Implement IBM Cloud Function Endpoints

Estimated time needed: 90 minutes

Congratulations! You are one step closer to finishing the capstone project. You created a web application and added user authentication using the Django framework in the previous modules. Additionally, you set up continuous integration so when you commit code to your repo, it automatically kicks off a new build of your web application. Take a pause and pat yourself on the back. However, we are not done yet! In this module, you are asked to create the backend services for your application.

The Django application will talk to the database using backend services hosted on IBM Cloud. You are asked to create these services in Python and JavaScript. You will write these services on the IBM Cloud Functions serverless platform. Finally, you will add an API gateway in front of the functions.

1. Load data into the database

You have been provided the dealership data as a JSON file `./data/dealerships.json`. Your first task is to upload this data into your Cloudant database. There are multiple ways to do this programmitcally. You will need the your service credentials.

> NOTE: Kindly ensure that your Cloudant Instance is created using the **IAM authentication** method only and not the IAM and Legacy Authentication method. To check the authentication method of your Cloudant instance, click Manage:

1. Navigate to the resources page - https://cloud.ibm.com/resources.
2. Click on the Cloudant service. If you don't have one already, create one here - https://cloud.ibm.com/catalog/services/cloudant.
3. Click on `Service credentials` on the left bar.
4. Click `New credential`. You can leave the default options on the popup.
5. Expand the newly created credentials and take note of the `apikey` and the `url`.
6. Before you import the data, create two databases using the Cloudant UI:

- Click `Launch Dashboard` on the Cloudant UI and then Click `Create Database`
- Create one database called `dealerships` with `Non-partitioned`
- Create another database called `reviews` with `Non-partitioned`

Then in Theia terminal, you can use the npm package `couchimport` to load data into your database.
Note you may need to install `couchimport` first:

```
1
```
```
npm install -g couchimport
```
Copied!

7. Set the following environment variables using the credentials in the newly created Service credentials:

```
1
```
```
export IAM_API_KEY="REPLACED IT WITH GENERATED `apikey`"
```
Copied!

```
1
```
```
export COUCH_URL="REPLACED IT WITH GENERATED `url`"
```
Copied!

These two environment variables will be used for the following `couchimport` commands

8. In your forked Github repo folder, change into the `data` folder

```
1
```
```
cd cloudant/data
```
Copied!

and use the following command to import the JSON data into the database.

- First import dealerships data into `dealerships` database

```
1
```
```
cat ./dealerships.json | couchimport --type "json" --jsonpath "dealerships.*" --database dealerships
```
Copied!

9. Do the same for existing reviews.

```
1
```
```
cat ./reviews.json | couchimport --type "json" --jsonpath "reviews.*" --database reviews
```
Copied!

You should now have two databases, `dealerships` and `reviews` with sample data in each database.

2. Working with Cloudant in IBM Cloud Functions

You are asked to use IBM Cloud Functions serverless platform to create backend services that the Django application will make use of. The `agfzb-CloudAppDevelopment_Capstone/functions/sample` folder has sample functions for both JavaScript and Python that you can use as basis for your code. We have provided the bare minimum error handling. All the files assume that you have bound the following properties to the individual functions or the package itself:

```
1
2
3
4
5
```
```
{
    "COUCH_URL": "",
    "IAM_API_KEY": "",
    "COUCH_USERNAME": ""
}
```
Copied!

Let's look at each function in more detail.

1. Sample Python function

   The code uses the python-cloudant SDK to communicate with the IBM Cloudant service. If you are developing locally, you need to `pip install` the package using the provided `requirements.txt` file. The python runtime on IBM Cloud functions provides this package and you don't have to install anything.

   The `agfzb-CloudAppDevelopment_Capstone/functions/sample/python/main.py` file contains the function that returns the names of all databases. The following code authenticates with IBM Cloudant service using the username and api key.

   ```
   1
   2
   3
   4
   5
   ```
   ```
   client = Cloudant.iam(
               account_name=dict["COUCH_USERNAME"],
               api_key=dict["IAM_API_KEY"],
               connect=True,
           )
   ```
   Copied!

   The following code retrieves all the databases:

   ```
   1
   ```
   ```
   return {"dbs": client.all_dbs()}
   ```
   Copied!

   The code needs to return a Python dictionary for it to be a valid IBM Cloud Python action.

   The following is returned when this function is invoked:

```
1
2
3
4
5
6
{
    "dbs": [
        "dealerships",
        "reviews"
    ]
}
```
Copied!

2. Simple Node.js function that **does not work**

   The node.js code uses the cloudant npm package to communicate with Cloudant service on IBM Cloud. If you are developing locally, you need to `npm install` the package using the provided `package.json` file. The Node.js runtime on IBM Cloud functions provides this package and you don't have to install anything.

   A common problem that you will face running asynchronous code on serverless platform is that the server returns before the asynchronous function finishes. The file `agfzb-CloudAppDevelopment_Capstone/functions/sample/nodejs` contains code you might expect to work, **however when you run it, you will not get any results back**.

```
1
2
3
4
5
cloudant.db.list().then((body) => {
        body.forEach((db) => {
            dbList.push(db);
        });
}).catch((err) => { console.log(err); });
```
Copied!

   The reason being `cloudant.db.list()` method returns a promise and IBM Cloud Function does not wait for this promise to get resolved. It immediately returns an empty object to the caller.

   The following is returned when this function is invoked:

```
1
{}
```
Copied!

3. Fix using promises

   One way to fix this is have your serverless main function return a Promise instead of the result itself. The `agfzb-CloudAppDevelopment_Capstone/functions/sample/index-promise.js` file illustrates this example

```
1
2
3
4
5
6
7
8
9
10
11
function getDbs(cloudant) {
    return new Promise((resolve, reject) => {
        cloudant.db.list()
            .then(body => {
                resolve({ dbs: body });
            })
            .catch(err => {
                reject({ err: err });
            });
    });
}
```
Copied!

   The above function returns a promise that is resolved when the results come back or rejected if there is an error. The main function can now use this promise returning function as follows:

```
1
2
3
4
5
6
7
8
9
10
function main(params) {

    const cloudant = Cloudant({
        url: params.COUCH_URL,
        plugins: { iamauth: { iamApiKey: params.IAM_API_KEY } }
    });

    let dbListPromise = getDbs(cloudant);
    return dbListPromise;
}
```
Copied!

   The following is returned when this function is invoked:

```
1
2
3
4
5
6
{
    "dbs": [
        "dealerships",
        "reviews"
    ]
}
```
Copied!

4. Fix using async await

   You can also use the async await keywords that act as syntactic sugar and make promise driven code easier to read and write. The file `agfzb-CloudAppDevelopment_Capstone/functions/sample/index-async-await.js` has the code that also returns the database names, but uses async await keywords

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
 async function main(params) {
     const cloudant = Cloudant({
         url: params.COUCH_URL,
         plugins: { iamauth: { iamApiKey: params.IAM_API_KEY } }
     });


     try {
         let dbList = await cloudant.db.list();
         return { "dbs": dbList };
     } catch (error) {
         return { error: error.description };
     }
 }
```
Copied!

   Notice the main function now has the keyword `async` in front of it and the `cloudant.db.list()` method that returns promise has the keyword `await` in front of it. This removes the need to write the `then` and `catch` clauses. The errors are handled using the regular `try` and `catch` clauses that you are already familiar with.

   The following is returned when this function is invoked:

```
1
2
3
4
5
6
{
    "dbs": [
        "dealerships",
        "reviews"
    ]
}
```
Copied!

3. Create backend services

Now that you know how to create IBM Cloud Function actions in Node.js and Python that can use the Cloudant SDKs to return information, use the same framework to create the following endpoints:

Please follow the instructions given here to create API Endpoint URL's.

1. Get all dealerships.

- Language: Node.js
- Endpoint: `/api/dealership`

- Method: GET
- Parameters: None
- Output: List of dealerships with details

```
[
    {
        "id": 1,
        "city": "Worcester",
        "state": "Massachusetts",
        "st": "MA",
        "address": "96 Mariners Cove Place",
        "zip": "01654",
        "lat": 42.3648,
        "long": -71.8969
    },
    {
        "id": 2,
        "city": "Topeka",
        "state": "Kansas",
        "st": "KS",
        "address": "3 Schlimgen Street",
        "zip": "66667",
        "lat": 39.0429,
        "long": -95.7697
    }
]
```

Copied!

Error:

- 404: The database is empty
- 500: Something went wrong on the server

2. Get all dealerships for a given state. You can enhance the previous function to add this functionality.

- Language: Node.js
- Endpoint: `/api/dealership?state=""`
- Parameters:
  - state: the abbreviated state name
- Output: List of dealerships from the state. If the call was `/api/dealership?state="California"`, the output would be:

```
[
    {
        "id": 8,
        "city": "San Jose",
        "state": "California",
        "st": "CA",
        "address": "83 Reinke Hill",
        "zip": "95173",
        "lat": 37.3352,
        "long": -121.8938
    },
    {
        "id": 17,
        "city": "Inglewood",
        "state": "California",
        "st": "CA",
        "address": "4 Glacier Hill Court",
        "zip": "90305",
        "lat": 33.9583,
        "long": -118.3258
    },
    {
        "id": 30,
        "city": "San Francisco",
        "state": "California",
        "st": "CA",
        "address": "0 Loeprich Drive",
        "zip": "94164",
        "lat": 37.7848,
        "long": -122.7278
    },
    ...
]
```

Copied!

Error:

- 404: The state does not exist
- 500: Something went wrong on the server

3. Get all reviews for a dealership.

- Language: Python
- Endpoint: `/api/review?dealerId=""`
- Method: GET
- Parameters:
  - dealerId: the unique id of the dealership
- Output: List of reviews for the given dealership

```
[
    {
        "id": 1,
        "name": "Berkly Shepley",
```

```
            "dealership": 15,
            "review": "some review",
            "purchase": true,
            "purchase_date": "07/11/2020",
            "car_make": "Audi",
            "car_model": "A6",
            "car_year": 2010
        },
        {
            "id": 2,
            "name": "Gwenora Zettoi",
            "dealership": 15,
            "review": "some review",
            "purchase": flase
        }
    ]
```

Copied!

Error:

- 404: dealerId does not exist
- 500: Something went wrong on the server

4. Post review for dealership

- Language: Python
- Endpoint: `/api/review`
- Method: POST
- Parameters:
  - JSON of the review as follows:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
{
"review":
    {
        "id": 1114,
        "name": "Upkar Lidder",
        "dealership": 15,
        "review": "Great service!",
        "purchase": false,
        "another": "field",
        "purchase_date": "02/16/2021",
        "car_make": "Audi",
        "car_model": "Car",
        "car_year": 2021
    }
}
```

Copied!

Error:

- 500: Something went wrong on the server

Submission

Take a note of the following URLs to submit for peer review:

1. URL of github repo of your solution code.
2. URL of the GET dealership endpoint `/api/dealership`.
3. URL of the GET reviews endpoint `api/review`.
4. URL of the POST endpoint `/api/review`.

Summary

In this lab, you uploaded the dealership data provided to you as a JSON file into the Cloudant database. You then created the different IBM Cloud Function actions served by the API gateway.

Author(s)

Upkar Lidder

Other Contributor(s)

Yan Luo

Priya

Changelog

Date	Version	Changed by	Change Description
2021-01-28	1.0	Upkar Lidder	Created new instructions for Capstone project