Image Classifier Engine Webservice Project

Developer Manual

Introduction

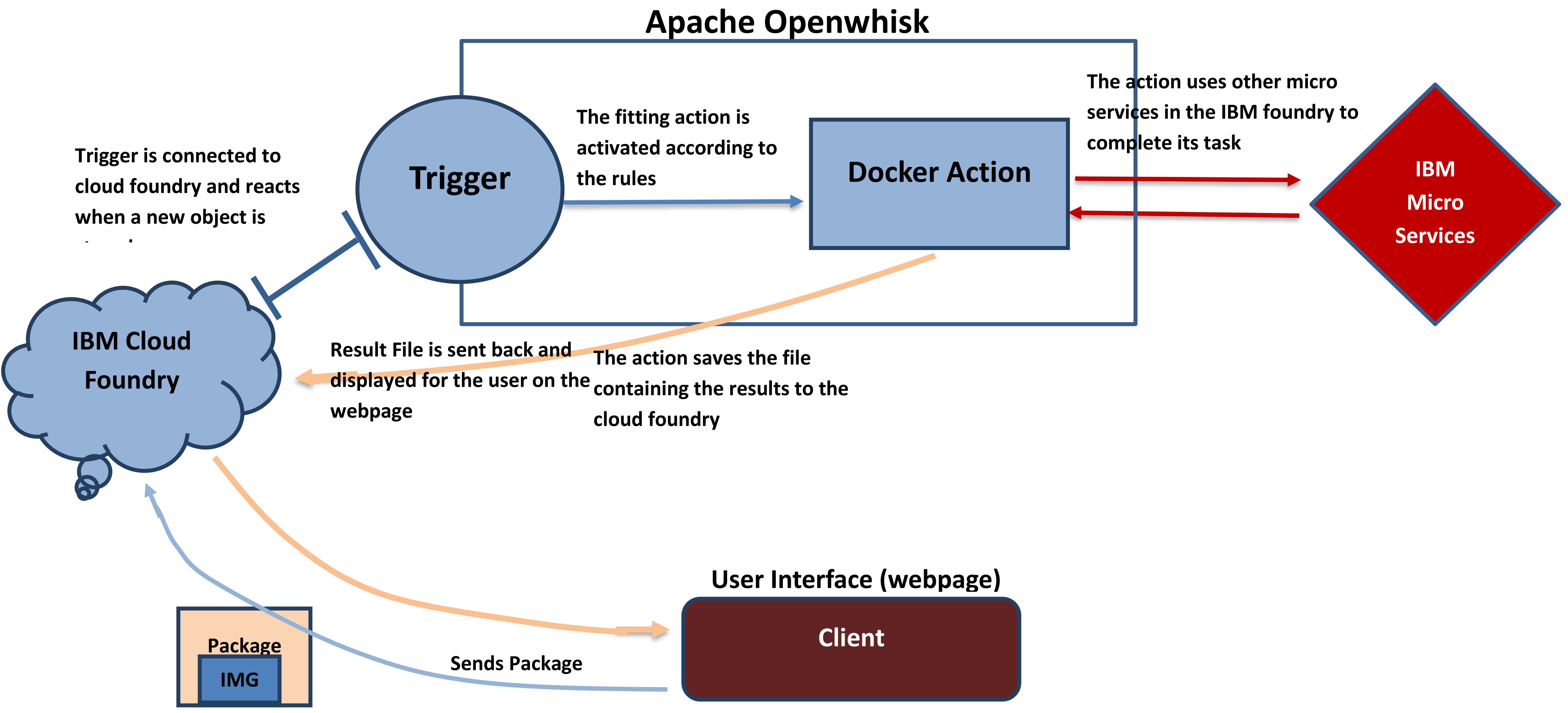
The project in general is an implementation of an image classifier, that acts like an engine, it receives an image, and returns the object presented in the image received.

In this manual, a thorough explanation will be presented to explain the different components that build up the project.

It will explain step by step every component on its own, and how it interacts with other components from the project.

We will start at first with general introduction to the architecture used, moving on to the backend side of the project and to finish with the frontend side, and how they connect with each other

Logical Architecture


Apache Openwhisk
Trigger is connected to cloud foundry and reacts when a new object is
Trigger
The fitting action is activated according to the rules
Docker Action
The action uses other micro services in the IBM foundry to complete its task
IBM Micro Services
IBM Cloud Foundry
Result File is sent back and displayed for the user on the webpage
The action saves the file containing the results to the cloud foundry
User Interface (webpage)
Client
Package
IMG
Sends Package

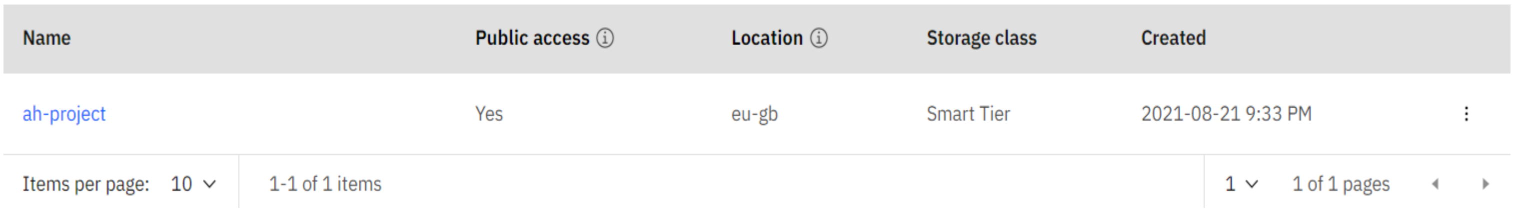
Backend side of the project

IBM Cloud Entities

IBM Cloud Foundry – Cloud Object Storage

We created a cloud object storage called **testData** to store image files in, and results. This object holds several properties which we will use later to connect to other components.

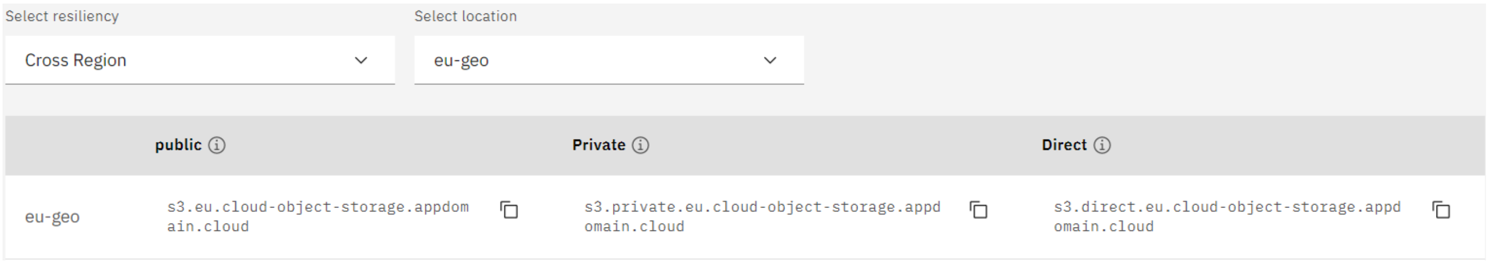
- **Bucket:**



Name	Public access	Location	Storage class	Created
ah-project	Yes	eu-gb	Smart Tier	2021-08-21 9:33 PM

Items per page: 10 1-1 of 1 items 1 1 of 1 pages

- **Endpoints:**

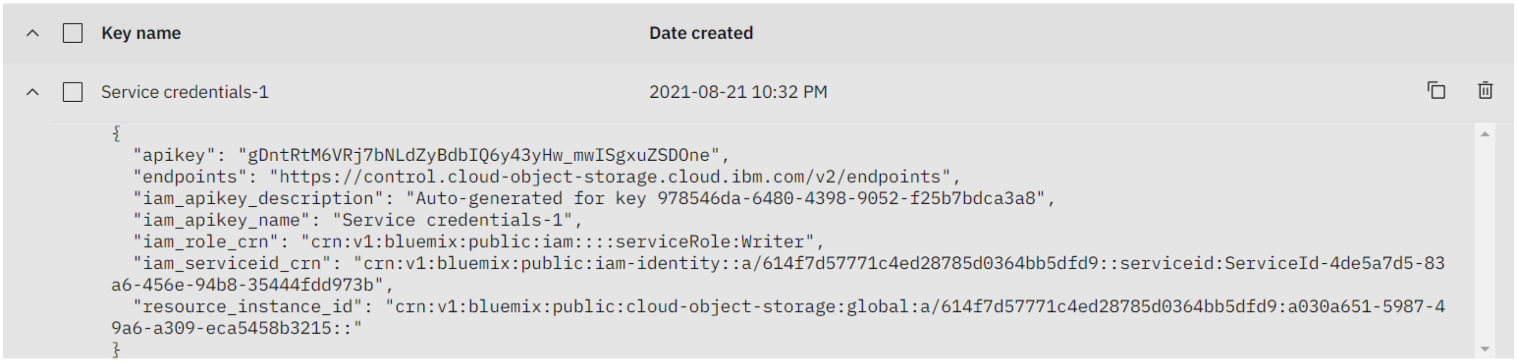


Select resiliency: Cross Region

Select location: eu-geo

	public	Private	Direct
eu-geo	s3.eu.cloud-object-storage.appdomain.cloud	s3.private.eu.cloud-object-storage.appdomain.cloud	s3.direct.eu.cloud-object-storage.appdomain.cloud

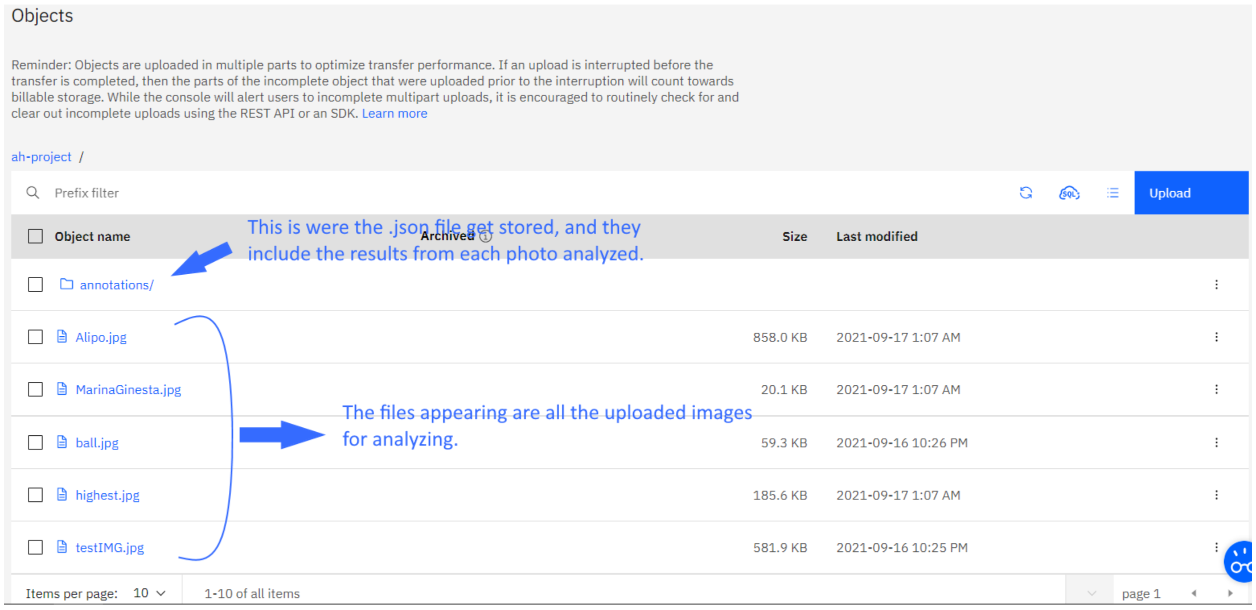
- **Service Credentials:**



Key name	Date created
Service credentials-1	2021-08-21 10:32 PM

```
{
  "apikey": "gDntRtM6VRj7bNLdZyBdbIQ6y43yHw_mwISgxuZSDOne",
  "endpoints": "https://control.cloud-object-storage.cloud.ibm.com/v2/endpoints",
  "iam_apikey_description": "Auto-generated for key 978546da-6480-4398-9052-f25b7bdca3a8",
  "iam_apikey_name": "Service credentials-1",
  "iam_role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Writer",
  "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity::a/614f7d57771c4ed28785d0364bb5dfd9::serviceid:ServiceId-4de5a7d5-83
a6-456e-94b8-35444fdd973b",
  "resource_instance_id": "crn:v1:bluemix:public:cloud-object-storage:global:a/614f7d57771c4ed28785d0364bb5dfd9:a030a651-5987-4
9a6-a309-eca5458b3215::"
}
```

Now you can find the actual data base in the bucket ah-project, which will hold the objects we need, as follows:


Objects

Reminder: Objects are uploaded in multiple parts to optimize transfer performance. If an upload is interrupted before the transfer is completed, then the parts of the incomplete object that were uploaded prior to the interruption will count towards billable storage. While the console will alert users to incomplete multipart uploads, it is encouraged to routinely check for and clear out incomplete uploads using the REST API or an SDK. Learn more

ah-project /

Object name	Archived	Size	Last modified
annotations/			
Alipo.jpg		858.0 KB	2021-09-17 1:07 AM
MarinaGinesta.jpg		20.1 KB	2021-09-17 1:07 AM
ball.jpg		59.3 KB	2021-09-16 10:26 PM
highest.jpg		185.6 KB	2021-09-17 1:07 AM
testIMG.jpg		581.9 KB	2021-09-16 10:25 PM

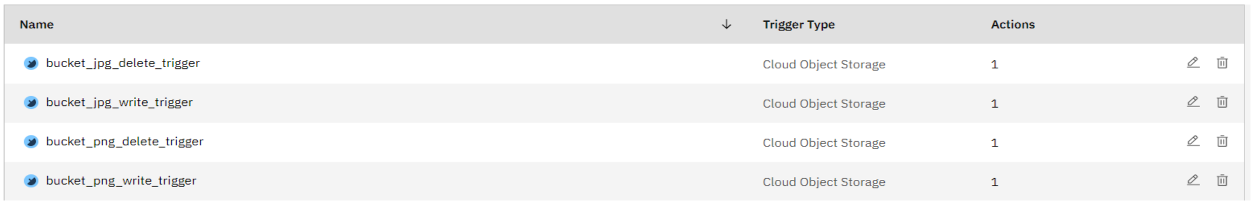
Items per page: 10   1-10 of all items   page 1

This is were the .json file get stored, and they include the results from each photo analyzed.

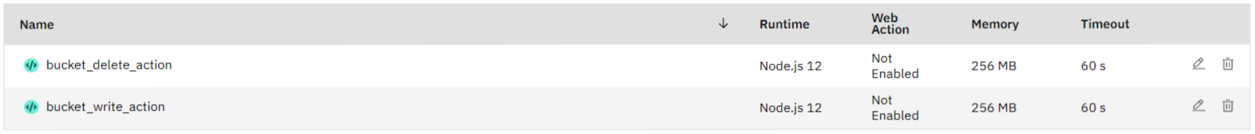
The files appearing are all the uploaded images for analyzing.

Actions & Triggers:

We created 4 triggers that are all connected to the testData cloud storage:


Name	Trigger Type	Actions
bucket_jpg_delete_trigger	Cloud Object Storage	1
bucket_jpg_write_trigger	Cloud Object Storage	1
bucket_png_delete_trigger	Cloud Object Storage	1
bucket_png_write_trigger	Cloud Object Storage	1

and 2 actions:


Name	Runtime	Web Action	Memory	Timeout
bucket_delete_action	Node.js 12	Not Enabled	256 MB	60 s
bucket_write_action	Node.js 12	Not Enabled	256 MB	60 s

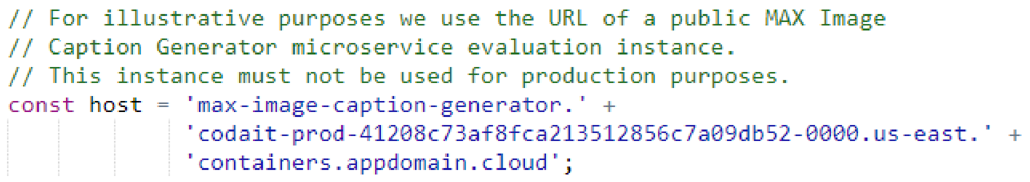
The triggers bucker_jpg_write_trigger and bucket_png_write_trigger both react when a new object is stored in the ah-project data base that resides in testData.

The triggers bucker_jpg_delete_trigger and bucket_png_delete_trigger both react when a delete object request arrive at the testData object storage.

Moving on to the actions, we have two actions as we mentioned already, one responsible for analyzing the image that gets stored in the ah-project data base, and one for deletion.

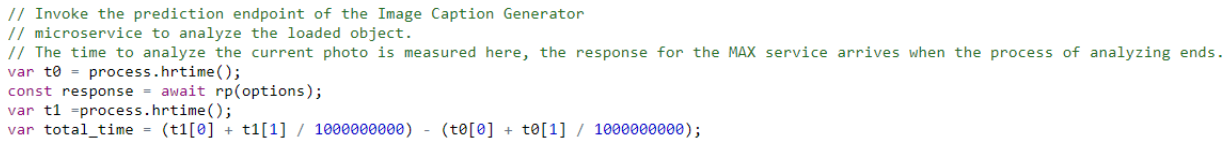
Bucket_write_action: This one is activated when a new object is stored in the database, and its job is to analyze the object that arrived and storing a new object in the database in a folder called **Annotation**, which hold **json** type files that contain the results from analyzed photos.

The code inside the action itself is well documented for a developer to understand, but the main point to look at is the max-image-caption-generator, which is a microservice found in the IBM foundry apps, which is build on OpenWhisk open source, and it uses means of AI and deep learning to analyze a photo. In the action code this microservice is used to analyze the current photo and to return a response for it, we use this response and add to the end of it the time of calculation and save all this in a file of type json in the Annotation folder inside the database, and that file is used later to show results to the user.



```
// For illustrative purposes we use the URL of a public MAX Image
// Caption Generator microservice evaluation instance.
// This instance must not be used for production purposes.
const host = 'max-image-caption-generator.' +
             'codait-prod-41208c73af8fca213512856c7a09db52-0000.us-east.' +
             'containers.appdomain.cloud';
```

Here in the photo you can see that the host is given the max-image-caption-generator, which later we use this host to send a photo and to get results back.



```
// Invoke the prediction endpoint of the Image Caption Generator
// microservice to analyze the loaded object.
// The time to analyze the current photo is measured here, the response for the MAX service arrives when the process of analyzing ends.
var t0 = process.hrtime();
const response = await rp(options);
var t1 =process.hrtime();
var total_time = (t1[0] + t1[1] / 1000000000) - (t0[0] + t0[1] / 1000000000);
```

And here we invoke the image caption generator and wait for response, and in that time we measure the time it take for it to finish.

```
// generate annotations file key from the object's key
const key_id = `annotations/${path.parse(args['key']).name}.json`;

step = 'save annotation in COS bucket';
var obj = JSON.parse(response)['predictions'];
obj.push({"index":"3","caption":"Time to analyze (seconds).","probability":total_time});
```

And here we create the json file which contains the results, and we add it to the annotation folder in the database for it to get used later.

Bucket_delete_action: this action is responsible for deleting the json file from the Annotation folder after a photo has been deleted from the database, so it gets rid of the json file that includes the results from that photo.

```
// Remove annotation from Cloud Object Storage
await cos.deleteObject({
                        Bucket: args['bucket'],
                        Key: `annotations/${path.parse(args['key']).name}.json`
                       }).promise()
```

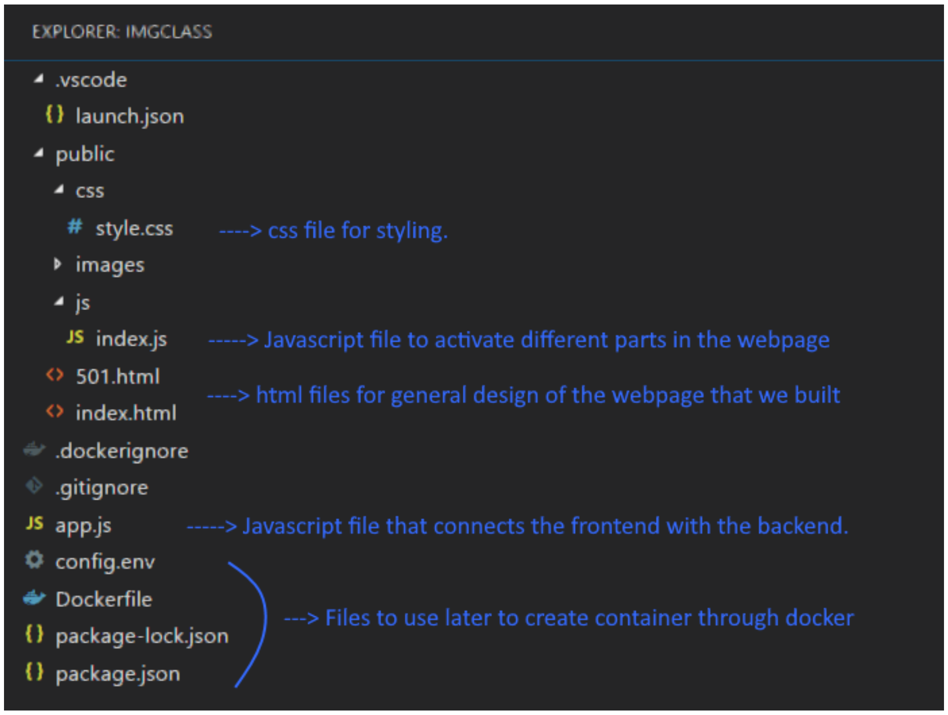
after we get all the information that we need about the file we are deleting, then we delete the json file from the object storage.

That concludes the backend side of the project. Explaining all the sized used in the IBM cloud foundry and storage, and the usage of microservices.

Client – Webservice (Frontend side of the Project)

This is section explains the UI we built, and the code it includes to connect to the backend side.

Initial look at the files:

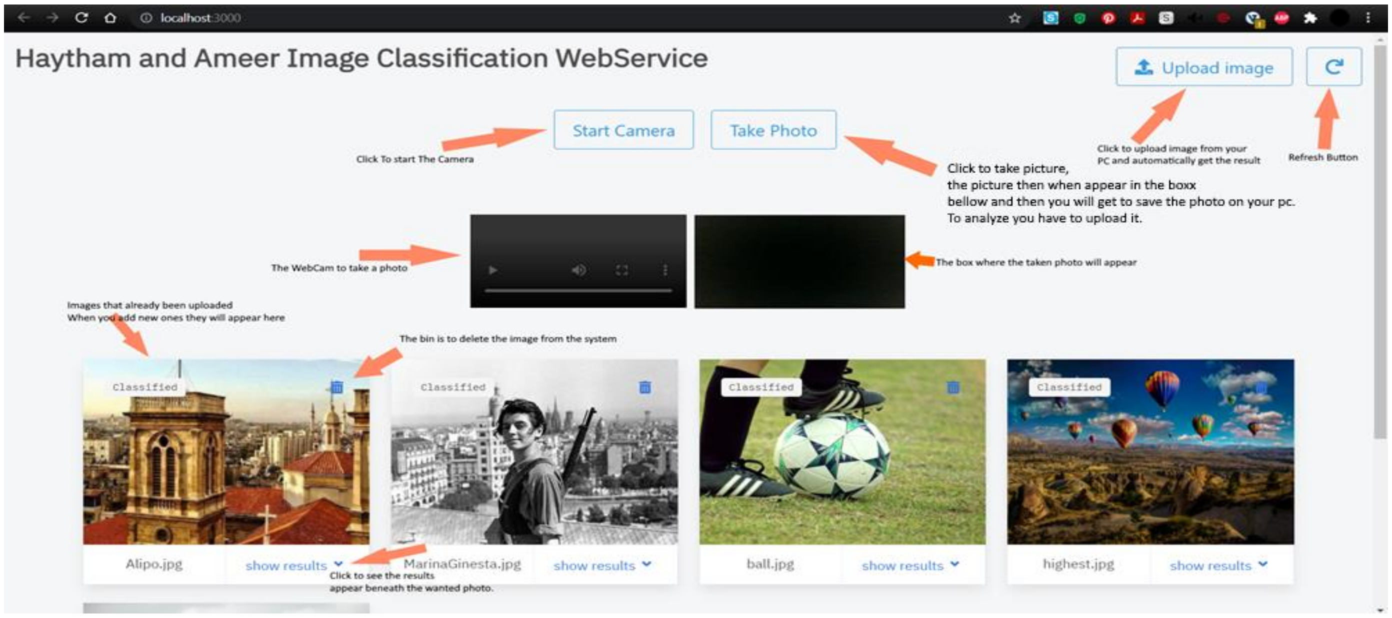


```
EXPLORER: IMGCLASS
◢ .vscode
   {} launch.json
◢ public
   ◢ css
      # style.css        ----> css file for styling.
   ▸ images
   ◢ js
      JS index.js        -----> Javascript file to activate different parts in the webpage
   <> 501.html
   <> index.html         ----> html files for general design of the webpage that we built
.dockerignore
.gitignore
JS app.js               -----> Javascript file that connects the frontend with the backend.
config.env
Dockerfile
{} package-lock.json    ---> Files to use later to create container through docker
{} package.json
```

Now the most important components are the index.html, index.js, app.js and the docker files.

General Design of the Webservice:

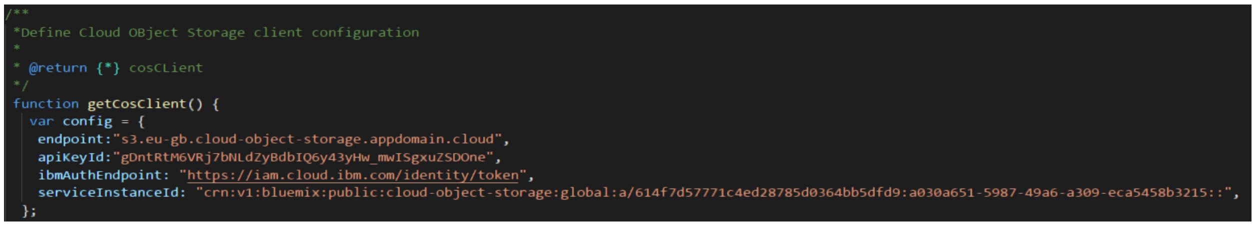
Which is represented in the index.html file. The general view of the webpage is:


localhost:3000
Haytham and Ameer Image Classification WebService
Upload image
Start Camera
Take Photo
Click To start The Camera
Click to upload image from your PC and automatically get the result
Refresh Button
Click to take picture, the picture then when appear in the boxx bellow and then you will get to save the photo on your pc. To analyze you have to upload it.
The WebCam to take a photo
The box where the taken photo will appear
Images that already been uploaded When you add new ones they will appear here
The bin is to delete the image from the system
Classified
Classified
Classified
Classified
Alipo.jpg
show results
Click to see the results appear beneath the wanted photo.
MarinaGinesta.jpg
show results
ball.jpg
show results
highest.jpg
show results

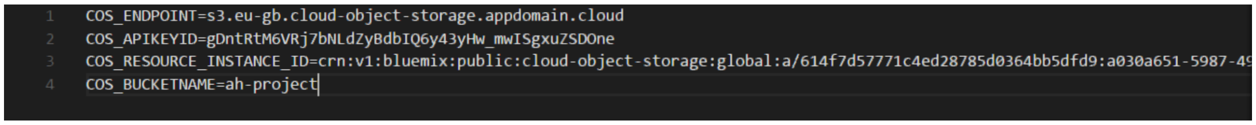
Connection with the backend side – app.js:

In this file general connection with the cloud storage is made, it includes a few functions which gets an object that represents the storage, and functions such as sending a file the storage or making a removing request from it.

In this file we use the credentials we talked about earlier that belong to the cloud storage, for instance:


```
/**
 *Define Cloud OBject Storage client configuration
 *
 * @return {*} cosCLient
 */
function getCosClient() {
  var config = {
   endpoint:"s3.eu-gb.cloud-object-storage.appdomain.cloud",
   apiKeyId:"gDntRtM6VRj7bNLdZyBdbIQ6y43yHw_mwISgxuZSDOne",
   ibmAuthEndpoint: "https://iam.cloud.ibm.com/identity/token",
   serviceInstanceId: "crn:v1:bluemix:public:cloud-object-storage:global:a/614f7d57771c4ed28785d0364bb5dfd9:a030a651-5987-49a6-a309-eca5458b3215::",
 };
```

This is also connected to the config.env file:


```
COS_ENDPOINT=s3.eu-gb.cloud-object-storage.appdomain.cloud
COS_APIKEYID=gDntRtM6VRj7bNLdZyBdbIQ6y43yHw_mwISgxuZSDOne
COS_RESOURCE_INSTANCE_ID=crn:v1:bluemix:public:cloud-object-storage:global:a/614f7d57771c4ed28785d0364bb5dfd9:a030a651-5987-4
COS_BUCKETNAME=ah-project
```

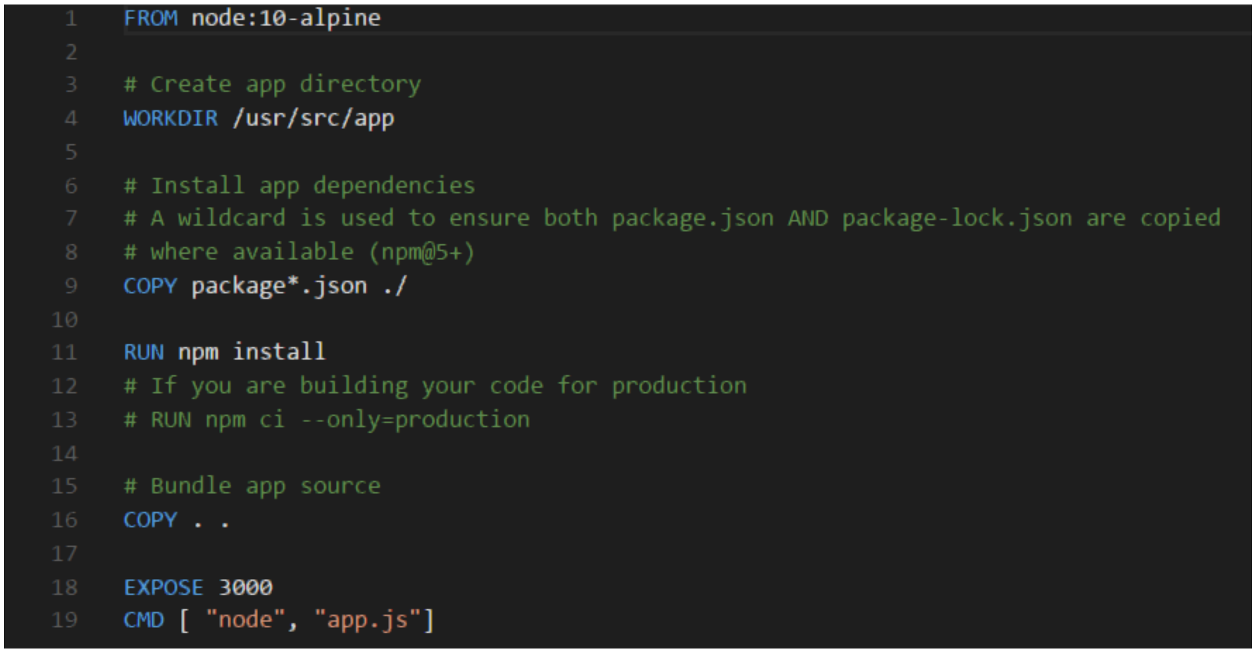
Behind the scenes – index.js:

This file is responsible for behind the scenes operations that happen behind the html layer, functions that operate buttons, starting the camera and taking the picture, and dynamically changing the view of the web based on current operations.

Docker Files – to fit the whole thing into container:

The main file between them is the Dockerfile, the other 2 include dependencies and such.

Dockerfile contains the commands to create docker container when using the docker commands.



```
FROM node:10-alpine

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 3000
CMD [ "node", "app.js"]
```

It is important to notice that this file specifies that the container will listen through the port 3000. That is important when running the image after building to establish a connection between the localhost and the container.

At the end of all this – Docker:

So to enable the service to run on any computer, we had to make the service into container, and we used docker for that purpose.

To create this container, run the cmd from the project file, and perform the following commands:

- Docker build -t imgclass .
  The imgclass will be the name of image created, that we will run later.
- Docker run -dp 3000:3000 --name imgclass imgclass
  The container will be named imgclass as well and will be ready to run.
  Notice that we specified that the container will listen at port 3000, actually both the container and the localhost will be using port 3000.

That concludes the developer manual. Any further information about specifications in the code itself will be found in the code as documentation. There you will be able to understand things more deeply and for you to change things to fit your own goals.