

Handover Document *of* **Qualee Doc Search Pilot Project**

by
Hung Nguyen

hung.nguyenvan@inapps.net

2020.10.08 | Version 1.1

Table of Contents

I. INSTALLATION OF THE PROJECT	3
1. Setting Up Services on AWS Console.....	3
2. Configuration of the Project's Backend.....	6
3. Configuration of the project's frontend.....	7
II. PROJECT IMPLEMENTATION ON AWS	9
1. Deploying the backend system to AWS Lambda and API Gateway.....	9
2. Deploy the frontend system to AWS S3 as a static website	10
3. Set a cronjob in AWS Lambda and CloudWatch Events to run every 60 seconds	11
III. PROCESSING FILE IN THE BACKGROUND BY BACKEND	13
1. Detecting and Extracting Written Texts From File By AWS Textract.....	13
2. Getting English Meaning of Word/Phrase Entities by AWS Comprehend.....	16
3. Management of Extraction Outputs by AWS DynamoDB.....	16
4. Adding Data to Elasticsearch Service	18
IV. USAGES.....	20
1. Access Web App.....	20
2. Uploading A New Document	20
3. Frontend Process of Uploading a File	22
4. Uploading a File with treating duplicates	23
5. Searching A Term.....	24
5.1 Searching a term, in general.....	24
5.2 Backend process of searching a term.....	26
6. Updating existing file	27
6.1. Calling file update in frontend:	27
6.2. Delete related information in backend	28
6.3. Reincarnation of the file at the end of updating process.....	29
7. Deleting a file	29
7.1. Calling file deletion in frontend	29
7.2. File deletion in backend	30

I. INSTALLATION OF THE PROJECT

1. Setting Up Services on AWS Console

The following resources and/or services are used to implement and to deploy the project. The account owner (root user) can give appropriate permissions to IAM accounts in order to access these services:

Service name	Purposes	Quantity (if applicable)
DynamoDB	Storing the following data: <ul style="list-style-type: none"> - <i>Texttract</i> data from processed documents - <i>Comprehend</i> data from processed <i>Comprehend JobID</i>. - All of other information of a record, one record reflex a document which is uploaded to AWS S3. Please refer to the following parts to see more details. 	1 table
Lambda	Deploying Serverless functions for the backend of the project.	
API Gateway	Making deploy serverless functions accessible by the Qualee web app in the form of APIs communicated by REST HTTP requests.	
Texttract	Detecting and extracting any data in the form of English texts from PDF files in S3, the returned data will be stored in DynamoDB by various processes that will be mentioned in the following parts.	
Comprehend	Detecting the <i>entities</i> , the minimal compound English phrases that can convey a concrete meaning from a long text (the <i>Texttract</i> outcomes in our case), and labelling each entity as <i>PERSON</i> , <i>LOCATION</i> , <i>ORGANIZATION</i> , <i>COMMERCIAL_ITEM</i> , <i>EVENT</i> , <i>DATE</i> , <i>QUANTITY</i> , <i>TITLE</i> or <i>OTHER</i> .	
S3	Storing raw PDF file without any modifications uploaded from user local computer via Qualee web app. Each file is accompanied with its uploaded date and time. This datetime will be updated each	2 buckets:

	<p>time user uploads a new version for it. Deleting a document is allowed. The document can (only) be downloaded by matching IAM credentials.</p> <p>Versioning is <u>not</u> turned ON for storing documents.</p> <p>Storing static website's files (HTML, JavaScript, CSS, etc.) in order to deploy the website to end users.</p>	<ul style="list-style-type: none"> - One for document uploads - One for static website
Cloud Formation	<p>Supporting uploading and deploying Lambda functions.</p> <p><i>*This service is not activated by user's manual manipulations on AWS Console but is accompanied automatically by default when using AWS Lambda and/or API Gateway.</i></p>	
Elasticsearch Service	<p>Hosting Elasticsearch server to serve the searching purpose of the project.</p> <p><i>*This is only the hosting platform for an Elasticsearch server, not the server itself. Developers still need to build the Elasticsearch server as usual manners.</i></p>	1 domain
Console	Accessing and monitoring services that developers are using.	
CloudWatch	Keeping track on logging info (errors, debugging output, etc.) of services that are implemented in the project.	

For the two S3 buckets, go into the ***Bucket Policies*** section and add the following settings to allow file access and loading from the end user:

■ In the ***Permission*** section, turn OFF all settings related to *blocking public access* on the S3 bucket:

Block public access (bucket settings)

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to prevent public access to your buckets and objects, you must turn off public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying this setting, you must first ensure that you have no public access to your buckets or objects within, you can customize the individual settings below to suit your specific needs.

Block all public access

Off

- Block public access to buckets and objects granted through *new* access control lists (ACLs)
Off
- Block public access to buckets and objects granted through *any* access control lists (ACLs)
Off
- Block public access to buckets and objects granted through *new* public bucket or access point policies
Off
- Block public and cross-account access to buckets and objects through *any* public bucket or access point policies
Off

■ Next, in the **Bucket Policies** section, set a policy in JSON format as follows:

Bucket policy editor ARN: arn:aws:s3:::qualee-document-data
 Type to add a new policy or edit an existing policy in the text area below.

```

1 {
2   "Version": "2012-10-17",
3   "Id": "Policy1546414473940",
4   "Statement": [
5     {
6       "Sid": "Stmt1546414471931",
7       "Effect": "Allow",
8       "Principal": {
9         "AWS": "arn:aws:iam::XXXXXXXXXX:user/iam_username"
10      },
11      "Action": [
12        "s3:GetObject",
13        "s3:GetObjectAcl",
14        "s3:PutObject",
15        "s3:PutObjectAcl"
16      ],
17      "Resource": "arn:aws:s3:::qualee-document-data/*"
18    }
19  ]
20 }
```

In line 9: in the Principal section, after “**AWS:**”, fulfil with the ARN of the assigned credentials of the IAM account set up in the backend settings as mentioned above.

In line 17: replace with the ARN of the S3 Bucket you are working on.

■ Then click **Save** to save the policy which is just created.

Perform the same actions as above with the *bucket of deploying the static website* of the project.

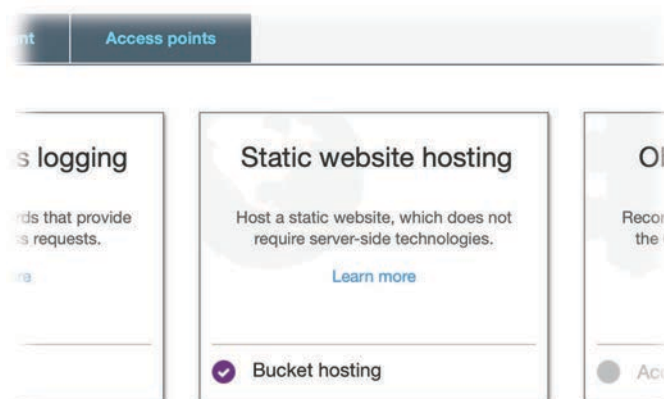
However, its Bucket Policy needs to be more open than a bucket used to store documents, as on the right:

```

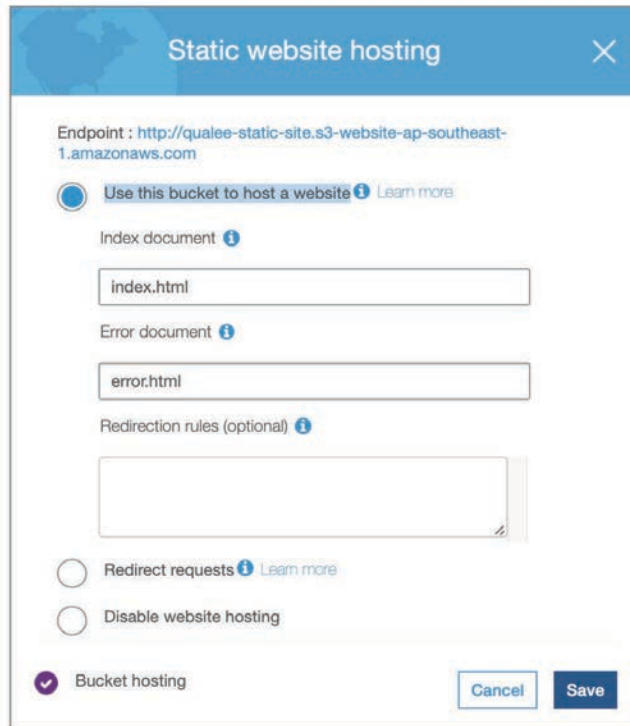
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Sid": "PublicReadGetObject",
6       "Effect": "Allow",
7       "Principal": {
8         "AWS": "*"
9       },
10      "Action": "s3:GetObject",
11      "Resource": "arn:aws:s3:::qualee-static-site/*"
12    }
13  ]
14 }
```

Also, in the bucket of static website:

■ Go to **Properties**, click on the **Static website hosting** section:



■ Click the **Static Website Hosting** tab, select **Use this bucket to host a website**, then click the **Save** button:



2. Configuration of the Project's Backend

The backend of this project is made by Python with Boto3 library which is a famous library of AWS/Serverless for Python 3.x. It provides various APIs for manipulation many services of AWS via Python code.

However, this Python project is set inside an outer Node.js project which makes the *Serverless offline* debugging easier and eases the deployment process from Serverless project to AWS Lambda/API Gateway.

In **config_es.py** file, fulfil the **host**, **region**, **aws_access_key_id** and **aws_secret_access_key** with appropriate credentials of IAM user, as illustrated below:

```
# FULFILL WITH OWNER IAM (OR ROOT USER) CREDENTIALS:
host = "https://xxxxxxx" # For example, my-test-domain.us-east-1.es.amazonaws.com
region = "us-west-xx" # e.g. us-west-1
service = "es"
credentials = boto3.Session(
    aws_access_key_id="XXXXXXXXXXXXXXXXXXXX", # ACCESS KEY
    aws_secret_access_key="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX", # SECRET KEY
).get_credentials()
```

Next, the name of *two indexes*, in which the data serving for searching is organised, must be also set; each of these two names will be concatenated to host name to establish full URL of Elasticsearch:

```
# NAME TWO INDEXES OF ELASTICSEARCH:
data_index = "data" # index for saving Textract and Comprehend outputs --> use for SEARCHING
keywords_index = "keywords" # index for saving suggestions (only Comprehend outputs) --> use for SUGGESTING
```

In **app.py**, the main file of the backend, fulfil with appropriate credentials of IAM user, as illustrated below:

```
# FULFILL WITH OWNER IAM (OR ROOT USER) CREDENTIALS:
tableName = "tablename" # DYNAMO DB'S TABLE NAME TO STORE ALL DATA TO
accessKey = "XXXXXXXXXXXXXXXXXXXX"
secretKey = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
s3BucketName = "qualee-document-bucket" # DYNAMO DB'S TABLE NAME TO STORE ALL DATA TO
region = "us-west-xx" # THE REGION IN WHICH THE AWS SERVICE IS REGISTERED
```

Finally, in `query_textract_cron.py`:

```
s3BucketName = "qualee-document-bucket"
region = "ap-southeast-xx"
tableName = "extract_results"
accessKey = "XXXXXXXXXXXXXXXXXXXX"
secretKey = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
```

3. Configuration of the project's frontend

The frontend of this project is implemented by **React** in **Next.js** framework. Its main job is to create a *static website* to deploy to an AWS S3 bucket; From this, it uses the *static website hosting* feature of AWS S3 to distribute this website to public users.

In the **pages/index.js** file, between lines 21 and 29, an AWS credentials must be set up to access AWS resources or services. Before that, however, the developer must ensure that the AWS SDK for JavaScript is installed into *node_modules* and present in the project's *package.js*.

The *region*, *accessKeyID*, *secretAccessKey*, and *bucketname* variables illustrated below must be set to the correct, specific, and correct way with the AWS identity that the developer is using:

```
var region = 'us-west-xx';
var s3 = new AWS.S3({
  accessKeyId: 'XXXXXXXXXXXXXXXXXXXX',
  secretAccessKey: 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX',
  region: region,
});
var bucketname = "qualee-document-bucket"; // Bucket of saving documents
```

It should also be mentioned that these identity settings must be exactly correct as it will directly affect the access URLs to the project services and/or resources. Here are two variables that will use the identities mentioned above to access the URLs, if the developer wants to ensure the accuracy of these URLs, they can hard code instead of appending the existing variables, as exemplified below:

```
export default function Home() {  
  const bucketURLPrefix = `https://${bucketname}.s3-${region}.amazonaws.com/`; // BUCKET URL  
  const hostURL = `https://f71nk1z64h.execute-api.${region}.amazonaws.com`; // HOST URL OF ALL APIS
```

Note that the **hostURL** variable is the same prefix between all backend endpoints that have been previously deployed to AWS.

II. PROJECT IMPLEMENTATION ON AWS

The implementation of this entire project to AWS will go through 3 main steps:

- Deploying the backend system to AWS Lambda and API Gateway
- Deploy the frontend system to AWS S3 as a static website.
- Set a cronjob in AWS Lambda Events for **/dev/runtextextract** to run every 60 seconds.

1. Deploying the backend system to AWS Lambda and API Gateway

In the source code of the backend, ensure the issues are covered below before deploying to AWS.

In the **serverless.yml** file, between lines 16 and 21, the developer needs to make sure that the *region* and *profile* of the project deployer are set up correctly with the *access key* and *secret key* used in the project as mentioned in previous sections. Programmers must carefully refer to the **named profile** settings of AWS, which will be used to fill the profile section of the provider as shown below.

```
16 provider:
17   name: aws
18   runtime: python3.6
19   stage: dev
20   region: ap-southeast-1
21   profile: inapps
22
```

For more details of AWS named profiles on local machine, refer it via [this link](#).

Next, it is necessary to ensure that **serverless**, **pipenv**, and **docker** are installed on the developer's local computer. However, the use of **docker** will be automated by Serverless, programmer does not need any action on this platform. After that,

- Run the **pipenv shell** command to activate **pipenv**, then run the **pipenv install** to install all the Python dependencies associated with the project.
- Run the command **npm install** to install the NodeJS packages installed in the project.
- Run **sls deploy** command to start deploying the backend to AWS. If successful, this will return a list of API endpoints deployed to AWS.

After successfully deploying the backend to AWS, still on the command line screen in the pipenv shell, type **serverless info** to list the entire list of endpoints deployed to AWS which could be similar as below:

```

stack: qualee-serverless-flask-dev
resources: 110
api keys:
  None
endpoints:
  ANY - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev
  ANY - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/{proxy+}
  GET - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/hung
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/sendbinary
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/comprehend
  GET - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/listalldocs
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/textractfrompdf_startjob
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/textractfrompdf_trackjob
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/querytexttract
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/deletedoc
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/listallsearchdata
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/search
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/suggest
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/startdetecttextract
  GET - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/runtexttract
  POST - https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/dev/deleterelatedinfo
functions:
  app: qualee-serverless-flask-dev-app

```

Somehow, the programmer needs to remember this endpoint list in order to deploy it into the project's frontend because the frontend will need to call these APIs through HTTP requests.

2. Deploy the frontend system to AWS S3 as a static website

In the working directory of the frontend it is very essential first to run the `npm install` command to install all the project dependencies.

In the `packages.json` file, around on line 11, change the following values:

- "`qualee-static-site`" to the name of the bucket used to store the static website of the project.
- "`inapps`" to the name of a named profile of AWS on the deployer's local machine

```

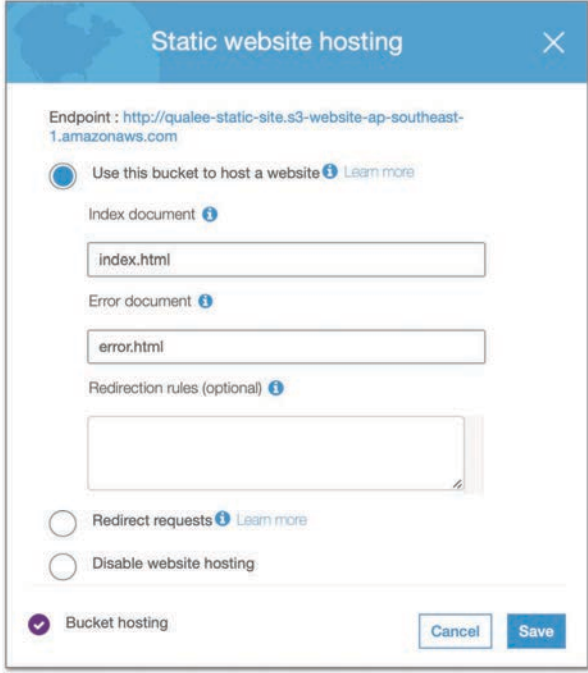
5   "scripts": {
6     "dev": "next dev -p 4000",
7     "build": "next build",
8     "start": "next start -p 4000",
9     "export": "next export",
10    "prepare": "next build && next export",
11    "upload": "aws s3 sync out s3://qualee-static-site --profile inapps"
12  },
13  "dependencies": {

```

Then, run the following commands:

- `npm run prepare`: used to initialise a set of static files for a pure HTML / CSS and JavaScript project. This set of files will be used to upload to S3.
- `npm run upload`: upload static website to S3.

When the process is complete, the developer opens the **S3 Console** and goes into the bucket that is set up for static web hosting. In the **Static website hosting** section mentioned earlier, click on the **Endpoint** URL:



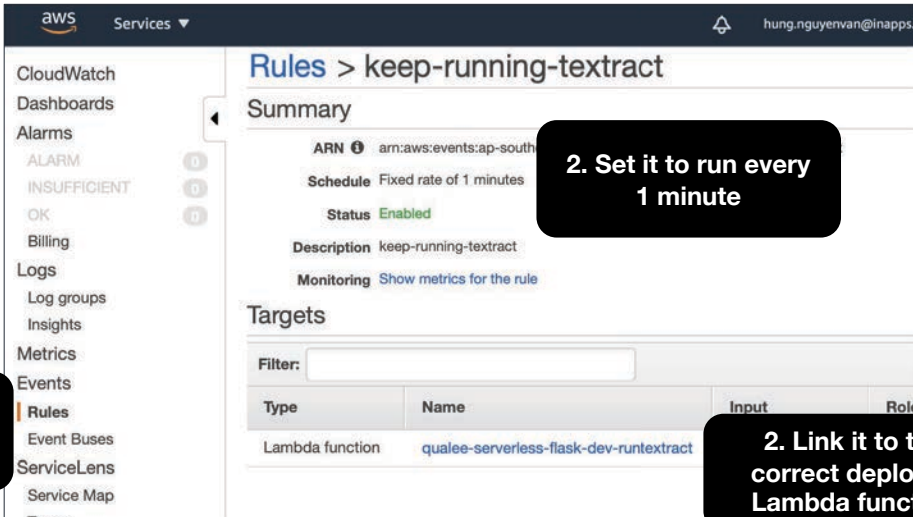
The browser will automatically open this website in a new tab.

3. Set a cronjob in AWS Lambda and CloudWatch Events to run every 60 seconds

Inside the deployed backend, there is **/dev/runtextextract** which must be set to run every 60 seconds to extract Textract and Comprehend data to inject into DynamoDB table. For each occurrence, the operations of this cronjob can be described as below:

- It will loop for all filename in S3, the length of the loop will be equal to the number of files in S3.
- For each loop, it will check if the file has its extraction data in DynamoDB or not. For more details, it checks if there is any record with its “*docname*” attribute in the table is equal to the filename or not:
 - If not, it will create a new record.
 - If yes, it will check if the “*data*” attribute has the element “*PENDING*”:
 - If yes, the Textract and Comprehend data need to be injected to this attribute, it means the entire process of Textract and Comprehend needs to be operated in order to gain the outputs.
 - If no, there is nothing to do with the record, the loop will go ahead to the next iteration’s item.

The setting up of cronjob in *AWS CloudWatch Events* is necessary, this setting can be summarised as the below instructions:



The screenshot shows the AWS CloudWatch console with the 'Rules' page selected. The rule 'keep-running-textract' is being configured. The left sidebar shows the navigation menu with 'Rules' highlighted under the 'Events' section. The main content area shows the 'Summary' and 'Targets' sections. The 'Summary' section displays the rule's ARN, schedule (Fixed rate of 1 minutes), status (Enabled), and description (keep-running-textract). The 'Targets' section shows a table with one target: 'Lambda function' with the name 'qualee-serverless-flask-dev-runtextextract'.

1. Create a new rule in Cloudwatch Events

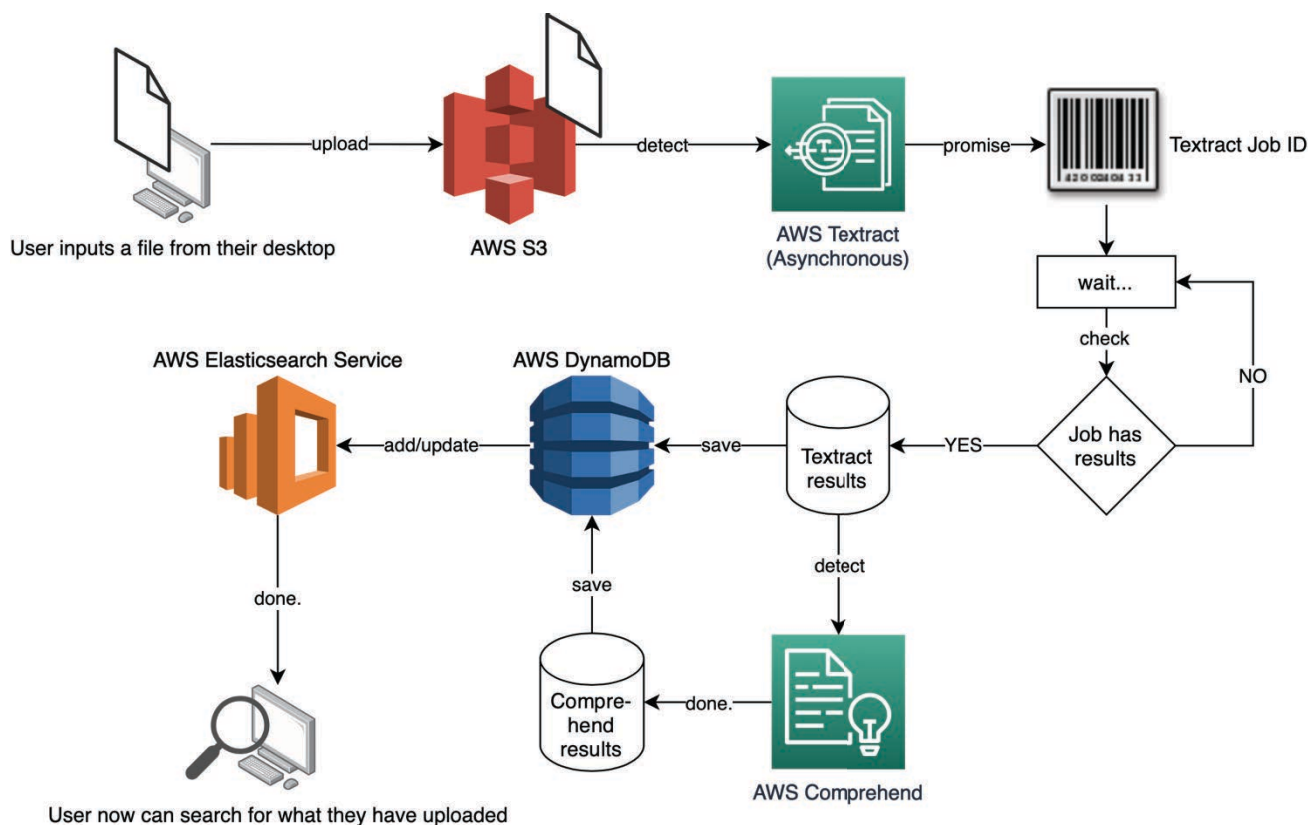
2. Set it to run every 1 minute

2. Link it to the correct deployed Lambda function

III. PROCESSING FILE IN THE BACKGROUND BY BACKEND

In general, each file after being uploaded will be processed via Textract and Comprehend to get as much as possible meaningful content. After that, the processed data will be organised and stored in a DynamoDB table, this actions helps to make sure a document to be processed with its Textract's job ID and to back up Textract data for future use, in order to prevent the fact that one document is processed multiple times by multiple job ids. In the meantime, data is also added to Elasticsearch database in order to return search results in the future.

The entire process can be summarised as the flow chart below:



1. Detecting and Extracting Written Texts From File By AWS Textract

After having done with uploading a new document, the entire process of extracting and getting meanings from file will be in the background that user is unable to interact with.

The first step of this process is to extract any written standard English texts from raw file. Textract is able to set processing language to other popular languages but this setting is not automatic but manual by developer's selection in their code. For example, if a French text is inputted under English Textract process, all accents and specific symbols of original language will be removed.

The process of Textract used in this project is **asynchronous**, which means the Textract results are not returned immediately when being requested. The asynchronous process will return a Job ID as an "invoice" for the promised job. After that, a *cronjob-like* mechanism is set in **AWS Lambda Events** to keep checking if the Job ID is already assigned the results or not, this mechanism is repeat every 60 seconds.

After each 60-second occurrence, *if the results are still not gained from Textract*, a one-element array including "PENDING" as a placeholder will be putted into DynamoDB table, as illustrated by the code below:

```
item = {
    "docname": documentName,
    "jobid": responseJobJD["JobId"],
    "datecreated": str(timestamp),
    "datemodified": timestamp,
    "data": ["PENDING"],
    "comprend": [],
}
try:
    table.put_item(
        Item=item,
        ConditionExpression="attribute_not_exists(docname)",
    )
except:
    return {
        "statusCode": 200,
        "body": "Item is already existing, no need to put item.",
        "headers": headers,
    }
```

If the result is gained from Textract, which means Textract detection was successful, it will be updated to DynamoDB by its matching Job ID (each job ID is one object in DynamoDB) to replace the placeholder "PENDING" mentioned above. The sample code to get the text detection by Textract is below, in which the variable **textList** is holding each "line" of text that Textract has detected:

```
response = textract.get_document_text_detection(JobId=jobitem["jobid"])
if response["JobStatus"] == "SUCCEEDED":
    for item in response["Blocks"]:
        if item["BlockType"] == "LINE":
            print("\n" + item["Text"])
            textList.append(item["Text"])
```

For example, the raw input is like this:



The Textract data to save to DynamoDB will look like this (in green):

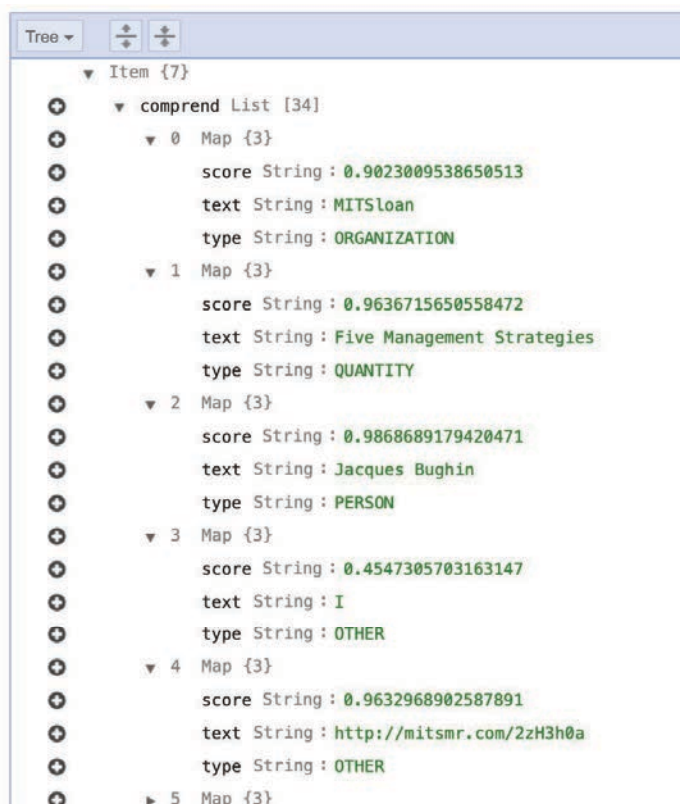
```
▼ data List [172]
0 String: MITSloan
1 String: Management Review
2 String: Five Management
3 String: Strategies for
4 String: Setting the Most
5 String: From AI
6 String: A global survey of C-level executives finds that AI is
7 String: delivering real value to companies that use it across
8 String: operations and within their core functions.
9 String: Jacques Bughin
```

After Textract result set is saved into database, backend keeps using this result for Comprehend process.

2. Getting English Meaning of Word/Phrase Entities by AWS Comprehend

It is obvious to see that the Textract data saved in DynamoDB table is a list/array. The *Python Boto3* backend re-concatenate all the elements in this array into a continuous text on which Comprehend will run the detection to output the word entities.

AWS Comprehend will analyse, predict and give **scores** to guess what category the item belongs to in Comprehend's available genres (types). Comprehend then selects the highest scoring judgment to make a prediction which has such the score become the final result. Comprehend's results, then, are also saved to DynamoDB, for example as the illustration on the right:



By default, AWS Comprehend only supports some main entities, including *PERSON*, *LOCATION*, *ORGANIZATION*, *COMMERCIAL_ITEM*, *EVENT*, *DATE*, *QUANTITY*, *TITLE* or *OTHER*. However, developer can build **custom entities** by injecting additional data to Comprehend to perform *NLP-machine learning* to make more accurate predictions in the context that the user provides.

3. Management of Extraction Outputs by AWS DynamoDB

After the operations of Comprehend and Textract completed, all outputs of these two processes will be injected into the DynamoDB database. DynamoDB is a NoSQL data system that allows storage in various different forms, not limited only by columns and rows. That is the reason that the output data of Comprehend and Textract is saved as *key-value* or *array* as shown above, making it easier to organise and retrieve data.

The structure of a record in DynamoDB can be illustrated as follows:

```

  ▼ Item {7}
  + ▶ comprehend List [34]
  + ▶ data List [172]
  + datecreated String : 1601902387864
  + datemodified Number : 1601902387864
  + docname String : Five Management Strategies (1).pdf
  + jobid String : a18522707ddd1cac1fc9426c3b6e4c90a3a5eb66352de6b47910a7fd13894886
  + size Number : 7872

```

Attribute	Data type	Explanation
comprehend	List of Map	<p>Each element in the list is a Map, which has:</p> <ul style="list-style-type: none"> - score: the score by which the prediction is selected. - text: the raw text of the labelled entity - type: the predicted label of the entity <pre> ▼ comprehend List [34] ▼ 0 Map {3} score String : 0.9023009538650513 text String : MITSloan type String : ORGANIZATION ▶ 1 Map {3} ▶ 2 Map {3} </pre>
data	List of String	<p>reflexes a LINE in Textract result.</p> <pre> ▼ data List [172] 0 String : MITSloan 1 String : Management Review 2 String : Five Management 3 String : Strategies for 4 String : Setting the Most </pre>
datecreated	String	The timestamp of the date on which the file is uploaded to S3
datemodified	Number	The timestamp of the date on which the file is uploaded to S3. This field is temporarily not used in the project for any purpose.
docname	String	The document name uploaded on S3
jobid	String	The Job ID of the latest Textract job that generated the results of text detection.
size	Number	<p>The size (in bytes) of the entire extracted text, the file content, from Textract.</p> <p><i>*This is not the file size of the uploaded document.</i></p>

4. Adding Data to Elasticsearch Service

The project's Elasticsearch database contains two indexes: one for saving Textract and Comprehend data, one for saving keywords for suggesting functionality. These two indexes are reflexed in Python backend with the two following functions.

Function #1: Uploading Textract and Comprehend outputs to Elasticsearch

```
def uploadToElastics(data, comprehend, docname):
    indexName = "data"
    timestamp = int(time.time() * 1000)
    path = indexName + "/_doc/" + str(timestamp)
    url = host + path

    # The JSON body to accompany the request (if necessary)
    payload = {
        "user": "hungnguy",
        "timestamp": timestamp,
        "body": data,
        "comprehend": comprehend,
        "title": docname,
    }

    resp = requests.put(
        url, auth=awsauth, json=payload
    ) # requests.get, post, and delete have similar syntax

    print(resp.text)
    return resp.text
```

The function ***uploadToElastics*** serves to uploading Textract and Comprehend outputs for the purpose of searching by end user. This function takes three obligatory params:

- ***data***: the *array* of Textract results
- ***comprehend***: the *array of key-value objects* of Comprehend results.
- ***docname***: the original filename, in *String*, of the file from which Textract and Comprehend gained outputs.

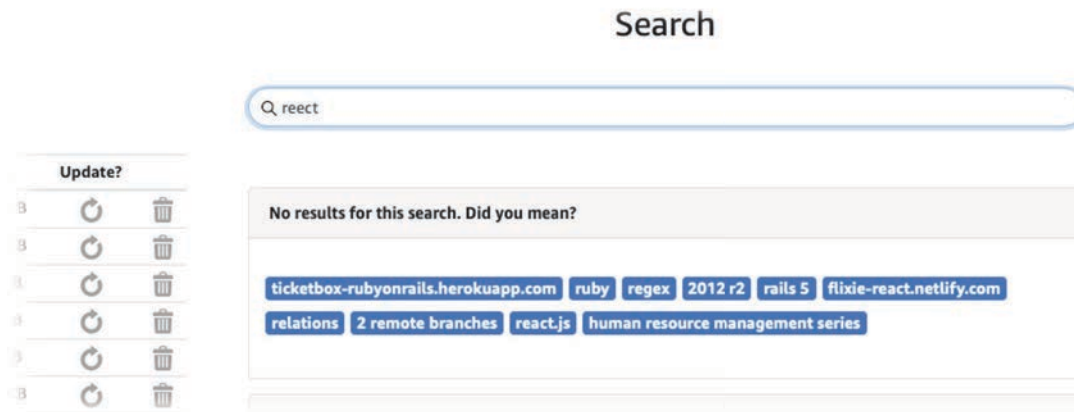
The payload, which signifies one object to be uploaded to Elasticsearch, should have ***user*** field. This could be the username of the server owner or the end user in case the web app would have authenticating feature of login/signup. The ***awsauth*** variable is a key-value object which includes *access key*, *secret key*, *region*, *service name* (in this case it is “es”) and the *session token* which was generated in the step of setting up the project in part 1.

```
awsauth = AWS4Auth(
    credentials.access_key,
    credentials.secret_key,
    region,
    service,
    session_token=credentials.token,
)
```

Function #2: Preparing data for suggestions

Suggestion feature is developed in this Qualee project in order to help users find out the exact keywords in case they did not memorise the correct form of the search term.

For example, when people search for keyword “*reect*”, this search is unable to result any output because the word “*reect*” cannot be found in any uploaded document since it is a mis-spelling mistake when user types it in the searching textbox. In this case, the suggestion feature will scan the Elastics database and guess the nearest correct form of this word which could be “*react*” or “*react.js*” as the following figure:



This function can be illustrated as the following figure:

The function ***uploadKeywordsToElastics*** serves to uploading Comprehend outputs for the purpose of suggestion search keywords for end user. This function takes three obligatory params:

Add Debug Configuration

```
def uploadKeywordToElastics(data: str, type: str, fromDoc: str):
    timestamp = int(time.time() * 1000)
    prepared_keyword = "_".join(data.split()).lower()
    path = "keywords/_doc/" + prepared_keyword
    url = host + path
    payload = {
        "user": "hungnguy",
        "timestamp": timestamp,
        "body": data,
        "type": type,
        "from": fromDoc,
    }

    r1 = requests.put(url, auth=awsauth, json=payload)
    return r1.text
```

- ***data***: Comprehend entity’s raw text, in *String*.
- ***type***: the entity’s type that Comprehend predicted, in *String*.
- ***fromDoc***: the original filename, in *String*, of the file from which Comprehend gained outputs, in *String*.

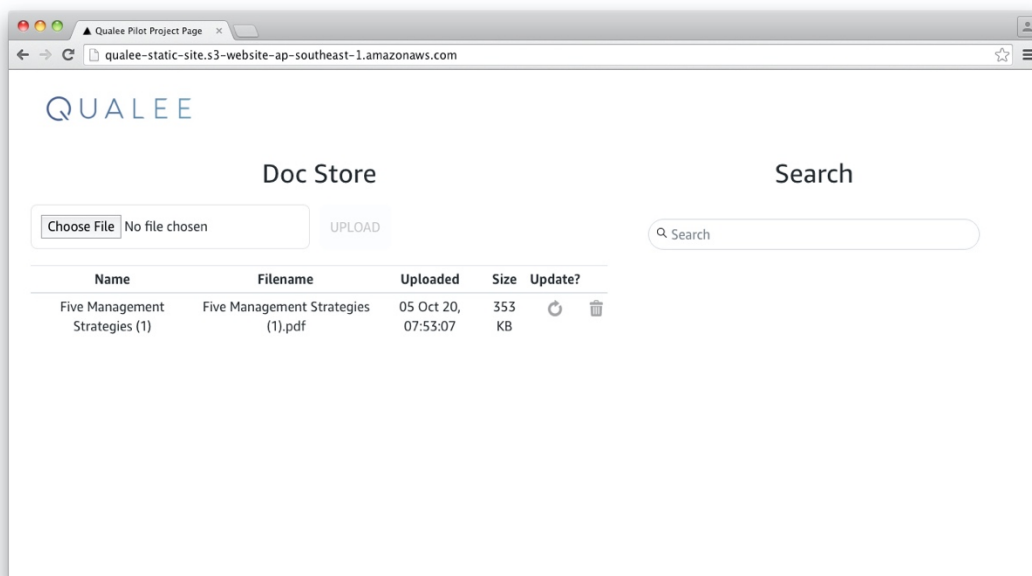
IV. USAGES

1. Access Web App

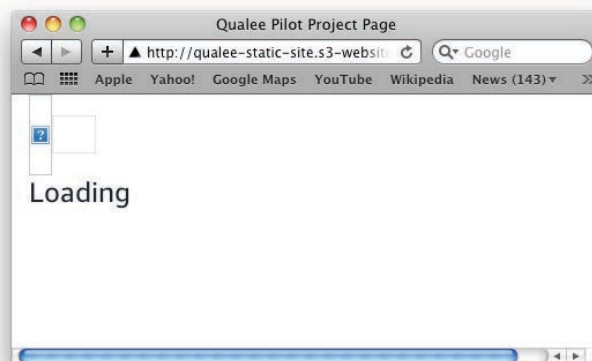
Access to <http://qualee-static-site.s3-website-ap-southeast-1.amazonaws.com>.

2. Uploading A New Document

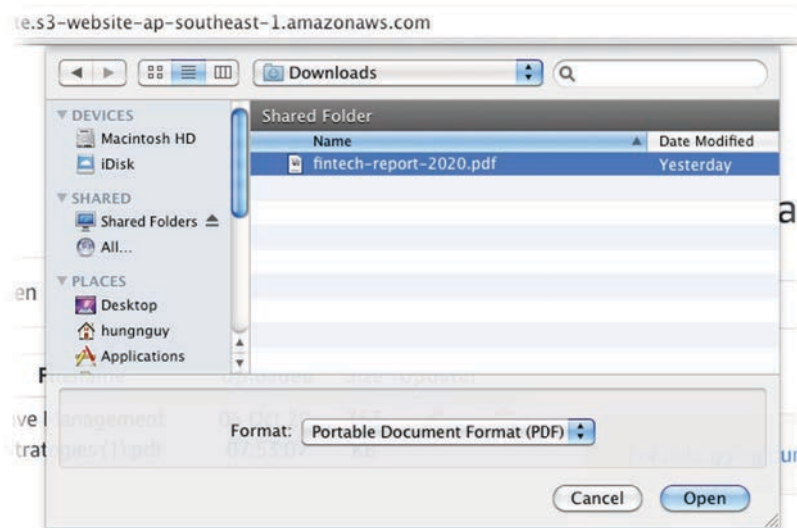
“Loading” text will appear on the web interface while the web app is preparing data for its operations. Main user interface of the web app will be loaded after a short while:



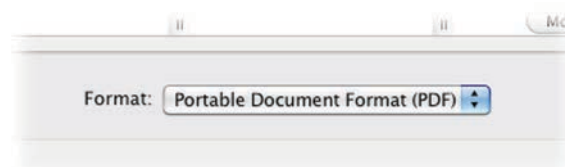
If the “Loading” text keeps appearing on the web browser so long, consider using another browser or that the system is not compatible with HTML5/CSS3:



When pressing **Choose File**, a dialog appears in order to pick a file for uploading:



The default format for uploading is **PDF**; if it is not chosen automatically, user can set it in **Format** dropdown box:

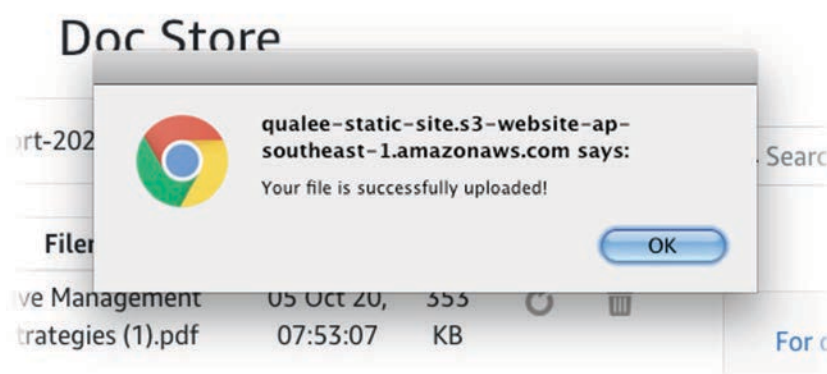


The project is intentionally configured to work on PDF format.

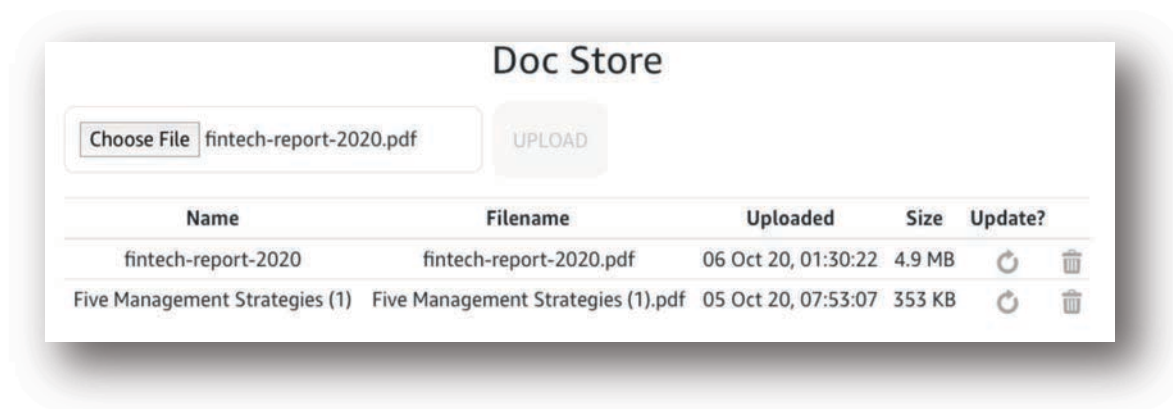
User must keep using one file format for the entire life of the file. If user chooses to upload with a file format other than the original/first version's one, it will crash all Textract and Comprehend processes in the future. For example, if user upload a **PDF** file to S3, the next version(s) of that file must be always in **PDF**.

After having chosen a file, user can click on **Upload** to start uploading.

The file will be uploaded to S3 storage, browser will inform user when the file is uploaded:



The file list will be updated to reflect the newly uploaded file, it will be usually shown on the top of the list. User can take a look on the Uploaded (date) and the file size of each file:



Immediately after uploading a file, the processing procedure will be triggered automatically and in the background. User must wait at least 1 minute to be able to search for any content in the latest uploaded file. This process is described in the following part.

3. Frontend Process of Uploading a File

The frontend engine receives file from the user through a multipart form and only allows uploading 1 file per time. This form then uses a function named ***newFileUpload()***, which takes one param as a file from form's ***e.target***, to initiate the S3 upload process through the AWS JavaScript SDK. The figure below is the main part of this function:

```
// START UPLOADING A FILE
var params = {
  Bucket: bucketname, Key: filePath, Body: file, ACL: 'public-read'
};

s3.upload(params, async (err, data) => {
  if (err) {
    console.log(err);
    alert('There was an error happening, please try again later.');
```

```
  } else {
    console.log(data);
    const startTextextract_resp = await axios.post(startTextextractJob, {
      "docname": data.key,
    });
    console.log(startTextextract_resp);
    alert('Your file is successfully uploaded!');
    getAllDocs();
    setFile(null);
  }
}).on('httpUploadProgress', function (progress) {
  var uploaded = parseInt((progress.loaded * 100) / progress.total);
  setProgressUploading(uploaded);
});
```

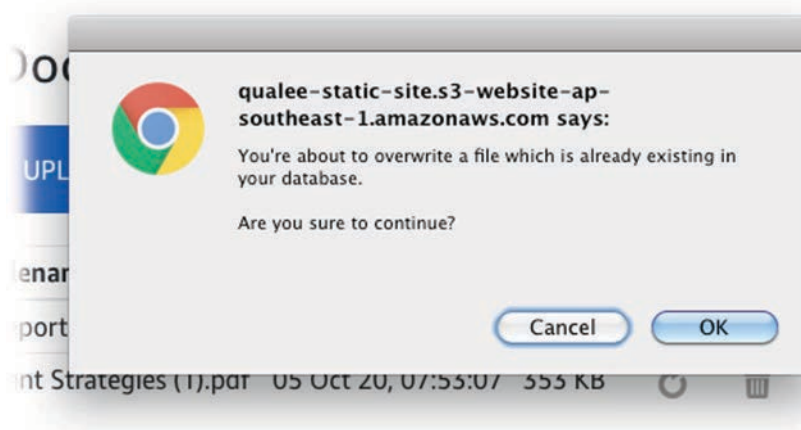
With the new file uploaded on S3, which means the file list is appended with a new element, frontend needs to refresh the file list by re-calling the endpoint `/dev/listalldocs` again.

After that, it requests to start processing with Textract through calling an API of `/dev/startdetecttextract` to request to start *in the background* the process including Textract, Comprehend and Elasticsearch, all of this takes about 1 ~ 10 minutes depending on the length and complexity of the document.

4. Uploading a File with treating duplicates

The process of uploading a file is described as the above part; however, if S3 detects that the **filename** (no matter what the file content is) is already existing in S3 bucket, the website will prompt user that they still want to continue or not, as illustrated below:

- If **yes** (clicking on OK), the old file will be overwritten with the one being uploaded.
- If **no** (clicking on Cancel), the file system on S3 keeps unchanged.



By pressing **OK**, user must take into account at their own risk that they will have no way to reverse to the old version before the overwrite. The new file, once uploaded, will be treated a “brand new” file.

The overwrite process will also *delete all* Textract, Comprehend and Elasticsearch data that related to the old version, then these data will be re-initialised, which means the file will go through Textract, Comprehend and Elasticsearch process again until meaningful data is gained from the file.

The code of checking duplication can be illustrated in the code below, in which, **doclist** is a *state* variable in React which holds the file list:

```

if (docslst.filter(function (e) { return e.docname === fileName; }).length > 0) {
  objectIsExisting = true;
  if (confirm("You're about to overwrite a file which is already existing in your databa
    deleteDocument(fileName, "0");
  } else { return; }
}

```

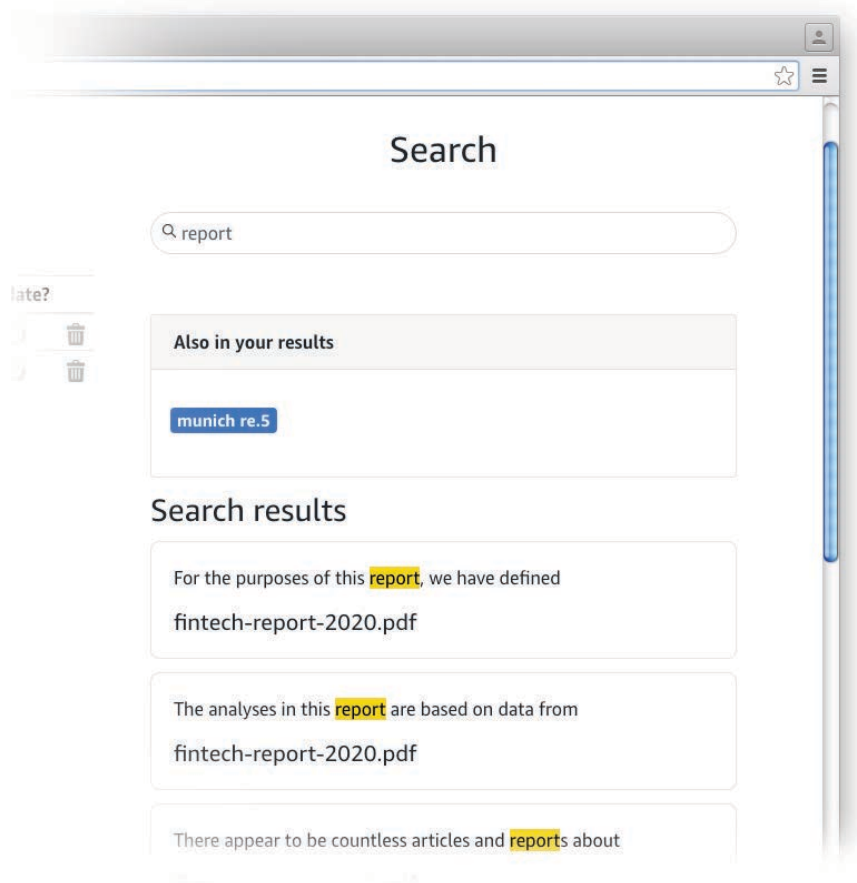
After clicking OK, the uploading process will happen in the normal way then user will see the **uploaded date** updated, they may also see the **file size** changed after uploading.

This duplication treating process is also the mechanism of *updating an existing file* which will be mentioned in the next parts.

5. Searching A Term

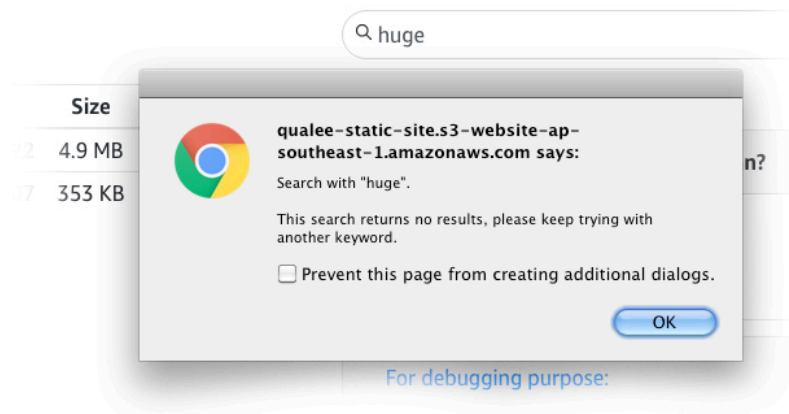
5.1 Searching a term, in general

User needs to input the term that they want to search on searching textbox in the web UI, after second(s), the search results will be shown in “Search results” list on the right of the main interface, some suggestions will also appear in “Also in your results” box based on the resemblance of the search term with some nearest neighbours in Elasticsearch’s “keywords” index:

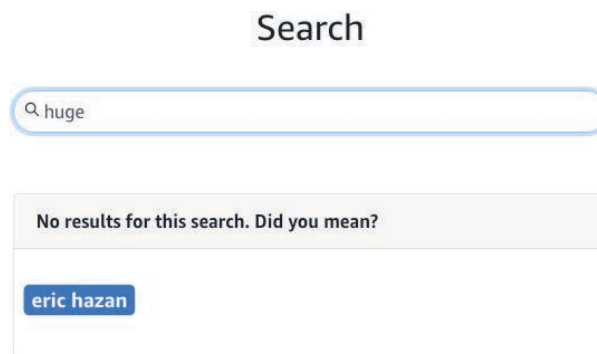


If the web app cannot find any matching result from Elasticsearch, it will show a dialog informing that the result is not found, as the following figure:

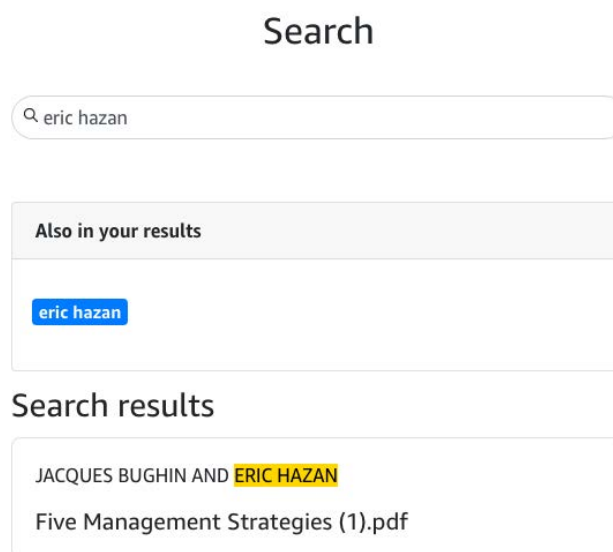
Search



In addition, for non-searchable term/keywords, the web app shows some suggestions thanks to suggestion API implemented in the Python backend:



When clicking on a suggestion, the web app will start to search with its text and list a new results list after the new search:



5.2 Backend process of searching a term

In general, when user presses **Enter** from keyboard on the search textbox, the frontend starts the process for searching with the search term, it will call the backend API **/dev/search**. After a short while, the API will return search result as an *Elastic search result object* from which the frontend will start treating data to make it into a JavaScript *list of objects* that can be shown on web UI.

By searching request is fired by the frontend, it will trigger the API “search” in backend, this search endpoint calls a core function **startsearch()** which takes a *query* string. This function returns search results from Elasticsearch domain’s URL by making GET request based on query type on Elastic mechanism:



```

Add Debug Configuration
def search(event, context):
    print("Coming to search(event, context)")
    query = json.loads(event["body"])
    query_string = query.get("term")
    data = startsearch(query_string + "*")
    # print(data)
    # toreturn = dict(data).get("hits").get("hits")
    return {
        "statusCode": 200,
        # "body": json.dumps(data),
        "body": data,
        "headers": headers,
    }

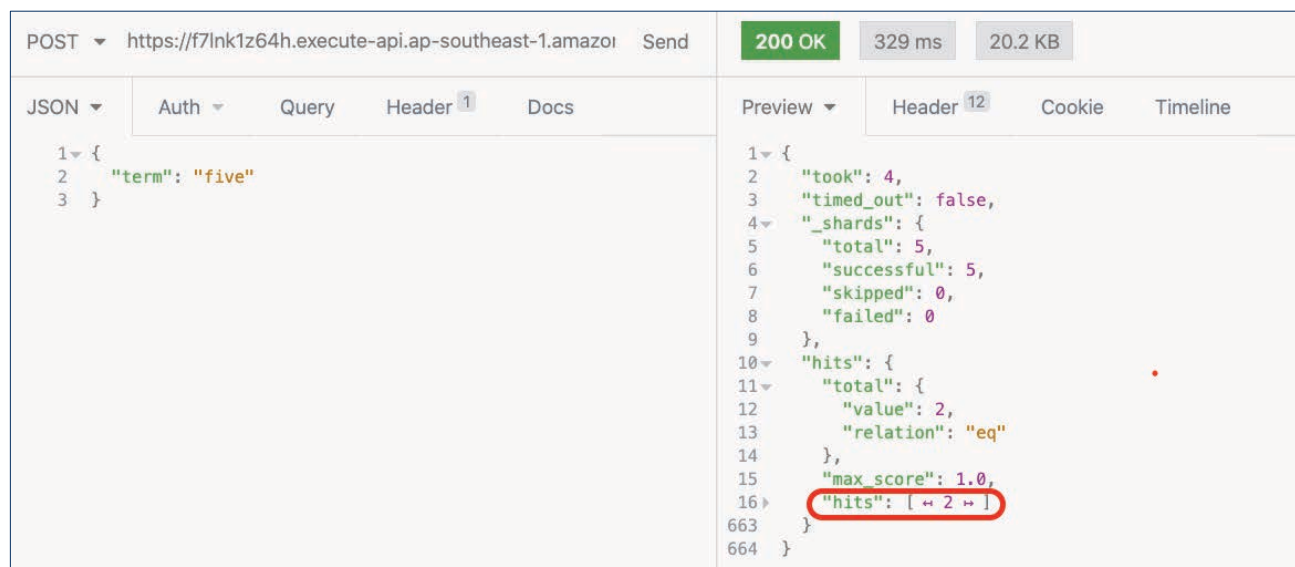
```

```

Add Debug Configuration
def startsearch(query: str):
    print("Query in progress: " + query)
    query = {
        "query": {
            "query_string": {
                "fields": ["body", "title"],
                "query": query,
                "analyze_wildcard": True,
            }
        }
    }
    print(url)
    r2 = requests.get(
        url,
        auth=awsauth,
        data=json.dumps(query),
        headers={"Content-Type": "application/json"},
    )
    # print(r2.text)
    return r2.text

```

The Elastic search result object which is returned by backend can look like this:



```

POST https://f71nk1z64h.execute-api.ap-southeast-1.amazonaws.com/ Send
200 OK 329 ms 20.2 KB

JSON Auth Query Header 1 Docs
1 {
2   "term": "five"
3 }

Preview Header 12 Cookie Timeline
1 {
2   "took": 4,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": {
12      "value": 2,
13      "relation": "eq"
14    },
15    "max_score": 1.0,
16    "hits": [ 2 ]
17  }
18 }

```

In the search results of term “five” above, the backend generates two objects in “hits” (the inner “hits”), this is the basis on which the frontend will treat/process to establish **a list** of search results in order to print to the screen for the user.


6. Updating existing file

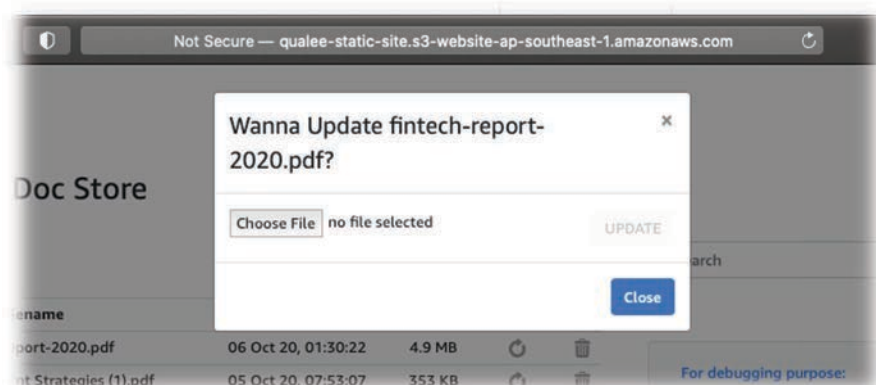
The process of updating an existing file will go through these treatments:

- Uploading new document to AWS S3 with exactly the same filename of the original file. *S3 has a built-in feature to overwrite a file with new version while the old one will be deleted completely.
- Deleting all Textract and Comprehend data that was stored in DynamoDB and Elastics database which are related to the filename being updated, for more details:
 - o The data record in DynamoDB which has such the filename in “*docname*” field will be removed completely.
 - o The Textract and Comprehend data in searching index of Elasticsearch which has such the filename in “*docname*” field will be deleted completely.
 - o The Comprehend data in *keyword* index of Elasticsearch which has such the filename in “*docname*” field will be deleted completely.
- Triggering the **/dev/startdetecttextract** to re-initialise the entire in-background processing procedure with the new file content.

User has to take into account a fact that they must keep track on the file content at their own risk since the filename is not allowed to be changed during the entire life of the file until user deletes it (by pressing delete button/icon on the web UI).

6.1. Calling file update in frontend:

When pressing on update button with icon , a dialog will appear and require user to input a file from their local so that they can feel free to choose a file with which the old one will be updated:



In the frontend's programming code, the Next.js frontend uploads a new file to S3 to upload a file as normal as mentioned in the previous parts thanks to AWS SDK for JavaScript. After that, it continues to call the API `/dev/deleterelatedinfo` to delete related information that previously accompanied to the file so that the newly uploaded one is considered a brand-new one.

The updated file will be shown on top of the file list in web UI with new uploaded date as well as, maybe, the file size.

6.2. Delete related information in backend

Deleting related information of the file will go through two main steps: delete its data from DynamoDB (with `table.delete_item()` function), then delete its data from Elasticsearch database (with `deleteFromElastics()` function).

The `deleteFromElastics()` function, of course, will delete data from both *searching index* and *keyword index*.

The screenshot of code below depicts the main parts of these two actions:

```
## DELETE FROM DYNAMO DB:
try:
    response_delete_db = table.delete_item(
        Key={"datecreated": searchInDynamoDB(documentname)},
        ConditionExpression="docname = :val",
        ExpressionAttributeValues={":val": documentname},
    )
    print(response_delete_db)
    list_of_deletion.append("Done deleting DYNAMO DB")
except ClientError as e:
    print(e)

## DELETE FROM ES :
try:
    response_delete_es = deleteFromElastics(documentname)
    print(response_delete_es)
    list_of_deletion.append("Done deleting ES")
except ClientError as e:
    print(e)
```



```
def deleteFromElastics(docname):
    # FROM DATA ES:
    pathMovies = data_index + "/_delete_by_query"
    url = host + pathMovies
    query1 = {"query": {"match": {"title": docname}}}
    r1 = requests.post(
        url,
        auth=awsauth,
        json=query1,
        headers={"Content-Type": "application/json"},
    )
    print(r1.text)

    # FROM KEYWORDS ES:
    pathKeywords = "keywords/_delete_by_query"
    url = host + pathKeywords
    query2 = {"query": {"match": {"from": docname}}}
    r2 = requests.post(
        url,
        auth=awsauth,
        json=query2,
        headers={"Content-Type": "application/json"},
    )
    print(r2.text)
    return r2.text
```

6.3. Reincarnation of the file at the end of updating process

The file, after being updated in S3 and deleted its data from DynamoDB and Elasticsearch, will go through the process of Textract and Comprehend to be searchable by users at the end.

7. Deleting a file

The process of deleting a file will go through these deletions:

- The data record in DynamoDB which has such the filename in “docname” field will be removed completely.
*The key “docname” is not primary key in this project’s DynamoDB. This process of deletion in DynamoDB requires it as a clue to return the primary key, the “datecreated”, then use this primary one via **dynamo_table.scan()** to delete the exact record, as the following figure via sub function **searchInDynamoDB()**:*

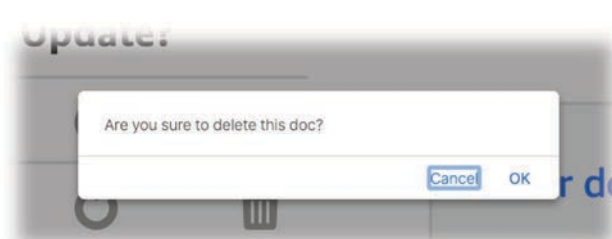
```

Add Debug Configuration
def searchInDynamoDB(docname):
    print("Coming to searchInDynamoDB")
    response = table.scan(
        FilterExpression=Attr("docname").eq(docname),
    )
    print(response["Items"])
    if len(response["Items"]) > 0:
        return response["Items"][0]["datecreated"]
    else:
        return 0
  
```

- The Textract and Comprehend data in searching index of Elasticsearch which has such the filename in “docname” field will be deleted completely.
- The Comprehend data in *keyword* index of Elasticsearch which has such the filename in “docname” field will be deleted completely.

7.1. Calling file deletion in frontend

When pressing on update button with icon , a dialog will appear to confirm that the user really wants to delete their file:



7.2. File deletion in backend

Deleting a file will go through three main steps:

- delete the file itself in S3
- delete its data from DynamoDB (with ***table.delete_item()*** function),
- delete its data from Elasticsearch database (with ***deleteFromElastics()*** function).

The ***deleteFromElastics()*** function, of course, will delete data from both searching index and keyword index.

The screenshot of code below depicts the main parts of these two actions:

```
## DELETE FROM S3:
try:
    response_deleteS3 = s3.delete_object(Bucket=s3BucketName, Key=documentname)
    print(response_deleteS3)
    list_of_deletion.append("Done deleting from S3")
except ClientError as e:
    print(e)

## DELETE FROM DYNAMO DB:
try:
    response_delete_db = table.delete_item(
        Key={"datecreated": searchInDynamoDB(documentname)},
        ConditionExpression="docname = :val",
        ExpressionAttributeValues={" :val": documentname},
    )
    print(response_delete_db)
    list_of_deletion.append("Done deleting DYNAMO DB")
except ClientError as e:
    print(e)

## DELETE FROM ES :
try:
    response_delete_es = deleteFromElastics(documentname)
    print(response_delete_es)
    list_of_deletion.append("Done deleting ES")
except ClientError as e:
    print(e)
```