FIT5225 Assignment 2 - TagStore Report

FIT5225 - TagStore Report
Kailash Sivaraman (30153867)
Henri Maurice Droulers (29377595)
Shing Pong Adrian Yip (26442833)
Hsin-Yu Pan (30198860)
Repository: https://github.com/Adryipan/TagStore.git

Table of Contents

System Design and Architecture

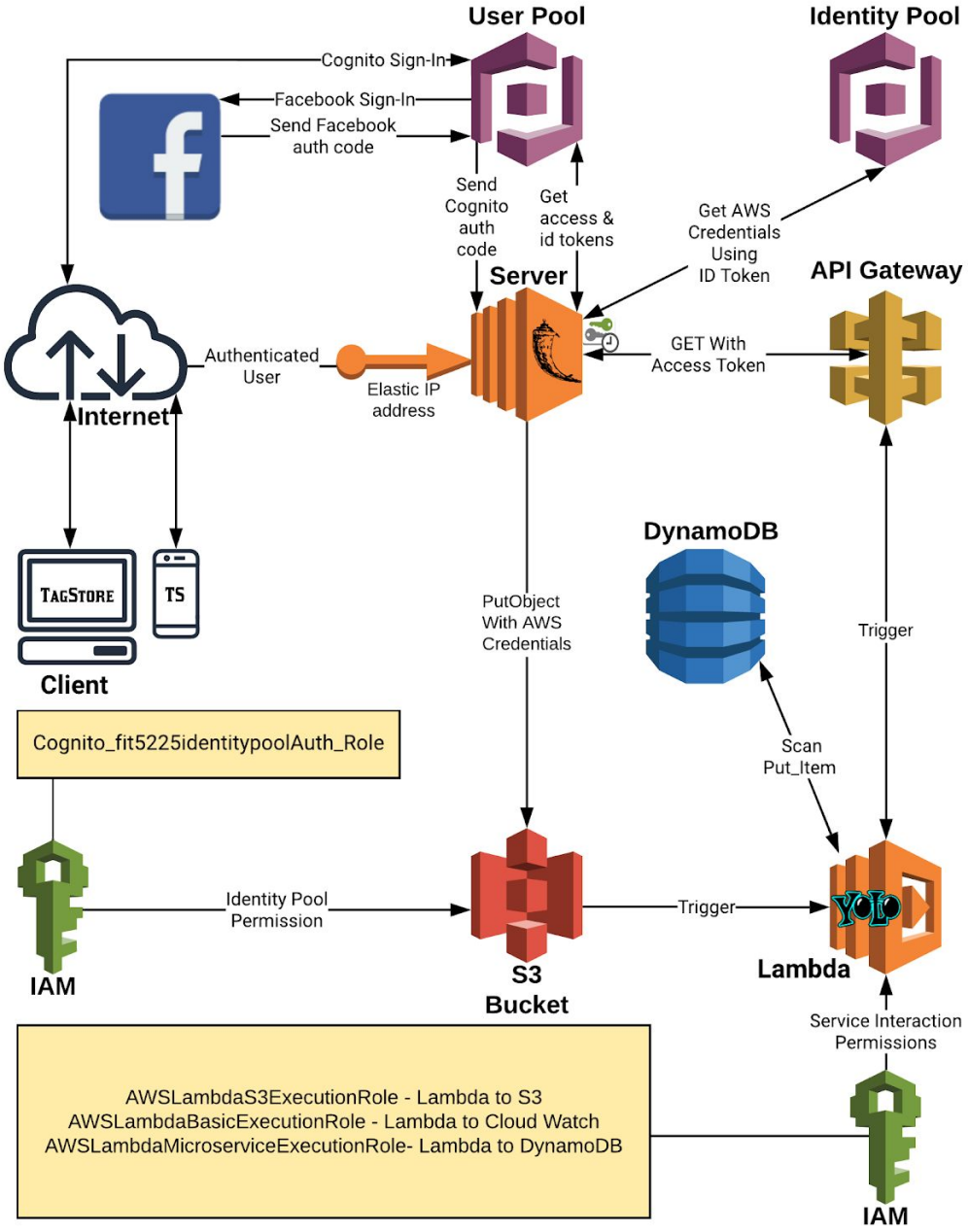


Figure 1.0 - System Design and Architecture

Client

The TagStore client is accessed by users with a web browser via the internet. It has a graphical user interface (GUI) comprising several web pages. The developers contemplated implementing a text-based interface but ultimately decided to opt for a GUI for usability purposes. The following table outlines each of the client screens.

Table 1.0 - GUI Pages

Page	Description
Sign In	An Amazon Cognito-hosted UI that accepts user credentials.
Sign Up	An Amazon Cognito-hosted UI enabling users to register an account.
Home	The landing page with a menu.
Search Image	Accepts tags as text input and returns the URLs of images in the database that contain the object(s) corresponding to the tag(s) provided. Also renders the matched images.
Upload Image	Users can select a file from their computer and upload it to TagStore. Provides notification as to the success or failure of the file upload.
Sign Out	A simple page confirming a successful sign out.

Authentication

Authentication is initiated by the user's interaction with the Cognito-hosted sign-in page. Pages of the website are inaccessible to unauthenticated users, forcing them to authenticate and login. When a set of credentials are provided as user input, they are routed to the Cognito User Pool which validates the credentials, and sends an authorization code to the application if the credentials are valid. Using the authorisation code, the application then makes an OAuth2/token call to the Cognito User Pool to gain access and ID tokens. Using the ID token, the application can then gain a set of temporary AWS credentials from the Cognito Identity Pool which are used to access internal AWS services (Specifically s3 due to permissions provided in IAM for authenticated users). And the access token returned by the call to the Cognito User Pool is sent as an 'auth' parameter for authorization and access to resources exposed through Amazon API Gateway.

The developers decided to use the Cognito-hosted sign-in page as it was already well-integrated into the Cognito ecosystem. The developers faced significant time constraints and preferred not to 'reinvent the wheel' with regard to this particular aspect of the project. The developers decided to require authentication for the resource used to GET image URLs and made available via Amazon API Gateway, as it is a public-facing resource so would be open to unauthorised use without requiring an access token. It is an inherent feature of AWS where AWS credentials, access tokens, and ID tokens expire after some time for security purposes.

External Identity Provider (Facebook)

Primary step is to create a developer account or use an existing account to establish as a developer in Facebook's developer site. We added the login feature first and set it up by providing a website (callback to cognito /oauth2/idpresponse). The same was put into valid oauth redirect uri option. All the changes have to be saved for facebook to provide responses. In the user pool of cognito, we added identity provider, set it up with app client id, secret and scopes (public_profile, email). Attributes of facebook such as id and email are mapped to cognito's username and email. Finally, we enabled the hosted ui to show

login with facebook feature. After clicking on continue with facebook, post authentication in facebook, an authorization code is sent to cognito. Cognito modifies this to a compatible code and returns it to the callback url. This feature currently works only for the developer of the app in facebook. But once this is published it can be accessed by all users.

Important Observation:
The code provided by facebook is not accessible to the server as cognito modifies it to a different code. In order to access facebook's code, a signup, sign in page should be created to bypass hosted ui features. This will allow the request and response to be controlled therein receiving the facebook code which can be exchanged for access token from oauth/access_token of facebook. The code sent from facebook can only be used once.

Server

A Flask webserver running on an Ubuntu 18.04 LTS instance managed by Amazon EC2 serves up the front-end and controls the flow of the application. The server incorporates a self-generated and signed certificate to initiate https protocol. There is also a secret used for the session, it is generated at random every time the server is run. Our decision to use Flask was in part due to our familiarity with the Flask framework and the Python programming language itself, and also due to the convenience of the built-in, production-grade webserver.

Once authenticated, the application running on the server retrieves image URLs and stores images in S3 at the user's request. Images uploaded to the server are sent to an Amazon S3 Bucket for storage. Requests for image URLs are routed to the resource exposed by Amazon API Gateway. The S3 bucket and the API gateway act as gateways to, and triggers for, the other Amazon services that comprise our distributed system, and we will expand further below on how and why we decided to use said services in our design.

Image Processing & Image Requests

Uploaded images are sent to a single Amazon S3 Bucket for storage. When a new image is sent to the Amazon S3 bucket from the web server, this event triggers a Lambda function for the processing of the image which involves passing the image to the YOLO library and writing the identified objects to the database. The particulars regarding the database will be expanded on in the following subsection. The developers' design choice for S3 as infrastructure for image storage was due to the ease in which we could pass images to and trigger the execution of our system's business logic based solely on the event of a new image upload.

Image URL requests are made to the resource we have defined and exposed via Amazon API gateway. Calling the resource's GET method with image tags as arguments triggers the execution of a Lambda function which retrieves the URL from the database for every image matching the specified criteria.

Scalability was the main reason we made the design choice of using Amazon Lambda for our application's processing of images and requests. We anticipate that demand for our service will be unpredictable so Amazon's promise of elastic, unlimited compute power will undoubtedly be invaluable regarding the need to both scale up to meet peak demand and scale down to avoid waste. Another attractive aspect of Lambda was the ability to precisely limit its access to the other components of our distributed system through the use of IAM roles which we will elaborate on in due course.

Data Persistence

Amazon's own NoSQL database, DynamoDB, was our service of choice for data persistence. Our database contains a single table which can be read from and written to by our Lambda functions. The development team decided to opt for a NoSQL database over a traditional relational database due to the nature of our system in that it is a cloud-based distributed system and based on our experience it's easier to share data across multiple components and create scalable cloud solutions with a NoSQL database. We also found that the structure and evolving nature of our data was particularly suited to a NoSQL database whereby whenever a new object is detected the database table automatically scales to accommodate that object by adding it as an attribute and indicating its presence in images with a simple "True" boolean value. This was particularly suited to our rapid development style, whereby adaptability of the NoSQL database was preferable to the more involved updating and preparation that comes with relational databases.

Role-Based Access

We used Amazon Identity Access Management (IAM) to define and assign roles to control access between components. The Lambda functions in particular have been assigned three roles to grant the necessary access capabilities. One role allows Lambda to access the S3 bucket so that it can process the image when triggered, another role allows Lambda to interface with Amazon CloudWatch for logging and monitoring purposes, and the third role allows Lambda read and write access to DynamoDB for the getting and putting of image URLs and tags. There is another role assigned to the S3 Bucket enabling the S3 bucket to be accessed by the webserver using temporary AWS credentials.

Role of Team Members

Kailash Sivaraman

- Creating and configuring user pool
- Creating and configuring identity pool
- Working with authorization code to get id and access tokens
- Exchanging access token for aws credentials using boto3
- Facebook user pool and identity pool configuration
- Configuring relations between facebook and cognito through app id, app secret and scopes
- Getting authorization code from facebook post authentication
- Ensuring all pages of website is not accessible for unauthorized users
- Configuring https with certificate creation
- Configuring session secret in flask
- Minor contribution to report.

Hsin-Yu Pan

- Front end development
    - Developed GUI using Flask, HTML and CSS.
    - Connected HTML elements to AWS endpoints using Flask
    - Processed errors and results to give feedback to the user.
    - Worked on making the websites look more professional.
- Report (user guide)

Henri Maurice Droulers

- Researched how to upload files remotely to Amazon S3.
- Implemented image upload to S3.
- Created Lambda function that identifies objects in images and is triggered by image uploads to the S3 bucket.

- Created DynamoDB database and table that has image URLs and object tags written to it by the Lambda function mentioned in the previous point.
- Wrote the “System Design & Architecture” and “Real World Application” parts of the report, as well as created the system architecture diagram.

Shing Pong Adrian Yip
- Mainly responsible for setting up API gateway to allow user to submit queries
- Implemented lambda function to query images from DynamoDB
- Implementing triggers for a lambda function to
    - Query the DynamoDB
    - Return the resulting image
- Implemented templates for integration between the python server and the HTML
- Answered Discussion questions of the report

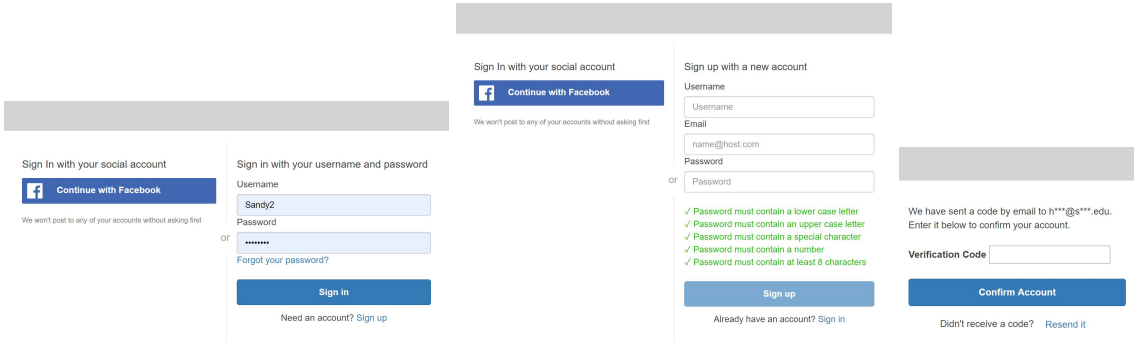
Where did the team work well
- Researching and implementing technical aspects
- Distributing work between the team
- Constructing the report

Where issues arose and how they were addressed
- Team members had different commitments with different units but we made sure that we met up when time permitted.
- One team member had to travel to the Monash campus for the internet and this issue is yet to be resolved.

User Guide

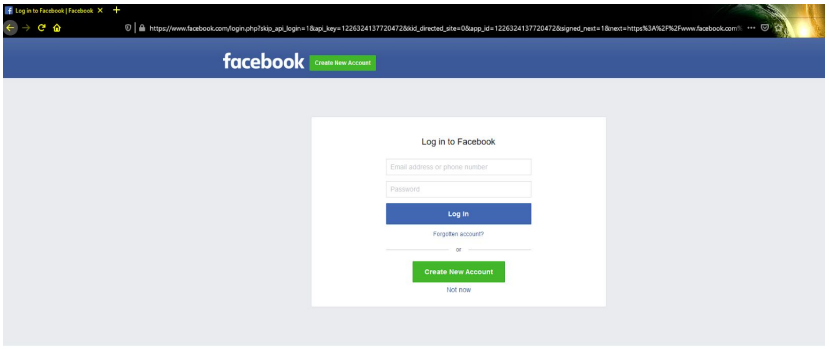
Link to our application:
https://fit5225-domain.auth.us-east-1.amazoncognito.com/login?client_id=5495h8gtqjm9cvm8sr05isfh5e&response_type=code&scope=email+openid&redirect_uri=https://174.129.206.198:4000

Sign-In & Sign-Up



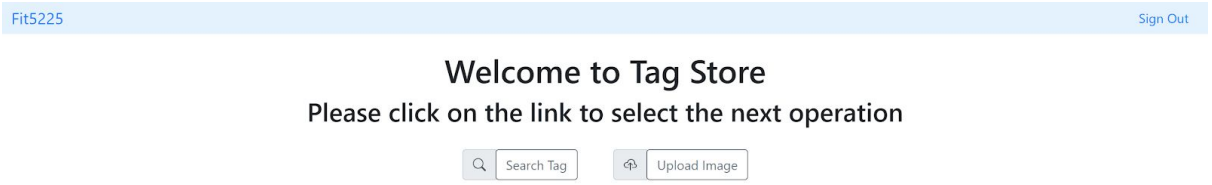
In the cognito sign in page, returning users have the option to sign in with their username and password or with their social media accounts. New users can also click the “Sign up” text below the “Sign in” button to get redirected to the sign up page, in which users are required to input their username, email, and password. Cognito will then send a verification email to the user with a verification code. After successfully verifying the account, the user will be signed in and redirected to the tag store home page.

Facebook sign in page



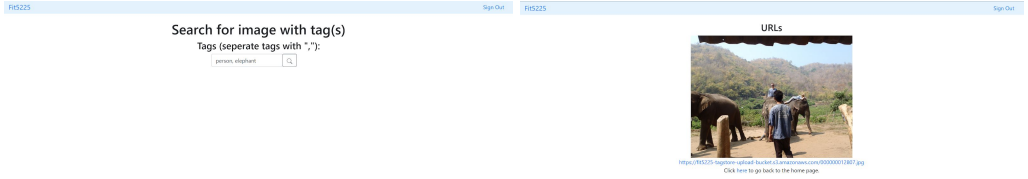
If users decide to sign in with Facebook, they will be redirected to the Facebook login page. The users can then use their Facebook credentials to log in. Once they have logged in, they will be redirected to our home page.

Home page



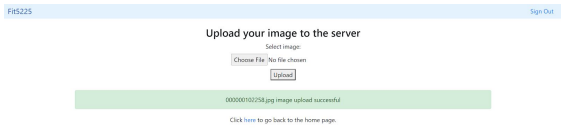
The home page provides two buttons which will redirect users to a search tag page and upload image page. The Fit5225 text on the top left corner will redirect users back to this page, and the Sign out text on the top right corner allows users to sign out from their accounts.

Search tag page & Result page



The search tag page allows users to search for images that contain the keywords the user inputs. Users can input comma separated tags in the edit text box, and click the magnifying glass button to search. Our system will return the images and their URLs that match the tags the user inputs. Users can also click on the text “fit5225” on the top left corner or the “here” text highlighted in blue at the bottom of the page to be redirected back to the home page.

Upload Image page



In the upload image page, users can click the “Choose File” button to select local image files they wish to upload. Clicking the Upload button will upload the selected image to our system. Once the images are uploaded successfully, the page will prompt a “image upload successful” message on the screen. Users can also click on the text “fit5225” on the top left corner or the “here” text highlighted in blue at the bottom to be redirected back to the home page.

Sign-Out Page

Users can click the Sign Out text on the top right corner to sign out from the account. Once the user is signed out successfully, the page will prompt a “Sign out successful” message on the screen.

Discussion

What are the updates you would perform to your application if you have users from all over the world?

To cope with a large user-base dispersed across the globe we plan to deploy the application across multiple availability zones, and create a content delivery network (CDN) involving the use of edge nodes. Content would be made specific to each zone on an as-needed basis which would require allocating one bucket per user.

If we were to anticipate that application usage would be continuous and growing then we would shift the business logic from Lambda's serverless architecture into a continuously running server, or eventually, a cluster of several servers whereby traffic is first received by a load balancer to handle multiple simultaneous requests from users. We would consider implementing the cluster within a custom-made Virtual Private Cloud (VPC) for security purposes as this would enable greater control over firewalls and access control lists (ACLs).

Some other updates we would perform:

- Support for different languages on the UI.
- Better error handling such that code is not shown in the web browser when exceptions are thrown.

What sort of design changes would you make to reduce the chance of failures in your application?

Our current implementation uses YOLO-tiny configuration and weight. However, it limits the detection performance and often returns false negative results as objects are not detected. To improve the application, full YOLO weight and configuration can be used.

Our application is currently running with a self-signed certificate. Modern browsers would prompt the users security warnings, which depending on the setting, some of the users may not be able to access the service. By obtaining a certificate issued by a Certificate Authority, users can access the application freely while it also improves our reputation. We would also use a proper domain name so as not to discourage users from using our system if their browser is uncertain about the domain name.

We only went through basic testing and debugging procedures at the current stage. By following formal software testing methods, such as including black box testing and white box testing, can help pick up potential errors and hence reduce the chances of failures.

What are the design changes you will make to increase the performance of your applications in terms of response time, query handling, and loading images?

We would perform an algorithm analysis of the business logic to identify any areas where the structure of the code and design of the logic could be improved to reduce computational complexity. Even minor improvements could make a significant difference if the program is in high demand.

One design change we would like to implement is regarding the initial processing of the images. Images can have very large file sizes (e.g. 5 - 10MB from a typical digital camera) and can be scaled down to less than 1MB while still retaining most of their quality so we would certainly add some logic to optimise the balance between image size and quality before they are stored in the S3 bucket.

Server resources would need to be increased to handle the computational load. Processing power and memory could be scaled automatically at the level of individual servers, but we anticipate that there will be some manual intervention required relating to cluster orchestration.

We would also add pagination to the webclient because as the image catalogue grows, so too does the potential for a user to get a large number of image hits and we would not want these displayed all on one page.

Real World Application

We propose a Smart Glasses™ system for the visually impaired. Headsets equipped with cameras would have the ability to identify and alert users to objects and its location relative to the user in their environment through the use of our object detection service.