OIDC Lab – part 2

This lab is a continuation of the OIDC lab from last week. You need to complete last week's lab before starting this one. Before starting this lab, you should have the following:

- Auth0 account and application
- Functioning REST API (API Gateway, Lambda)
- VueJS code cloned and running on your local machine. This webapp should be calling your REST API (with the obtained ODIC access_token) and receiving a response.

The last step in our OIDC two-part lab is to secure your API to only allow access to requests made with a valid token. This week's lab is based on this Auth0 Tutorial, and you find the descriptions of the steps valuable in your learning outside of lab. When this lab is complete, you will have secured your API to authorized user access only.

Specifically, your lambda authorizer will:

- Confirm that an access token has been passed via the authorization header of the request
- Verify the RS256 signature of the access token is using a public key that is obtained from the JKWS endpoint
- Ensure the access token has the required issuer ("iss") and audience ("aud") claims

Every time API Gateway receives a request, it checks to see if a custom authorizer has been configured for that API. If it has, API Gateway defers the authorization check to this lambda function. This new lambda authorizer function determines whether the request is allowed to proceed or denied. The return value of the custom lambda authorizer function will be a security policy that dictates the permissions allowed.

Let's begin!

Log into Auth0

- Go to "Applications" – "APIs". Take note of the following values which you will need in the sections below:
  - Identifier - Use this as the "Audience" of your .env file. Replace "https://your-api-gateway" with your identifier.
  - Domain - Use this as the {yourDomain} value of your .env file.

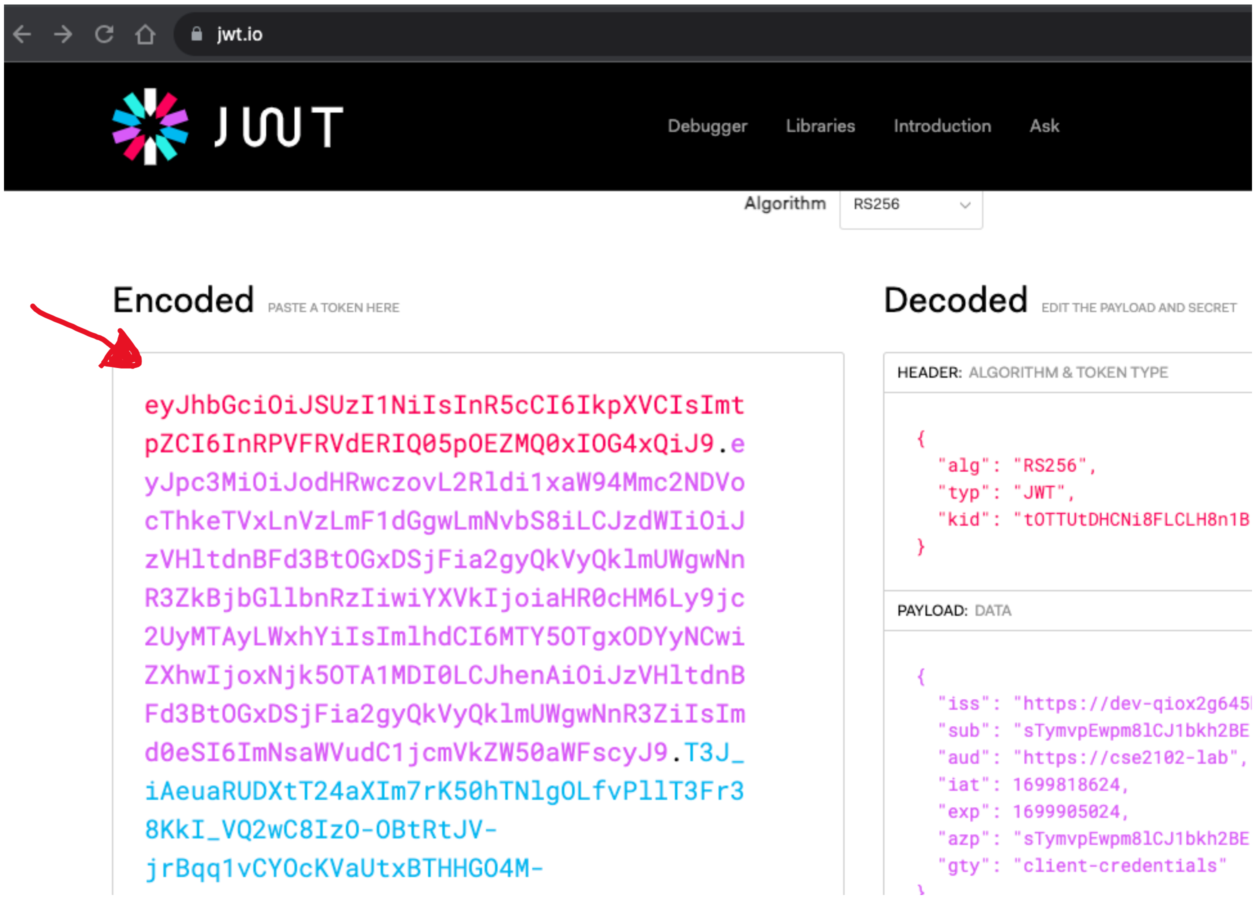
Create the custom authorizer

- Clone this repo
- In a Terminal, go to the root of the cloned repo and run "npm install"
- Create a ".env" file. Substitute both {yourDomain} and https://your-api-gateway with the values above.

```
JWKS_URI=https://{yourDomain}/.well-known/jwks.json
```

```
AUDIENCE=https://your-api-gateway
TOKEN_ISSUER=https://{yourDomain}/
```

- Test locally
  - Get a test token from the Auth0 Dashboard > Applications > APIs. Select your API and select the "Test" tab.
  - Copy the entire "Curl" command and run it in a terminal window.
  - Copy the value of the "access_token" emitted.
  - In your browser, navigate to https://jwt.io and paste your access_token value into the "Encoded" box and you should see the decoded token.



Algorithm RS256

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImt
pZCI6InRPVFRVdERIQ05pOEZMQ0xIOG4xQiJ9.e
yJpc3MiOiJodHRwczovL2Rldi1xaW94Mmc2NDVo
cThkeTVxLnVzLmF1dGgwLmNvbS8iLCJzdWIiOiJ
zVHltdnBFd3BtOGxDSjFia2gyQkVyQklmUWgwNn
R3ZkBjbGllbnRzIiwiYXVkIjoiaHR0cHM6Ly9jc
2UyMTAyLWxhYiIsImlhdCI6MTY5OTgxODYyNCwi
ZXhwIjoxNjk5OTA1MDI0LCJhenAiOiJzVHltdnB
Fd3BtOGxDSjFia2gyQkVyQklmUWgwNnR3ZiIsIm
d0eSI6ImNsaWVudC1jcmVkZW50aWFscyJ9.T3J_
iAeuaRUDXtT24aXIm7rK50hTNlgOLfvPllT3Fr3
8KkI_VQ2wC8IzO-OBtRtJV-
jrBqq1vCYOcKVaUtxBTHHGO4M-
```

Decoded EDIT THE PAYLOAD AND SECRET

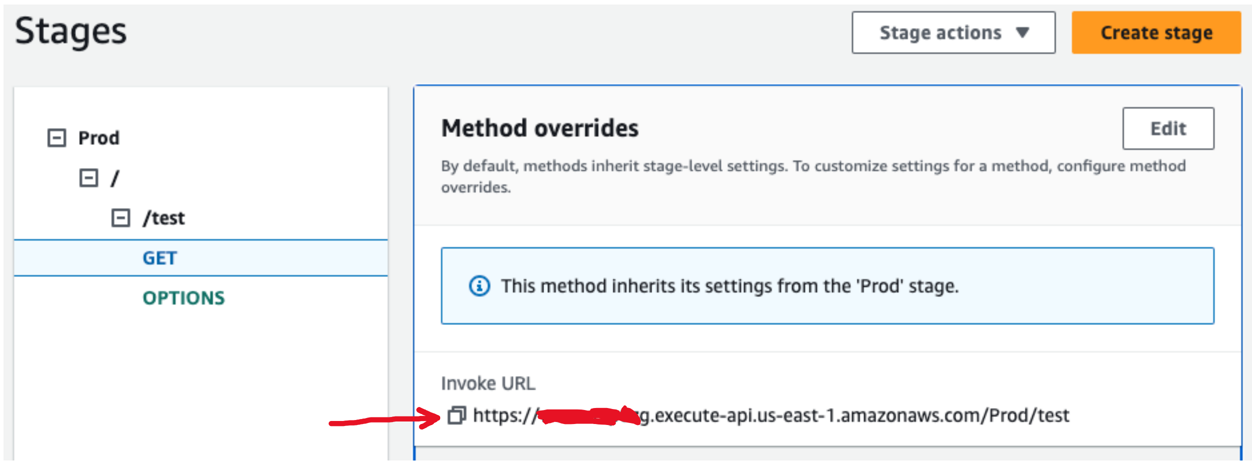
HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "tOTTUtDHCNi8FLCLH8n1B
}
```

PAYLOAD: DATA

```
{
  "iss": "https://dev-qiox2g645
  "sub": "sTymvpEwpm8lCJ1bkh2BE
  "aud": "https://cse2102-lab",
  "iat": 1699818624,
  "exp": 1699905024,
  "azp": "sTymvpEwpm8lCJ1bkh2BE
  "gty": "client-credentials"
}
```

  - Open your newly cloned repo in VSCode.
  - In the root of your project, find the "event.json.sample" file, copy it, and name the new file "event.json". In this new file, replace the following values:
    - ACCESS_TOKEN with your access_token value (from above)
    - The value of methodArn with the ARN from your API Gateway GET method.



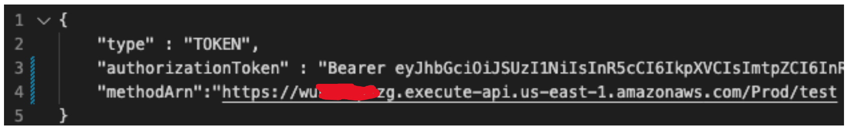
By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.

This method inherits its settings from the 'Prod' stage.

Invoke URL

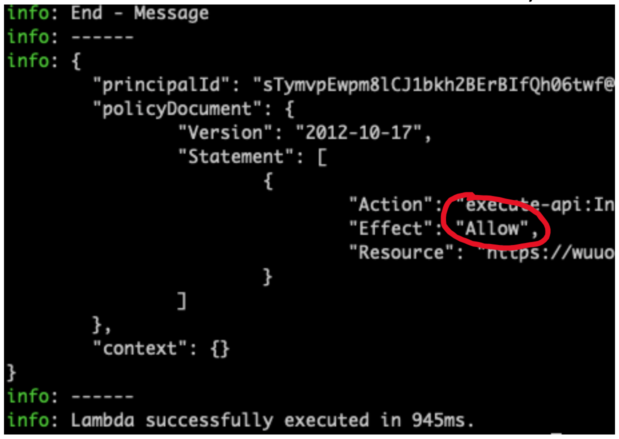
https://[redacted]g.execute-api.us-east-1.amazonaws.com/Prod/test

You file should look similar to this:



```
{
    "type" : "TOKEN",
    "authorizationToken" : "Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6InR
    "methodArn":"https://wu[redacted]zg.execute-api.us-east-1.amazonaws.com/Prod/test
}
```

- In the terminal, run a local test by entering "npm test ." (Note: don't forget the dot at the end of that command!)

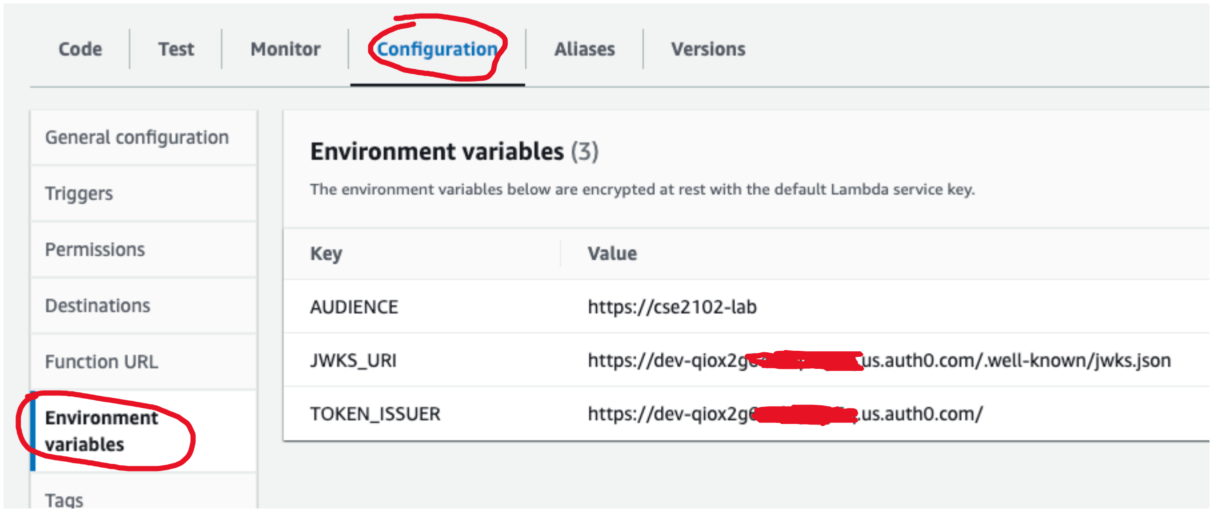


```
info: End - Message
info: ------
info: {
        "principalId": "sTymvpEwpm8lCJ1bkh2BErBIfQh06twf@
        "policyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                        {
                                "Action": "execute-api:In
                                "Effect": "Allow",
                                "Resource": "https://wuuo
                        }
                ]
        },
        "context": {}
}
info: ------
info: Lambda successfully executed in 945ms.
```

- If you see a message like the one above with the "Allow" status, then your test worked!
- Try editing the access_token value in your event. json by deleting a couple characters. Now your token is invalid because it was tampered with! Rerun the "npm test ." and see if you can get the Disallow or Unauthorized status.
- "Undo" your changes to event.json so that the token is valid once again.

- Deploy the custom lambda authorizer to AWS
  - In a terminal, type "npm run bundle"
    - That command will generate a "custom-authorizer.zip" file that can be uploaded to lambda.
  - In the AWS Leaner Lab, select the Lambda service.
  - Click "Create Function".

        - Name it: "my-custom-lambda-authorizer"
        - Select "Node.js 18.x"
        - Click "Change default execution role", select "User an existing role", and select "LabRole".
        - Click "Create Function"
    - Click the button "Upload from" and ".zip file". Click "Upload" and navigate to your newly created "custome-authorizer.zip" file. Click "Save" to complete the upload.
    - Note: you will NOT be able to see the code in the inline code editor within AWS. And you will also notice that Lambda functions do NOT need to be a single literal function. You can deploy a full application to a Lambda function as long as it is not over the size limit.
- Create environment variables
    - Note: Do you remember how we had 3 values in our .env file? That works fine for running the code locally. But on AWS, we need to create these 3 "environment variables" in the lambda configuration since .env files will not work in lambda.
    - In the Learner Lab and with your AWS function still selected, click the "Configuration" horizontal tab and then the "Environment Variables" vertical tab.
    - Click "Edit" and add the 3 variables and their corresponding values from your .env file. Your screen should resemble the screenshot below.

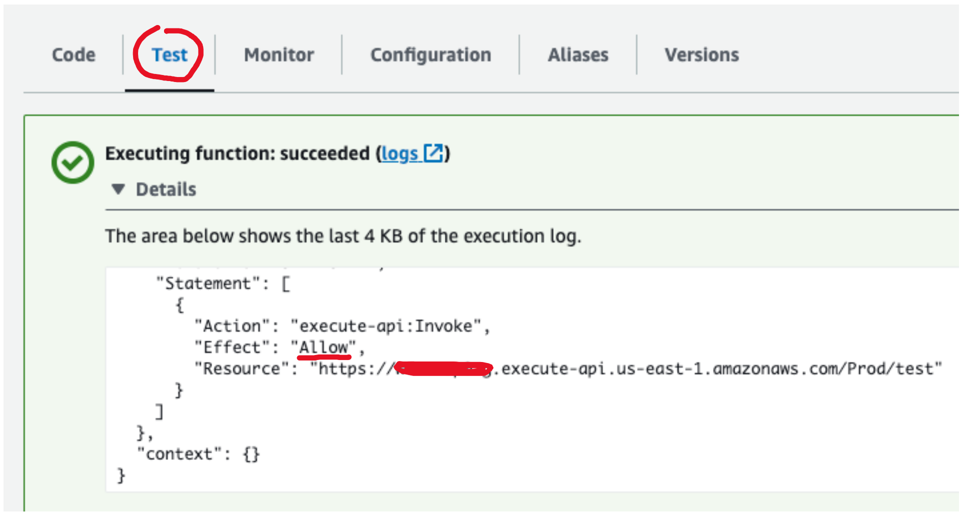


Environment variables (3)

The environment variables below are encrypted at rest with the default Lambda service key.

Key	Value
AUDIENCE	https://cse2102-lab
JWKS_URI	https://dev-qiox2g[redacted]us.auth0.com/.well-known/jwks.json
TOKEN_ISSUER	https://dev-qiox2g[redacted]us.auth0.com/

- Test your lambda function
    - Click the "Test" horizontal menu
    - Give it a name like "authtest"
    - Copy the entire contents of your event.json file into the "Event JSON" box.
    - Click "Save" and then "Test"
    - You should see results similar to your local test



Executing function: succeeded (logs)

Details

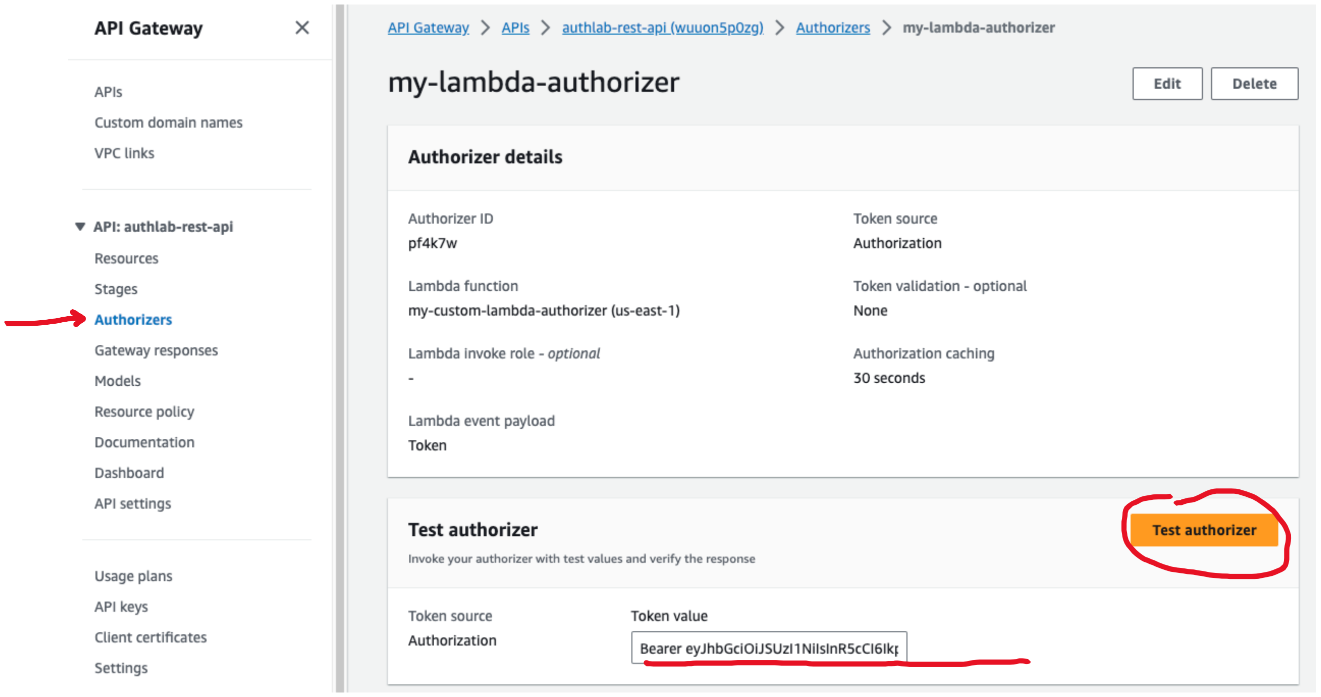
The area below shows the last 4 KB of the execution log.

```
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow",
        "Resource": "https://[redacted].execute-api.us-east-1.amazonaws.com/Prod/test"
      }
    ]
  },
  "context": {}
}
```

Configure API Gateway to use the custom lambda authorizer

Now you need to configure your API Gateway to use your new Lambda Authorizer function.

- In your AWS Learner Lab, go "API Gateway" and select "Authorizers" in the vertical menu. Click "Create authorizer".
- Give it a name like "my-lambda-authorizer" and select the lambda function you created above. (Note: Make sure you select the correct lambda function! If you named your lambdas as we have suggested, it should be called "my-custom-lambda-authorizer". If you cannot find your new lambda function, make sure you are in the correct region. E.g. us-east-1.)
    - Authorizer type: Lambda
    - Keep the default: "Token" event payload
    - Lower the TTL to 30 seconds (instead of the default 300)
    - Enter "Authorization" as the Token Source
    - Lambda invoke role - you should be able to keep this blank
    - Click "Create Authorizer"
- Test your authorizer
    - Add the "Bearer ey........" value from your event.json file to the "Token value". (Note: it will be very long.)
    - Click "Test Authorizer". You should again see that same "Allow" message as you received in previous tests above.



my-lambda-authorizer

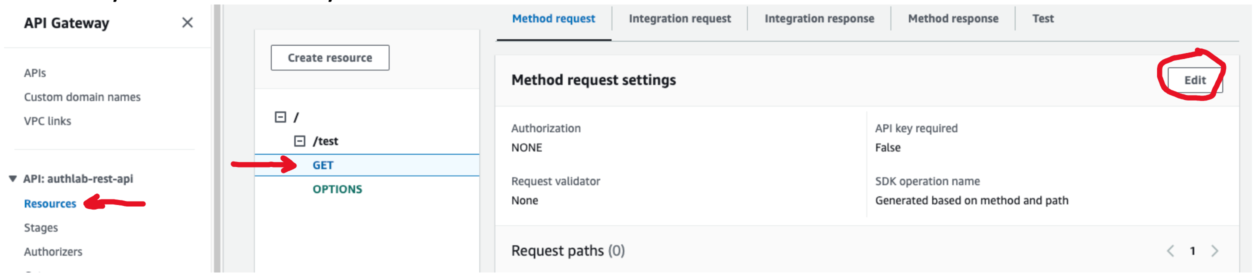
Authorizer details

Authorizer ID: pf4k7w
Token source: Authorization
Lambda function: my-custom-lambda-authorizer (us-east-1)
Token validation - optional: None
Lambda invoke role - *optional*: -
Authorization caching: 30 seconds
Lambda event payload: Token

Test authorizer
Invoke your authorizer with test values and verify the response

Token source: Authorization
Token value: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp

- Configure your REST API to use your new authorizer
  - Edit your API Gateway "GET" Resource

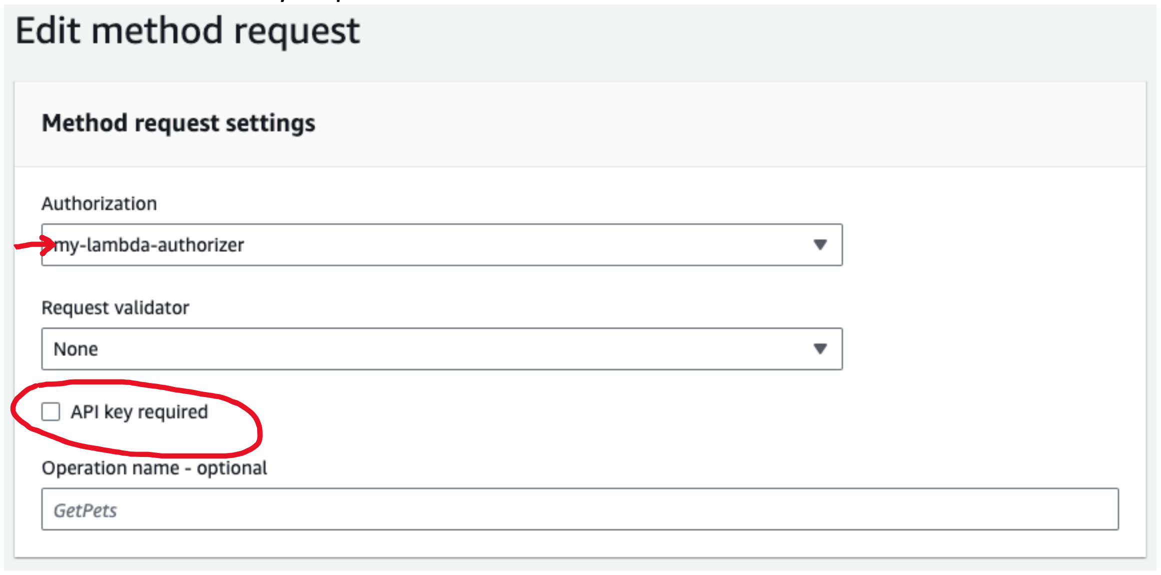


Method request settings

Authorization: NONE
API key required: False
Request validator: None
SDK operation name: Generated based on method and path

Request paths (0)

  - Select your "my-lambda-authorizer" from the Authorization list
  - Leave the "API key required" UNCHECKED.



Edit method request

Method request settings

Authorization: my-lambda-authorizer

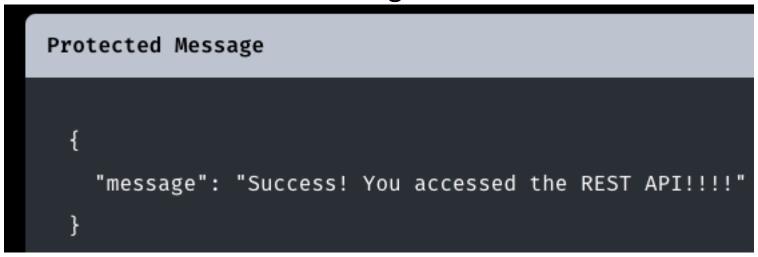
Request validator: None

☐ API key required

Operation name - optional: *GetPets*

    - Click "Save"
    - Click "Deploy API". Select your "Prod" stage and click "Save".

- Test your API
    - Curl from the Terminal
        - curl https:// aabbccddee.execute-api.us-east-1.amazonaws.com/Prod/test
            - What response did you get? Why?
        - curl -H "Authorization: Bearer eyaaaaa" https://aabbccddee.execute-api.us-east-1.amazonaws.com/Prod/test
            - Do you get a different result when including the Authorization Bearer in the header of the HTTP request? Why?
        - Note: Use YOUR specific URLs instead of the ones labeled "aabbccddee" above. Hint: look in your .env file... Use YOUR access_token instead of the one labeled "eyaaaaa" above. Yes, it will be very long. Hint: look in your event.json file.
    - Test from the webapp
        - Click "Log Out" if you already logged in earlier
        - Click "Log In" and select the Google login flow
        - Did the profile page populate with your data from your Google identity?
        - Click on the "Protected" menu on the top of the webpage. Do you see your message generated from the lambda function?
        - Try modifying the message generated by lambda and see if your webapp shows this new message.



```
{
  "message": "Success! You accessed the REST API!!!!"
}
```

You have completed the lab!

Conclusion

Congratulations! You have secured your API to <u>only</u> allow authenticated requests with valid access tokens. In this way, your API and your Webapp are now leveraging a Federated Identity provider called Auth0 for authentication.

Auth0 also supports many additional features that we have discussed in class:

- You can easily add new "Social Connection" types (Authentication - Social) such as Facebook, Apple, LinkedIn, etc.

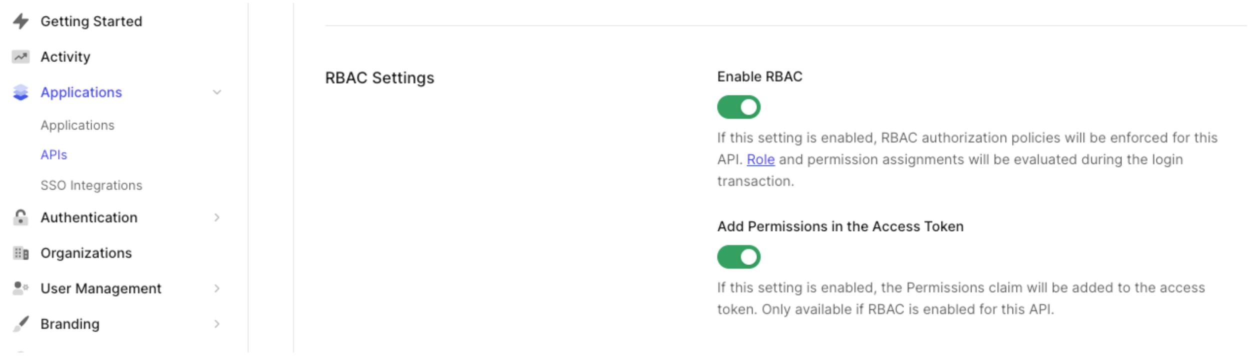
- You setup Multi-factor authentication (Security - Multifactor Auth) like phone, email, push notifications, and DUO.
- You can easily send your security logs to a log aggregators like Splunk.
- Auth0 also supports protection (Security - Attack Protection) from attack vectors such as Bots, Brute-force, and suspicious IP addresses.

All of these security improvements are available to your application without you having to write any of the code!

Potentially Useful Information for Group Projects

In Auth0, you will notice that the "User Management" tab allows you to create Users and Roles for Role-base Access Control (RBAC). You might consider creating Roles for "Student", "Instructor", and "Admin". Once a user logs in for the first time, they will appear in the list on the Users screen. You will be able to assign users to a give role. In that way, you can create some test users to function as students, instructors and admin.

You can also create "Permissions" for a Role (Roles - Permissions). Select your API in the list, and then create a set of permissions such as, "read:courses", "write:courses", etc. Then you need to configure your API to use RBAC by selecting the options "Enable RBAC" and "Add Permissions in the Access Token".



RBAC Settings

Enable RBAC

If this setting is enabled, RBAC authorization policies will be enforced for this API. Role and permission assignments will be evaluated during the login transaction.

Add Permissions in the Access Token

If this setting is enabled, the Permissions claim will be added to the access token. Only available if RBAC is enabled for this API.

Once you have deployed your API, you will notice that the access_token now contains the assigned permissions granted to a given user. (Note: use http://jwt.io to examine the new contents of one of your tokens. Hint: you can use the Developer Console in Chrome and go to the Network tab to view all of your http traffic. Click on the commands generated when you go to the "Protected" tab and you will see your access_token in the header of some of the requests.)

To make full use of Users and Roles in your project, you would also need to modify the custom lambda authorizer javascript code to check if the required permission for the given endpoint resource (e.g. GET courses, POST courses, GET students, etc) is present in the access token. Then you could adjust your authorizer code to Allow/Deny based on more logic than the simple token validation that was done for this lab. You have the endpoint that is being called (context variable in the lambda function handler) and the decoded token that contains the permissions array (decoded.permissions[0], etc).

A basic login with Auth0 should be attainable for all groups. But the use of Roles and Permissions in the lambda authorizer might be a stretch for some groups. Successfully getting it to work for your group project will be seen as "extra credit" for sure! Please tell your TA which students were the ones to successfully implement these features.

However, make sure you properly prioritize all your features so that if you are not able to get the Roles/Permissions working, that you still have a great project to show at the end of the semester.