Cloud Applications Architecture
Lab 6

Aim of the Laboratory

1. Delegating user management (Cognito)
2. Setting up the API Gateway
3. Working with Queues and Lambda

Intro

Our app is working great. It does what it's supposed to do, it is secure, and it scales decently. In many cases, the current state might be good enough especially if your aim is not to reach Facebook/Amazon-like scale. However, if you do, a few things can be done to facilitate that.

Right now, the app is a **monolith**. There is one artifact/service providing and handling all the functionality. The bigger the monolith gets, the harder it will be to add features since the whole application has to be built, tested, and deployed. Moreover, if there is a bug making the app unusable, the entire system becomes unusable.

The natural approach and way forward are to break the application into separate services (a.k.a. microservices). This might be a good medium to a long-term goal, but in the short term, we can apply this approach only for new features and/or to extract certain features/domains that are divided (e.g. user management).

User Management

Our spring application does not have to handle users. Instead, we can use dedicated services to help us manage users and offer the most common operations such as password reset, email verification, etc.

Cognito

Cognito is the AWS service dedicated to user management. It has 2 components:

- **User pools**
- Identity pools

For our purpose, we will leverage **user pools**. (Identity pools enabling us to grant access to the AWS account/infrastructure itself - think of it as social sign-in for your AWS account). We will create a user pool that will enable us to adopt a new authentication/authorization scheme based on OAuth.

Follow the next steps:

1. Go to Cognito, choose Manage User Pools
2. Create a new user pool named caacourse
3. Edit the attributes (first section) as follows:
    a. Change to **Email address or phone number** instead of Username - **Allow email addresses**
    b. Take a look at what attributes you can collect.
    c. Can you think of any custom attributes that might make sense to collect? (during user registration)
4. You might want to relax the password policy to make it easier
5. On step "MFA and verifications", change the recovery to **Email only**
6. On step App clients, add a new app client as follows
    a. Give it any name you wish
    b. **Uncheck** the **Generate client secret**
    c. **Check** Enable username password-based authentication (ALLOW_USER_PASSWORD_AUTH)
    d. Create it
7. Review and create the user pool.
8. Copy the user pool ID.
9. Go to the App client settings and enable Cognito User Pool as the **Identity Provider**.

Feel free to take a look at the AWS Docs for User Pools (there are lots of features we don't cover in the lab).

Users

The usual approach involves having a client/web application that has the Cognito SDK (actually part of the amplify framework) installed and calls its methods to sign in/up users, generate tokens, and so on. However, we will create a user manually and use the **AWS CLI** to validate it (change its password) and retrieve the **idToken** which will enable us to call our application.

- Create a new user under **Users and Groups**
    - Use an email address as the username
    - Don't set a phone number (also uncheck the verified box)
    - Set the email address and make sure it is verified automatically (the checkbox below is set)
- Either install the AWS CLI and configure it or launch a new EC2 instance (Amazon Linux 2-based instances have the AWS CLI already installed) and enable it to call Cognito by creating a role. If you choose to launch a new EC2 instance, first create an IAM role as follows:
    - Go to IAM, Roles
    - Create a new role for EC2
    - Search for the Cognito Power User policy and attach it.
    - Name your role something suggestive, maybe ec2-cognito-admin-role.

    - Assign the newly created role to the new instance.
    - SSH into the instance (using the Connect button or an SSH client)
- Initiate the auth flow with the following command

aws cognito-idp initiate-auth --auth-flow USER_PASSWORD_AUTH --auth-parameters USERNAME=<your_cognito_user_email>,PASSWORD=<your_cognito_user_password> --client-id <your_cognito_app_client_id>

- Respond to the password challenge with the following command (Cognito requires the users to change their passwords if they were created by the admin):

aws cognito-idp respond-to-auth-challenge --client-id <your_cognito_app_client_id> --challenge-name NEW_PASSWORD_REQUIRED --session "<session_id_from_previous_response> --challenge-responses USERNAME=<your_cognito_user_email>,NEW_PASSWORD=<your_cognito_user_new_password>

- It's easier to assemble the command in a text editor. The session ID might contain some new line characters that have to be removed (you can also search for online tools)
- The response should contain an **IdToken**, **RefreshToken,** and **AccessToken**. We need the **IdToken**.
- If the token expires, you can retrieve a new one using the first command (with *initiate-auth*).

Deploy the Application

We will leverage CloudFormation again to set up our infrastructure (RDS, ECS, ALB). Create a new stack using the following template:

https://caa-lab-templates.s3-eu-west-1.amazonaws.com/lab6.json

You only have to provide the URL based on which the application will perform the token validation (it will use the URL to fetch various information about the identity provider - your user pool). (Leave the other parameter with the default value).

While the deployment occurs, ensure you have an API client (e.g. Postman) available so we can test our app.

After the deployment is complete, test the API as follows:
- Get the ALB DNS name and append */api/products* to it.
- Set the authorization to bearer and provide the idToken retrieved using the AWS CLI
- You should get a few products as a response

Extend the App

To make our lives easier, we will deploy an additional service in front of our application - API Gateway. Follow these steps:
- Go to API Gateway in the AWS console

- **Build** a new **REST API** (not HTTP)
- Select **Import from Swagger or Open API 3**
- You can find the Open API specification of our app here: https://caa-lab-templates.s3-eu-west-1.amazonaws.com/openapi.yaml
- Create the API
- Wire the API to the backend
    - Go to API Resources, */products* API, GET method
    - Select HTTP as the integration type
    - Paste the load balancer DNS name followed by */api/products*
    - Save. You should be able to test the API from the console (set as headers the following: Authorization: Bearer <your_idToken>)

A New Endpoint

API gateway enables us to add new endpoints not necessarily part of the app. We will add a new endpoint for */stocks*. This is intended to be called by our providers after they deliver new products. We will send/buffer all requests to an SQS queue from where a lambda function will read and process them (for now, just log them).

First, let's create the SQS queue:

- Go to SQS and press Create queue
- Give it a name and remember it.

Next, let's create the function that will process the messages from the queue:

- Go to Lambda and Create a new function
- Choose Use a blueprint and search for sqs
- You should find sqs-poller. Configure it.
- Give it a name and also provide a name for the role that will be created.
- Choose the SQS queue you created previously and **enable the trigger**
- Glance over the content of the function - it only outputs the received messages.
- Create the function

Before proceeding to extend the API, we have to define an IAM policy and role that will enable API Gateway to send messages to the queue:

- Go to IAM, Policies.
- Create policy
- Choose SQS as service
- Choose SendMessage (under Write) as action
- Under Resources, **add** the **ARN** by providing the region eu-west-1 and your queue name.
- Review, name, and create the policy
- Move to **Roles** and create a new one
- Choose API Gateway as the use case and create the role
- Search for it and attach the policy you created previously

Now, let's create the new API endpoint:

- Go back to your API in API Gateway

- Click on */stocks* and, from Actions, choose Create Method. Select POST as the method from the drop-down.
- Select AWS Service as integration type
- Select eu-west-1 as the region
- Select Simple Queue Service (SQS) as service type
- Select POST as HTTP method
- Change the action type to Use path override and set it to *<aws_account_id>/<queue_name>*
- As the execution role, set the ARN of the role you created previously.
- Save
- Go to **Integration Request**
- Under URL Query String Parameters, add the following:
    - Name: *MessageBody*, Mapped from: *method.request.body.MessageBody*
    - Name: Action, Mapped from: *'SendMessage'* (including the single quotes)
- Go back to the method overview and select Method Response
- Add the HTTP status 200
- Go back to Integration Response and set the HTTP status regex to 200.

Test the New Endpoint

From the method execution page, choose *TEST*. Provide the **request body** as follows:
{
  "MessageBody": "well….whatever you want. This will be the content of the message"
}
The response should be 200.
Go to the lambda function, under the monitoring tab, and choose View logs in CloudWatch.You should have at least one log stream. Find your message.

The API is not yet reachable by your providers. You have to deploy it. If you reach this point and want to deploy it, you have to add integrations for all methods or delete them.

Cleanup

As usual, delete everything you created such as:
- The lambda function
- The SQS queue
- The CloudFormation stack
- The API Gateway
- The EC2 instance (if you created one for the AWS CLI)
- The IAM roles and policy
- The Cognito user pool