} as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to /foo. To handle requests to /, add a new ANY method on the / resource."

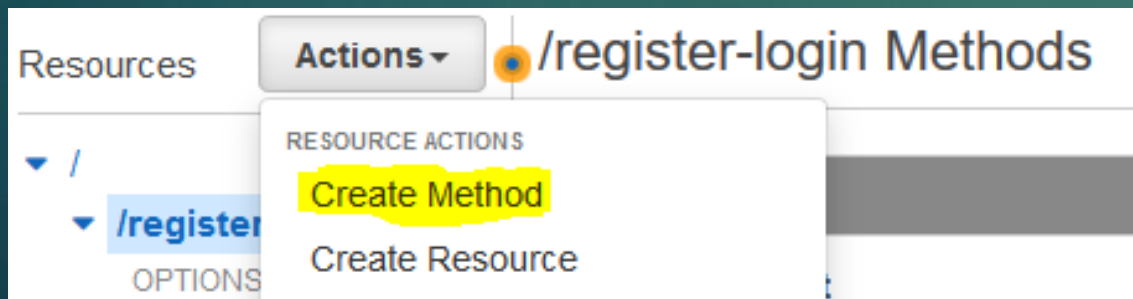
At the bottom right, there are two buttons: 'Cancel' and 'Create Resource'.

API Gateway – Resource created (register-login)



You should see a screen similar to this after the resource is created

API Gateway – Create Method



- Select the resource and now click “Create Method”
- Choose “POST”

API Gateway – Configure POST (register-login)

Click on the “POST” method and enter the configuration as below

/register-login - POST - Setup

Choose the integration point for your new method.

Integration type ☒ Lambda Function ⓘ

☐ HTTP ⓘ

☐ Mock ⓘ

☐ AWS Service ⓘ

☐ VPC Link ⓘ

Use Lambda Proxy integration ☐ ⓘ

Lambda Region ▼

Lambda Function ⓘ

Use Default Timeout ☒ ⓘ

- Select the appropriate region to choose the lambda function which we had created earlier

API Gateway – POST - Method Response (register-login)

Click on the “POST” method created and choose “Method Response”

← Method Execution /register-login - POST - Method Response

Provide information about this method's response types, their headers and content types.

HTTP Status
200

Response Headers for 200

Name
Access-Control-Allow-Origin
+ Add Header

Response Body for 200

Content type	Models
application/json	Empty
+ Add Response Model	

+ Add Response

Method Response

HTTP Status: 200

Models: application/json => Empty

Click on “Add Header”
and provide value as
Access-Control-Allow-Origin

API Gateway – POST - Integration Response (register-login)

Click on the “POST” method created and choose “Integration Response”

← Method Execution /register-login - POST - Integration Response

First, declare response types using [Method Response](#). Then, map the possible responses from the backend to this method's response types.

Lambda Error Regex	Method response status	Output model	Default mapping
-	200		Yes

Map the output from your Lambda function to the headers and output model of the 200 method response.

Lambda Error Regex

Content handling

[Cancel](#) [Save](#)

▼ Header Mappings

Response header	Mapping value
Access-Control-Allow-Origin	*

▼ Body Mapping Templates

Content-Type
application/json

[+ Add mapping template](#)

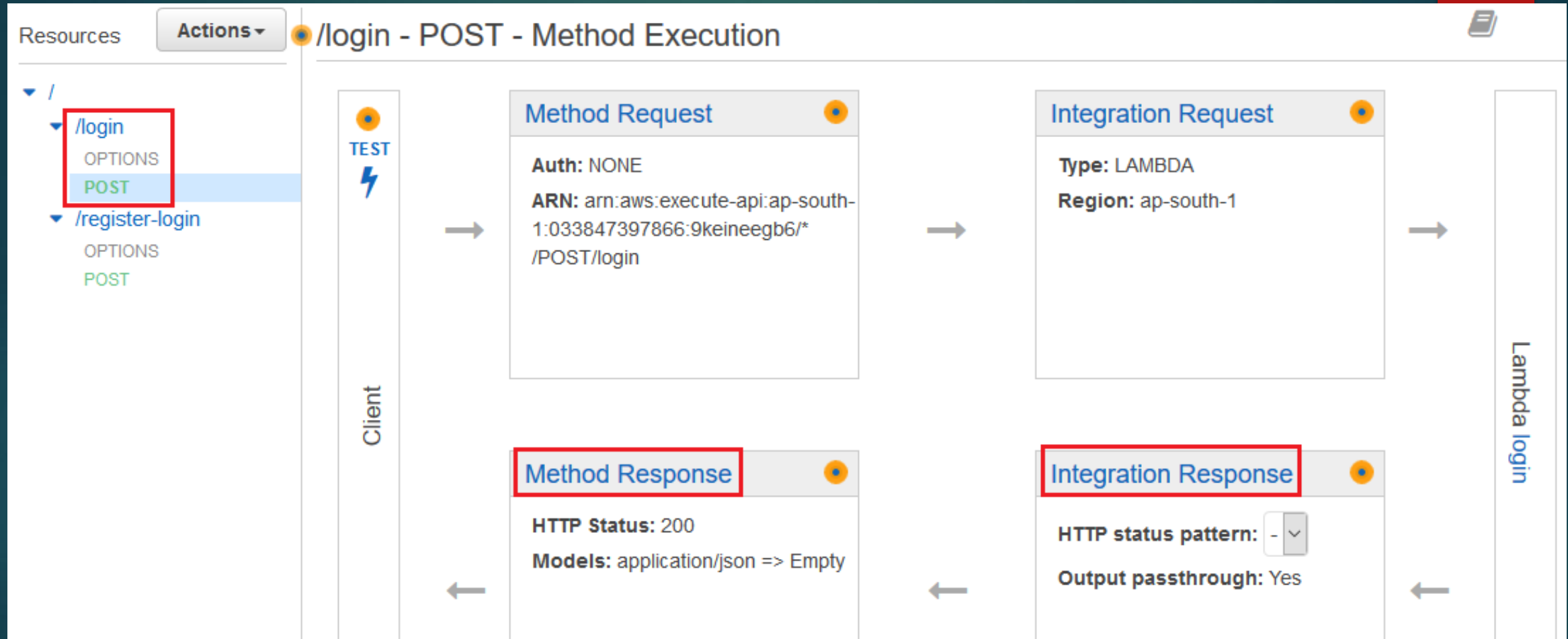
Integration Response

HTTP status pattern:

Output passthrough: Yes

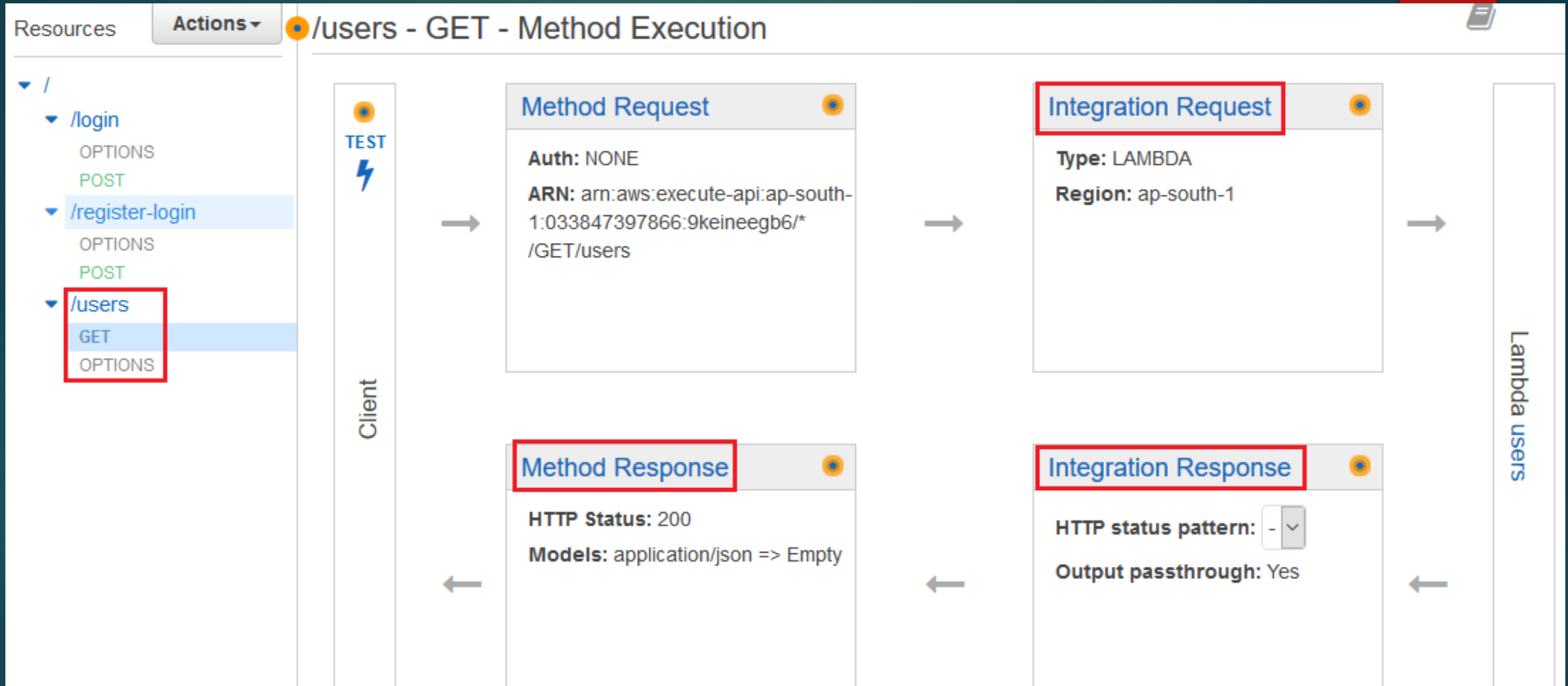
Click on the pencil icon next to *Access-Control-Allow-Origin* and add the value ‘*’

API Gateway – Create Resource (login) & POST Method



- Create *login* resource and associate **POST** method
- Configure *Integration Response* & *Method Response*
- Follow identical steps as the previous *register-login* for configuration

API Gateway – Create Resource (users) & GET Method



- Create *login* resource and associate **POST** method
- Configure *Integration Response* & *Method Response* as done previously

API Gateway – GET – Integration Request (users)

Click on the “GET” method created and choose “Integration Request”

▼ Body Mapping Templates

Request body passthrough

☐ When no template matches the request Content-Type header ⓘ

☒ When there are no templates defined (recommended) ⓘ

☐ Never ⓘ

Content-Type
application/json

+ Add mapping template

application/json

Generate template:

```
1 {  
2   "user_id" : "$input.params('user_id')"  
3 }
```

Configure the *Body Mapping Templates* with *Content-Type* as *application/json*

```
{  
  "user_id" : "$input.params('user_id')"  
}
```

users.html, showUserInfo() - lambda101.js & users.py (Lambda)

```
<script>
...
$(document).ready(function(){
  showUserInfo();
});
</script>
```

users.html

THE
HOLY
TRINITY

```
$.ajax({
  url: info_url,
  type: 'GET',
  data: {"user_id": user_id},
  dataType: 'html',
  async: false,
  success: function(data)
  {
    var result = $.parseJSON(data);
```

showUserInfo() – lambda101.js

Generate template:


```
1 {
2   "user_id" : "$input.params('user_id')"
```

AWS Lambda config
(Integration Request)

```
29 ... if cursor:
30 ... cursor.close()
31
32 def lambda_handler(event, context):
33     user_id = event['user_id']
34     return users(user_id)
35
```

users.py


It's time to deploy!

Actions  [← Method Explorer](#)

METHOD ACTIONS
[Edit Method Documentation](#)
[Delete Method](#)

RESOURCE ACTIONS
[Create Method](#)
[Create Resource](#)
[Enable CORS](#)
[Edit Resource Documentation](#)
[Delete Resource](#)

API ACTIONS
Deploy API
[Import API](#)
[Edit API Documentation](#)
[Delete API](#)

Deploy API 

Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

Deployment stage

[New Stage] ▼

Stage name*

lambda101

Stage description

Lambda 101 Dev Test

Deployment description

My first deployment of Lambda

[Cancel](#) [Deploy](#)

Choose *[New Stage]* and provide appropriate values

Get the deployed API endpoints

The screenshot displays the AWS API Gateway console interface. On the left, the 'APIs' section lists 'lambda101' with a 'Stages' link highlighted in yellow. The main panel shows the 'Stages' for 'lambda101', with a 'POST' method for the '/login' endpoint highlighted in yellow. The right panel, titled 'lambda101 - POST - /login', shows the 'Invoke URL' as 'https://[redacted].execute-api.ap-south-1.amazonaws.com/lambda101/login' and the 'Settings' as 'Inherit from stage'.

- After deployment, the APIs would be available at Stages
- For example, click on **POST** method created for */login* and see the URL
- Similar ones would exist for the **POST** of */register-login* and **GET** of */users*

Enable the APIs – Edit the JS functions

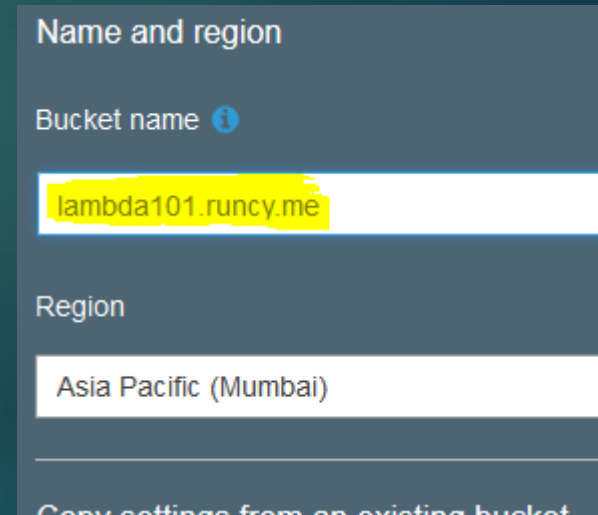
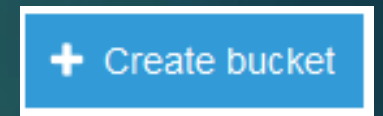
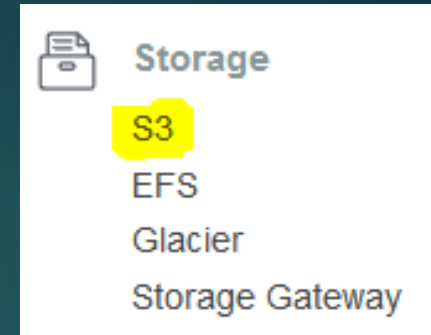
```
/*-Login-*/  
function login(auth_details)  
{  
    var result = null;  
    if((auth_details.email) && (auth_details.password))  
    {  
        $("#error").css('visibility', 'hidden');  
        passwordValue = SHA256(auth_details.password)  
        login_url = 'https://api.execute-api.execute-api.ap-south-1.amazonaws.com/lambda101/login';  
        var obj = new Object();  
        obj.email = auth_details.email;  
        obj.password = passwordValue;  
  
        var jsonObj = JSON.stringify(obj);  
  
        $.ajax({  
            url: login_url,  
            type: 'POST',  
            data: jsonObj,  
            dataType: 'json',  
            success: function(result)  
            {  

```

- Integrate each of these APIs with the relevant functions defined in lambda101.js to have them eventually invoked

Let's host the web files

- Login to AWS Console
- Select “S3” from Storage
- Click “Create bucket”
- Provide appropriate name (a subdomain or domain that you own host hosting the site)
- Click “Create”



Enable Static Website Hosting

- Select the bucket that you created earlier
- From the “*Properties*” tab select *Static website hosting*
- Provide appropriate *Index document* and hit Save
- You will now see an endpoint available which will serve you the website contents

Static website hosting

Endpoint : <http://lambda101.runcy.me.s3-website.ap-south-1.amazonaws.com>

☒ Use this bucket to host a website [Learn more](#)

Index document [i](#)

Error document [i](#)

Redirection rules (optional) [i](#)

☐ Redirect requests [Learn more](#)

☐ Disable website hosting

[Cancel](#) [Save](#)

Enable appropriate Bucket Policy

- Click on the “Permissions” tab
- Select “Bucket Policy” sub-tab
- Enter the below policy to make it world readable

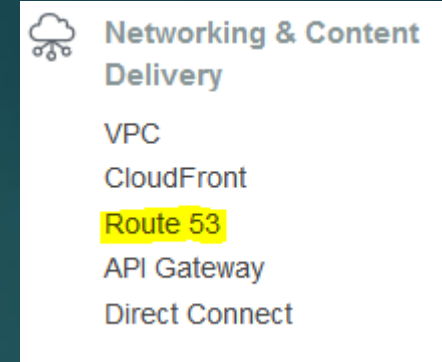
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::lambda101.runcy.me/*"
    }
  ]
}
```

The screenshot shows the AWS IAM console interface for a bucket named 'lambda101.runcy.me'. At the top, there are three tabs: 'Overview', 'Properties', and 'Permissions'. The 'Permissions' tab is selected and highlighted in yellow. Below the tabs, there are three buttons: 'Access Control List', 'Bucket Policy', and 'CORS configuration'. The 'Bucket Policy' button is highlighted in blue and has a 'Public' label. Below these buttons, the text 'Bucket policy editor' is followed by the ARN 'arn:aws:s3:::lambda101.runcy.me'. A prompt says 'Type to add a new policy or edit an existing policy in the text area below.' Below this is a text area containing a JSON policy document. The policy is numbered 1 through 12 on the left side of the text area.

```
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Sid": "PublicReadGetObject",
6       "Effect": "Allow",
7       "Principal": "*",
8       "Action": "s3:GetObject",
9       "Resource": "arn:aws:s3:::lambda101.runcy.me/*"
10    }
11  ]
12 }
```

Let's setup DNS

- Login to AWS Console
- Select “Route 53” from Networking
- Select your *Hosted Zone* for the website*
- Click Create Record Set



Create Record Set

* Assuming you have a website that is managed with Route 53. Setting will vary from provider to provider if using anything else like GoDaddy, Big Rock etc...

Create Record Set

- Provide the subdomain name on which you want the site to be available
- Select Type as “A” record which is an alias to the S3 bucket that was created earlier
- Click Create button
- Wait for sometime for records to propagate (usually 3-4 mins)

The screenshot shows the 'Create Record Set' form in the AWS Route 53 console. The form is titled 'Create Record Set' and contains the following fields and options:

- Name:** A text input field containing 'lambda101' followed by a dropdown menu showing '.runcy.me'.
- Type:** A dropdown menu with 'A – IPv4 address' selected.
- Alias:** Radio buttons for 'Yes' (selected) and 'No'.
- Alias Target:** A text input field containing 's3-website.ap-south-1.amazonaws.com'.
- Alias Hosted Zone:** A dropdown menu with '— S3 website endpoints —' selected. Below this, a list of options is shown, with 's3-website.ap-south-1.amazonaws.com' highlighted in yellow. The list includes:
 - CloudFront distribution: example.cloudfront.net
 - Elastic Beanstalk environment CNAME: example.elasticbeanstalk.com
 - ELB load balancer DNS name: example-1.us-east-1.elb.amazonaws.com
 - S3 website endpoint: s3-website.us-east-2.amazonaws.com
 - Resource record set in this hosted zone: www.example.com
- Routing Policy:** A dropdown menu with 'Simple' selected.
- Evaluate Target Health:** Radio buttons for 'Yes' and 'No' (selected).
- Create Button:** A green button labeled 'Create' at the bottom right.

Your site is *now* LIVE!!!





Thank You!

Q & A

Runcy Oommen
<https://runcy.me>