Rapid Edge of DMS Web Application’s Establishment: Flask+Celery+Redis+Cloudant

Fang Wang, Yi Lu, Zhuo Yang

# System Introduction

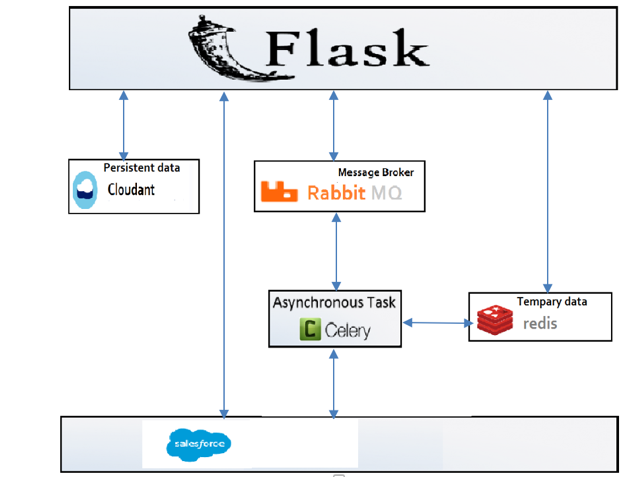
The DMS (Data Migration System) Web System was developed by Python, and it is suitable for cross-platform and cross-system business data’s migration. The system depends on IBM Cloud platform and provides high reliability and convenience, that makes it possible to set up a system in a short time and migrate business data quickly.

The DMS system can be divided into four modules:

* Web UI: As a direct contact with users, this module is a very important one in the system. This system uses Flask as the framework to render the background returned data to the front-end.
* Test module: This system uses Jupyter to test the testing code.
* Data Processing module: Considering that Panda has good support for different types of data files such as ‘.csv’, ‘.txt’ and ‘.pickle’, this system uses Pandas for data processing.
* Data storage module: This system uses Salesforce for data storage. Due to the long-term of writing data into Salesforce and the large amount of data, this system doesn’t write data into Salesforce directly but sends the data to Redis and Cloudant for a better user experience. The system uses Redis as a cache to store temporary variables and uses Cloudant to store the persistent variables (when the data amount is small, write the data into Salesforce directly). According to Flask’s working mechanism, data needs to be serialized first and then stored before deserialized and used. This system uses Pickle to serialize and deserialize data, uses Celery task scheduling to dispatch the message queue and temporary data, and finally complete the server communication task by writing data into Salesforce.

The following figure is an architecture diagram of a DMS Web System based on Flask:

Figure 1. System Architecture Diagram



# Technology Introduction:

## Flask

Flask is a lightweight Web framework based on Python, it’s WSGI toolkit uses Werkzeug and temple engine uses Jinjia2. For Flask doesn’t depend on special tools or repositories and it doesn’t have data abstract layer, form validation or any other functions that the existing repositories can qualified, it keeps a simple core and easy to extend. Thus, Flask is defined as a micro framework, it can add application functions by extending. Flask has its own development server and debugger, integrated unit test and Restful request dispatching, and supports secure cookies.

Jinjia2 is a temple engine based in Python, it supports Unicode, has integrated sandbox execution environment and supports for selecting auto-escaping. Jinjia2 has an automatic HTML transfer system which can effectively prevent XSS. By template inheritance mechanism, Jinjia2 uses similar layouts to all temples and speeds up temple execution time by converting source code to Python bytecode at the first loading.

## Redis

Redis is a Key-Value storage system which written in ANSIC and complies BSD protocol. Redis has follow characteristics: support data persistence, support string, map, list, set, sorted set, etc data structures and support data backup.

Redis can write the updated data into disk and set down the changing operations into appended files periodically and achieve Master-Slave synchronization on this basis. So, data can be synchronized from the Master server to any number of Slave servers, slave servers can be the master server which is associated with other slave servers. Due to the realization of publish/subscribe mechanism, it is possible for slave servers to subscribe a channel and receive the master server’s complete published records when synchronize tree from anywhere.

MQ

MQ message queue is a method for applications’ communication. Applications can communicate by reading and writing messages. MQ is one kind of Producer-Customer patterns’ realization. Producer-customer pattern consists of three modules: producer, customer and cache. Cache performs as a mediation, producer puts data into the cache and customer takes data from the cache. In this system, Flask is the producer, Salesforce is the customer and MQ is the media cache. Using producer-customer pattern can effectively reduce the mutual dependence between Flask and Salesforce. Because of the existence of cache, customers don’t need to obtain data from the producer, which helps support concurrent tasks and reduce congestion.

RabbitMQ is an open source implementation of AMQP (advanced message queue protocol), which developed by erlang. As a message queue management tool, RabbitMQ is responsible for process communication tasks between servers after integrated with Celery. RabbitMQ is used as follow steps:

1. The client connects to the message queue and opens a channel;
2. The client declares an exchange, a queue, and sets the relevant properties;
3. The client binds the relationship between exchange and queue by routing key;
4. The client sends messages to the exchange, then the exchange posts the messages to the queue according to the message key and the binding set.

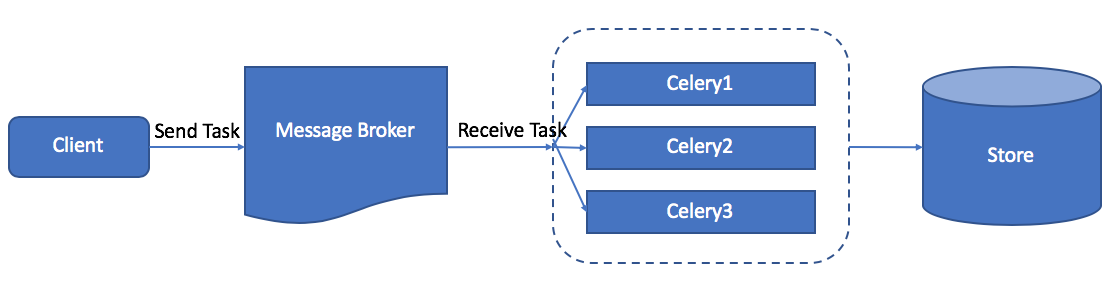
RabbitMQ’s three commonly use exchange type:

* Fanout: be able to post all the message which sent to this exchange to all the binding queues.
* Direct: send messages to the queues whose binding key matches routing key.
* Topic: routing messages to queues whose binding key matches routing key.

Celery

Celery is a distributed asynchronous message task queue based on Python, by which can realize tasks’ asynchronous processing and task schedule execution on distributed machines, threads and processes by task queue. Moreover, Celery has high availability and efficiency of automatically trying to re-execute the task when the task fails or interrupted. For Celery’s architecture consists of message broker, task execution unit and task execution result storage, it needs a message broker’s cooperation to receive and send task messages. In this system, the message broker is achieved by Redis+RabbitMQ, in which RabbitMQ perform as Broker and Redis performs as backend. Figure 2 is the work flowchart of Celery:

Figure 2. Celery’s Work Flowchart



Cloudant

IBM Cloudant NoSQL Database for Bluemix (brief: Cloudant) perform as a database in the whole system, it is a multitenancy, independent installed service based on Apache CouchDB. Cloudant is a NoSQL database service (DBaaS). This service is an operational data storage which is used to store JSON files, build up index and tackle parallel reading and writing operations. Clients can get, put or query JSON files from application program through HTTP API and build up index.

Pickle

Pickle is one of Python’s standard modules which doesn’t require a separate installation and has two main kinds of interfaces:

Serialization: this operation can save the running program’s object information into files and store them permanently.

Deserialization: this operation can create the object which saved by the pervious program from the files. The modularized object is stored in a binary file with suffix ‘.pkl’.

# Rapidly Set up Project Environment

With understanding of the roles that different technologies play in the system, we are now ready to set up a project.

Build Python Environment

For Flask is a micro framework based on Python, it needs to ensure the computer has Python environment. Input Python into the computer terminal to check whether there’s a Python environment. If there’s a version number, Python has been installed. If there’s an error, no Python environment exists. If your computer is macOS, it owns Python 2.7 itself.

Build Flask Environment

1. Install Virtualenv

There are multiple versions of Python, and different versions are incompatible. So, using different versions of Python in different projects can cause external library incompatible. Therefore, we need to install the corresponding version according to the project ‘s requirements. Virtualenv provides a good support to all the versions of Python, based on that we can conveniently select installations according to the project requirements.

* Linux/macOS X: install by $ sudo easy\_install virtualenv or use pip $ sudo pip install virtualenv. But the latter one needs to ensure we have pip environment already.
* Ubuntu: install by $ sudo apt-get install python-virtualenv

After the virtual environment has been set up, we can start to create Flask working environment as List 1:

List 1. Create Flask Work Environment

$ mkdir my-project

$ cd my-project

$ virtualenv venv

When there comes New python executable in venv/bin/python

Installing distribute…done. Successfully installed. When working in a program, execute the following command to activate virtualenv:

$ . venv/bin/activate (Mac/Linux)

$ venv\scripts\active (Windows)

1. Install Flask

After virtualenv’s activation, use command $ pip install Flask to install and activate Flask.

1. Build a Flask demo, as List2:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

    return 'Hello World!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

save as hello.py, run $ python hello.py in the terminal and then run localhost: 5000/ in the browser, the Flask program has been started. We can also modify the calling method by app.run(host=’0.0.0.0’)

Redis Installation and Use

Due to the large amount of data, we need a cache to store temporary vairables, so we choose Redis. Redis can be installed by the command line:

* macOS: install by $ brew install redis, and start by $brew services start redis or $ redis-server /usr/local/etc/redis.conf
* Linux: install by the order:

$ wget http://download.redis.io/releases/redis-2.8.17.tar.gz

$ tar xzf redis-2.8.17.tar.gz

$ cd redis-2.8.17

$ make

Start by: $ cd src

$ ./redis-server

* Windows: Download package from <https://github.com/MSOpenTech/redis/releases>. Save the Redis-x64-xxx.zip package to disk C, after unzip the package, rename it as redis. Open a CMD window and use command to switch directory to C:\redis and run redis-server.exe redis.windows.conf.

For convenience, we can put Redis path into the system environment, then we don’t need to input path anymore. ‘redis.windows.conf’ can be omitted, if it is omitted, the default one will be enabled. At this time, open another CMD window (note: don’t close the former one, or we will not be able to visit the server). Switch to Redis directory to run redis-cli.exe -h 127.0.0.1 -p 6379.

Set key-value: set myKey abc

Get key-value: get myKey

* Ubuntu: install by command:

$sudo apt-get update

$sudo apt-get install redis-server

* Start by command: $redis-server

After installation, we need to config Redis, Redis’ default config file redis.config is under the root directory. We can obtain and set all Redis config through Redis’ config command: $redis 127.0.0.1:6379>CONFIG GET CONFIG\_SETTING\_NAME.

After finishing config, Redis is only one step away from being used. In order to import it into program, we use iredis.Redis(host=url, port=port, db=0) to connect Redis to the local database and finally connect RPC through redis\_client=redis.from\_url (REDIS\_URL).

## MQ Installation and Use

Now we have cache, we also need communication. In our system we use message queue to achieve communication. Install through $ brew install rabbitmq. Enter the installation directory after installation and execute $ sbin/rabbitmq-server to start MQ service. Then, configure the broker URL in Celery. We will discuss how to configure in detail in Celery’s chapter.

## Celery Installation and Use

With a message queue, what we need now is a role to dispatch the queues. Here comes Celery. Install through the command $ sudo easy\_install Celery or $ sudo pip install Celery.

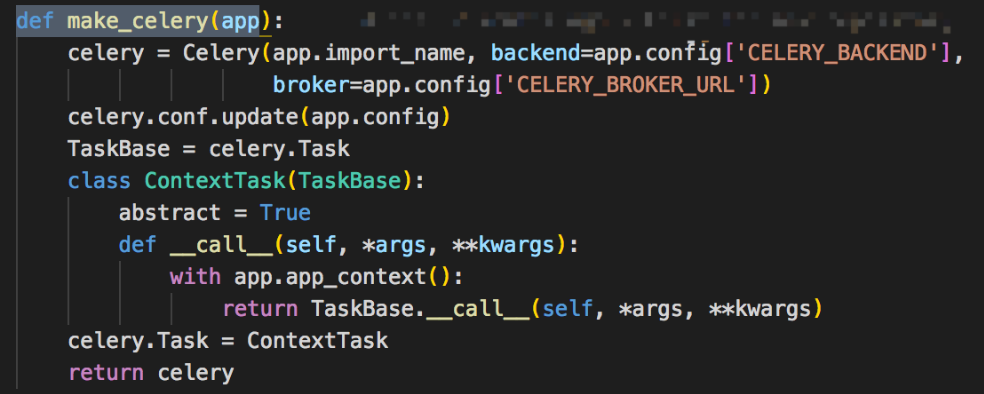
Similarity, Celery needs to be configured after installed:

* When Celery uses the default Broker, we configure RabbitMQ as follow: app.config.broker\_url=’ amqp://guest:guest@localhost:5672’
* When Celery uses Redis as Broker, we need to install Redis component by the follow command: $pip install -U celery-with-redis.

Then configure: app.conf.broker\_url = 'rediss://admin:password@host.composedb.com:39379'. After configuration, we can create dispatch task as the follow steps:

1. Import Celery: from celery import Celery, then write it into configure by app.coonfig.update as Figure 3:

Figure 3. Declare Celery



1. Create Celery task as List3:

List 3.Create Celery task

@celery.task(bind=True)

def run(x):

return x

@celery.task(bind=True)

def power(x):

return x\*x

1. Start the Celery Worker:

$ celery -A name.py worker -l info

1. Call the function declared in Celery in the Python script, and we can run the Python script.

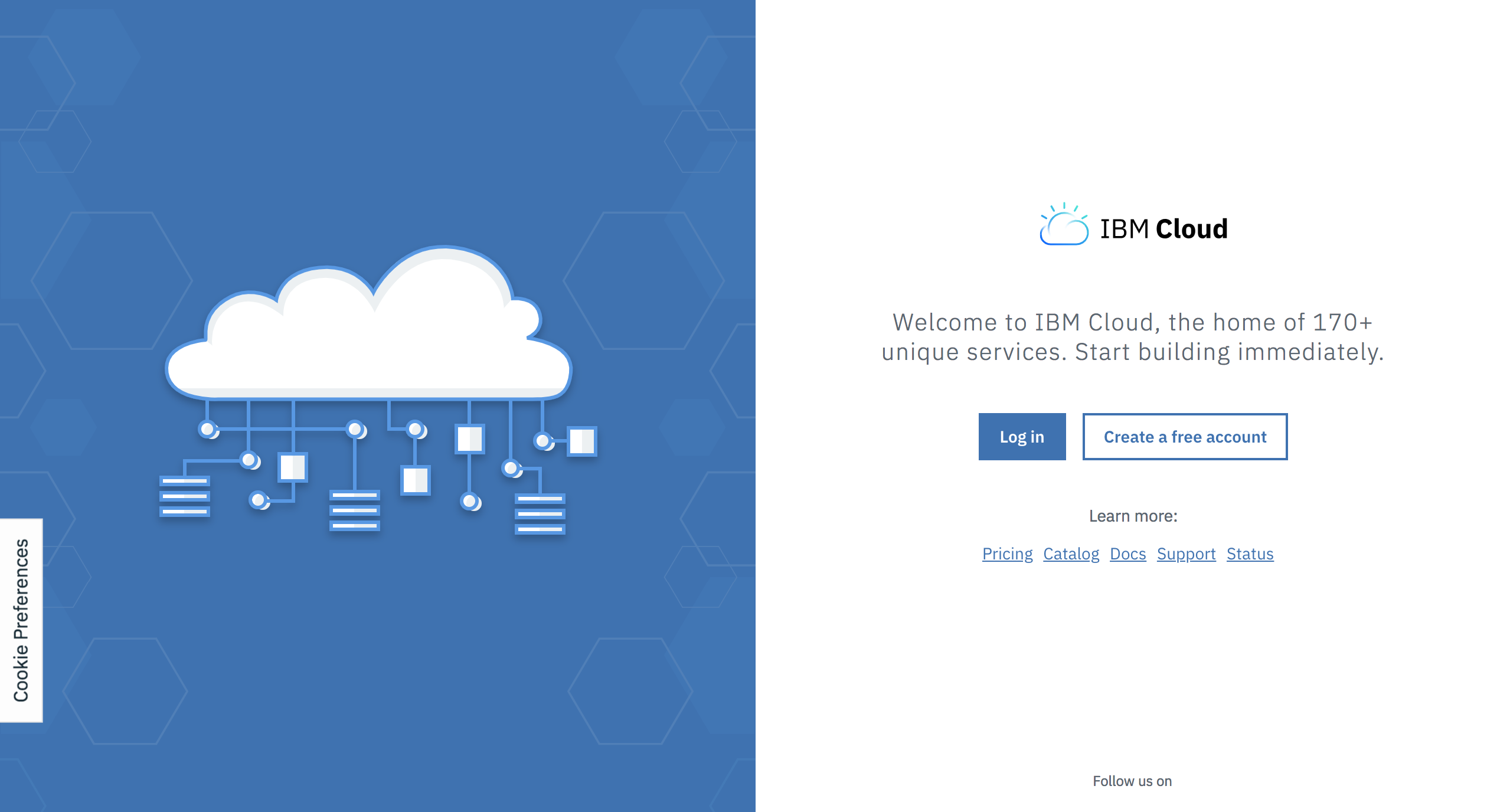
From project.tasks import run, power

run.delay(args)

## Rapidly Use Cloudant

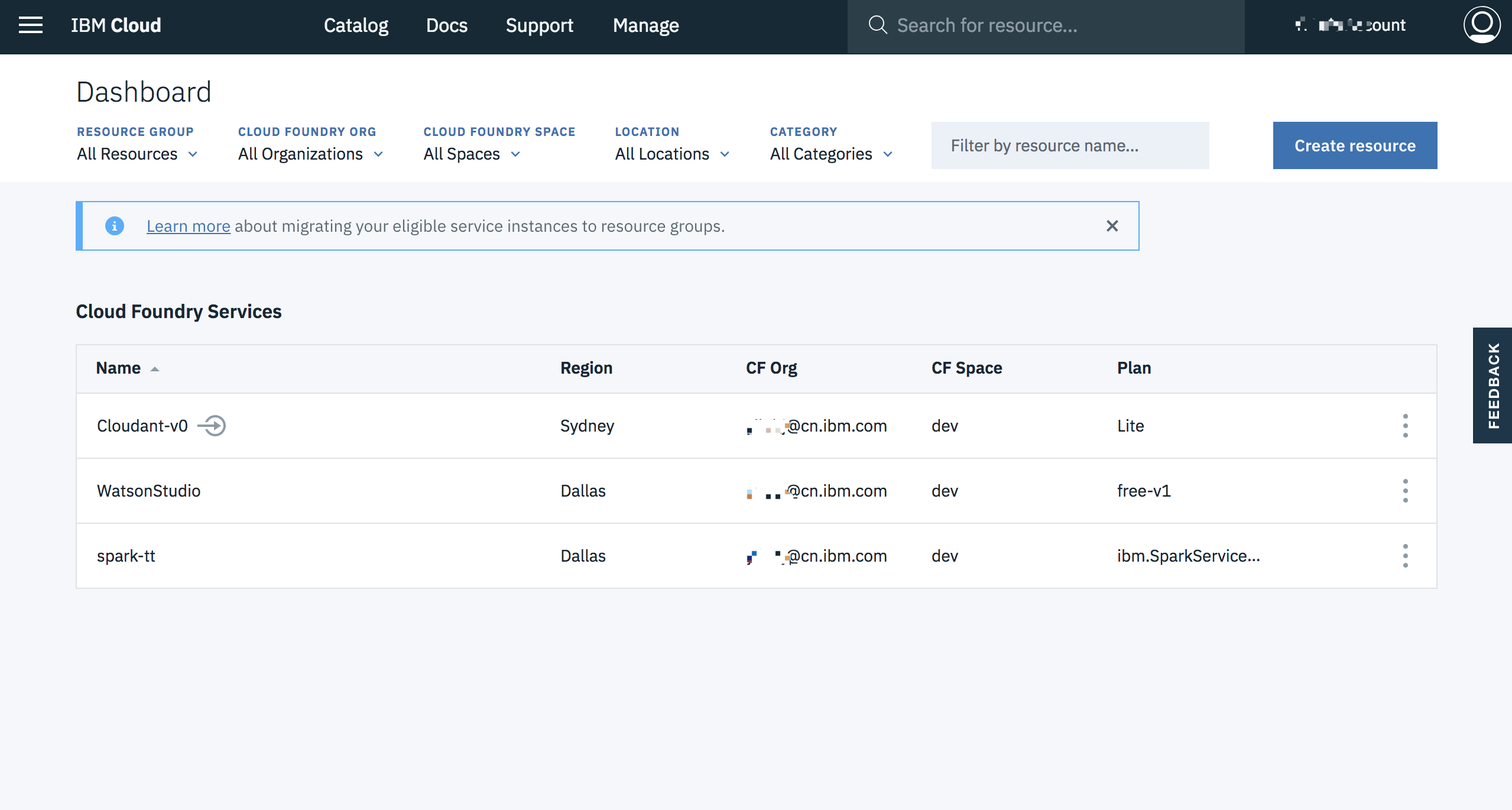
Apart from temporary variables, persistent data also needs to be stored in a project. CLoudant is needed to store persistent data. Cloudant requires you to create an IBM Cloud account and connect it to your IBM ID before creating a database. Enter url <https://console.bluemix.net/> into the browser and then get into IBM Cloud’s register page as Figure 4:

Figure 4. IBM Cloud Entrance



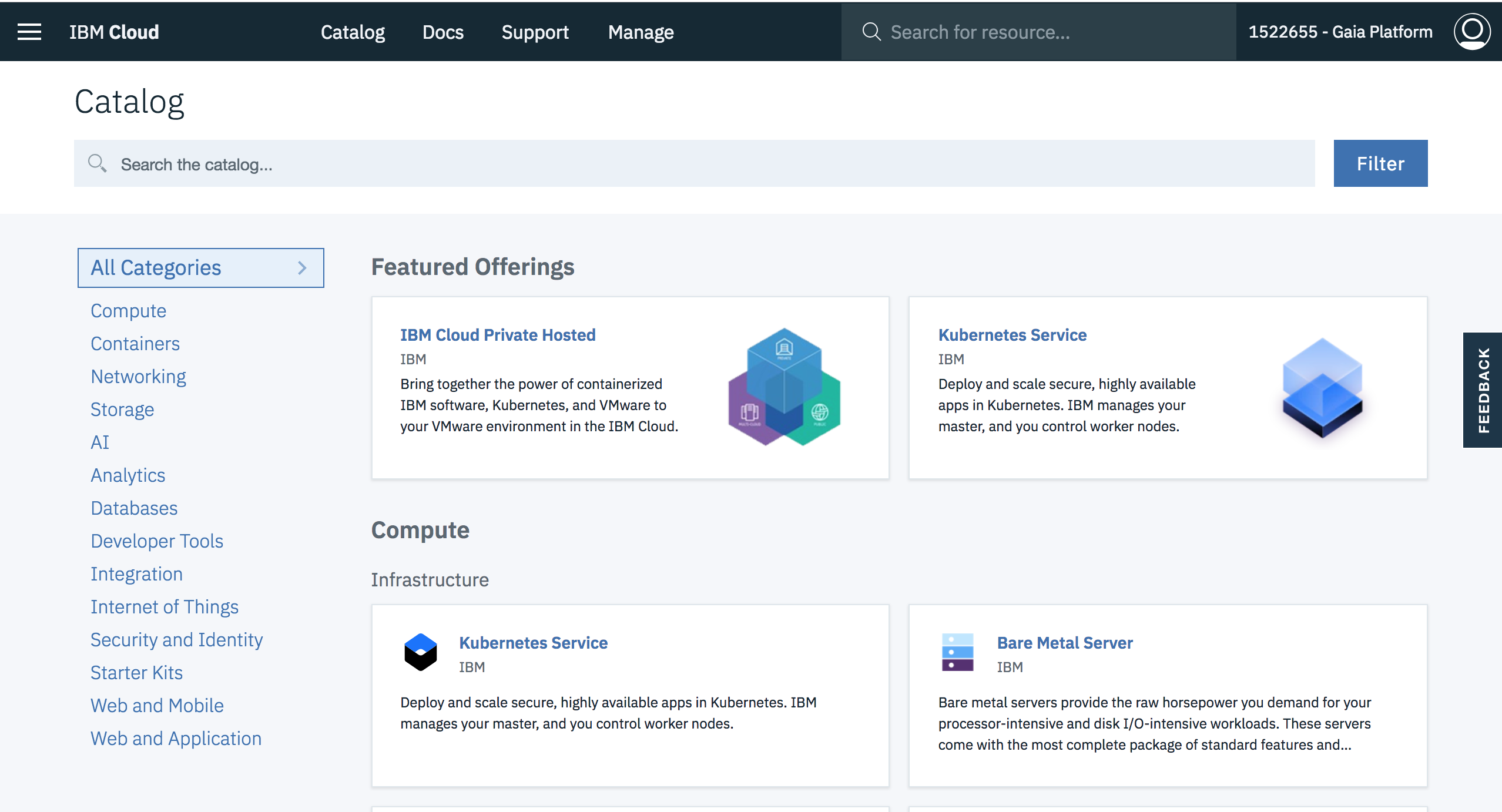
After entering the official website and creating an IBM Cloud account and then you can see the resource information under the account as Figure 5:

Figure 5. Dashboard Files



Click Catalog and enter Databases > Cloudant

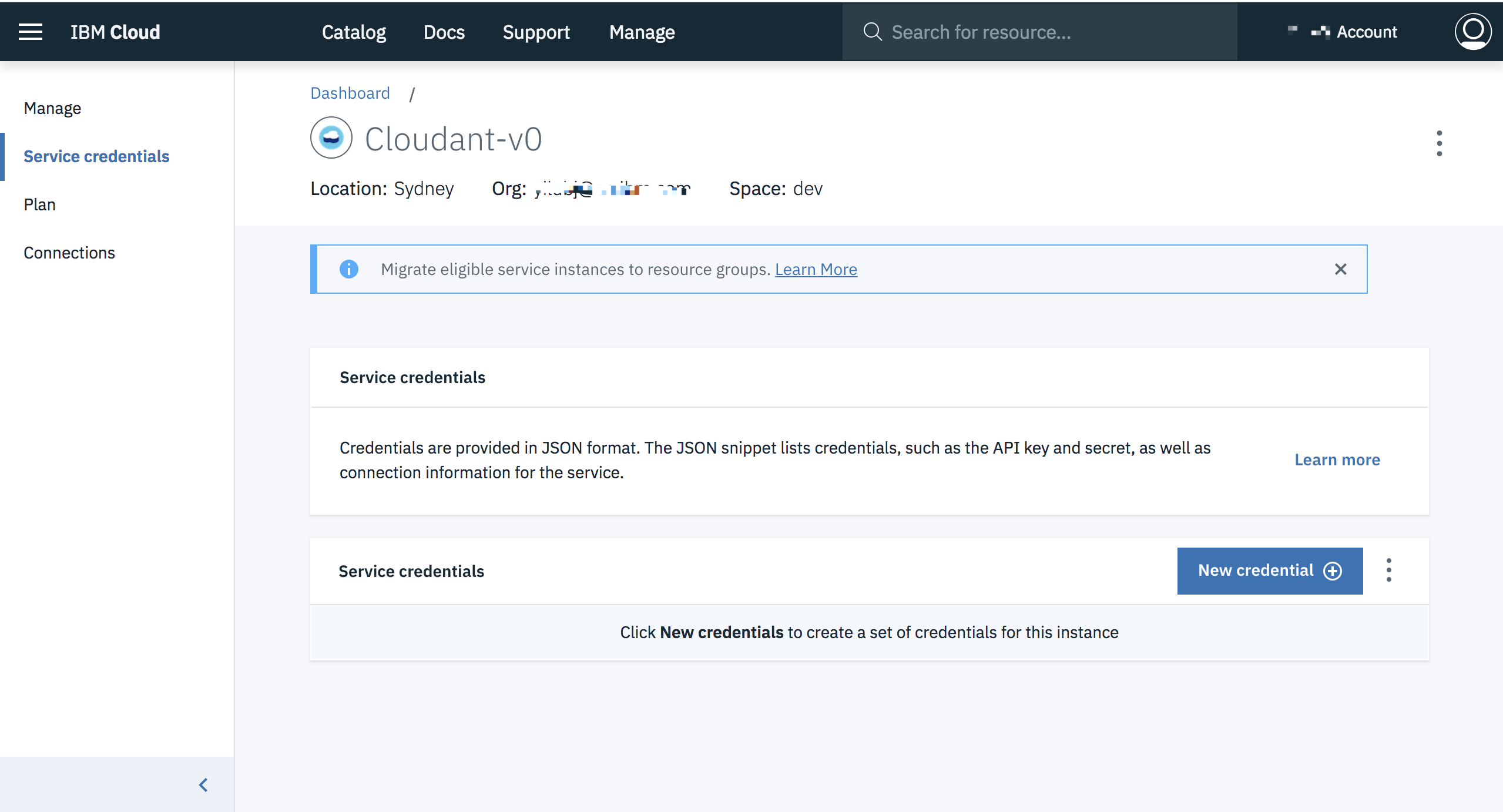
Figure 6. Cloudant Entrance



After getting an IBM Cloudant account, install Python library to use Cloudant through: $ pip freeze. Check whether Cloudant 2.8.1 has been installed in the installation list, if not, run $ pip install cloudant==2.8.1. Then import Clooudant into Python file by import cloudant.

Now, we can connect Cloudant service into IBM Cloud. Login into IBM Cloud first, and then add Cloudant NoSQL DB service into IBM Cloud space, open the dashboard, click and get into service and click the service credential on the left to add credential as Figure 7:

Figure 7. Add Credential Page



Make basic authentication in script: client = Cloudant (USERNAME, PASSWORD, url=url, connect=True)

Then, create database:

1. Define database name in Python script: databaseName = “<DatabaseName>”.
2. Create database: myDatabase = client.create\_database(databaseName).
3. Check whether the database has been created successfully through:

If myDatabase.exists():

Print “’{0}’ successfully created .\n”.format(databseName)

If the database has been created successfully, we can store the data and do some operations, such as:

* Define a group of data as List 4.
* Obtain data row by row through Python as List 5.

List 4. Define a group of data

sampleData = [

   [1, "one", "boiling", 100],

   [2, "two", "hot", 40],

   [3, "three", "warm", 20],

   [4, "four", "cold", 10],

   [5, "five", "freezing", 0]

 ]

List 5. Obtain data row by row through Python

# Create documents by using the sample data.

# Go through each row in the array

for document in sampleData:

 # Retrieve the fields in each row.

 number = document[0]

 name = document[1]

 description = document[2]

 temperature = document[3]

 # Create a JSON document that represents

 # all the data in the row.

 jsonDocument = {

     "numberField": number,

     "nameField": name,

     "descriptionField": description,

     "temperatureField": temperature

 }

 # Create a document by using the database API.

 newDocument = myDatabase.create\_document(jsonDocument)

 # Check that the document exists in the database.

 if newDocument.exists():

     print "Document '{0}' successfully created.".format(number)

## Use Pickle

When processing data, there may come up with some intermediate data. In Web application, data should not be stored in session, which can lead to memory problems. Serializing and storing data to hard disk is a good idea. Pickle is Python’s best data serialization and deserialization tool. For Python has already integrated Pickle, we can reference it directly by: import pickle directly.

Pickle has its own serialization and deserialization API:

* Store data: pickle.dump(obj, file, [,protocol])
* Get data: pickle.load(file)

Pandas can also operate Pickle by simple API:

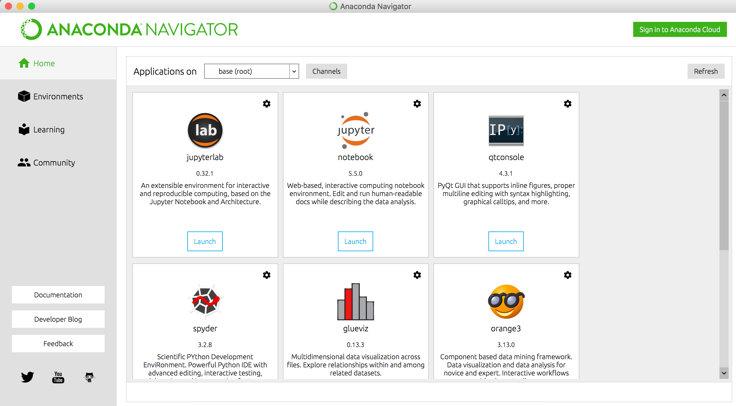
* Store data: df.to\_pickle(‘test.pkl’)
* Get data: df.read\_pickle(‘test.pkl’)

# Create Test Script

An entire system tends to be large, when code is changed, it can influence anywhere of the system. So, we need break points to help with debugging. By creating test script, we can determine the feasibility of the code and move the feasible code to system can decrease the debug effort. Now we begin to create and install Jupyter:

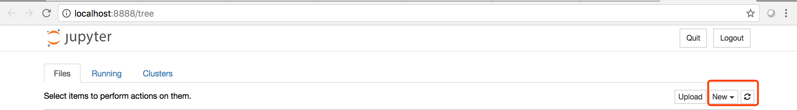
1. Install Jupyter notebook. macOS owns Python itself. Install through: $ pip install –user jupyter.
2. Start Jupyter: $ jupyter notebook. Or start through anaconda-Navigator. When enter into the application, click Jupyter notebook’s Launch button, Jupyter is started.

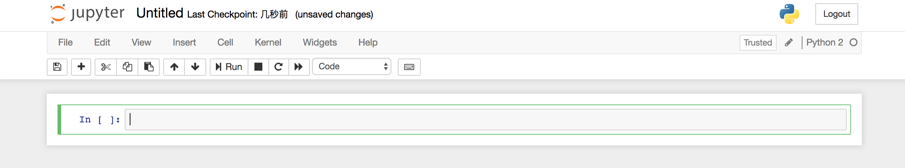
Figure 8. Jupyter’s login page



When Jupyter is started, page in browser will show the file list of the current directory. Operate files by New or Upload button. A new ‘.ipynb’ file can be created by clicking New button.

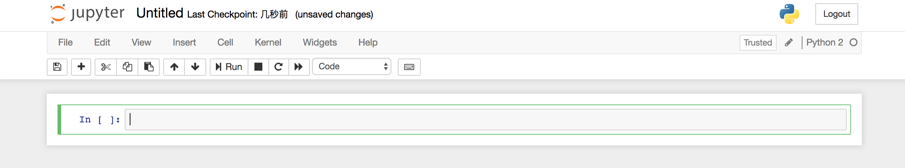
Figure 9. Jupyter’s New File Page





1. As Figure 10, enter the webpage you can test the code, click Run button to run the code.

Figure 10. Jupyter Test



# Data Processing

Due to it involves amount of data processing in this system, use Pandas to process data. Pandas is a Python data analysis package which can be used for data analysis. Pandas provides many standard libraries and APIs for rapidly processing data and functions, such as data reading, filtering, sorting, grouping, selection, cleaning, etc operations.

Install Pandas through $ pip install pandas and import by pandas as pd. Pandas’s API can support both files and data. For example:

* Read files:

pd.read\_excel(filename);

pd.read\_pickle(filename);

pd.read\_csv(filename);

* Store files:

pd.to\_csv(filename);

pd.to\_pickle)filename);

pd.to\_excel(filename);

# Data Storage

After data is processed, we need to store data into Salesforce. It can be achieved by Simple Salesforce:

1. Run $ sudo pip install simple\_salesforce to install Simple Salesforce.
2. Import Salesforce into script: from simple\_salesforce import Salesforce and authentication configuration.
3. Create sf = Salesforce(instance\_url=cred[‘instance\_url’], session\_id=’’) through API and use sf.apexecute to import Apex (one kind of Object-Oriented Programming language, which can execute program functions in most processes). Finally store data into this object.
4. Use sf object to store data:

sf.Contact.create({‘LastName’: ‘Smith’, ‘Email’: ‘example@example.com’}).

# Project Deployment

This project is deployed on IBM Cloud. First login to IBM Cloud and enter the dashboard page. Click Catalog, there comes a series of available services. Choose Python entrance, after setting down information, click Create button and create an application environment. After a while, there will be a newly created application environment. After that we can upload the application to the IBM Cloud.

Figure 11. IBM Cloud Item Catalog

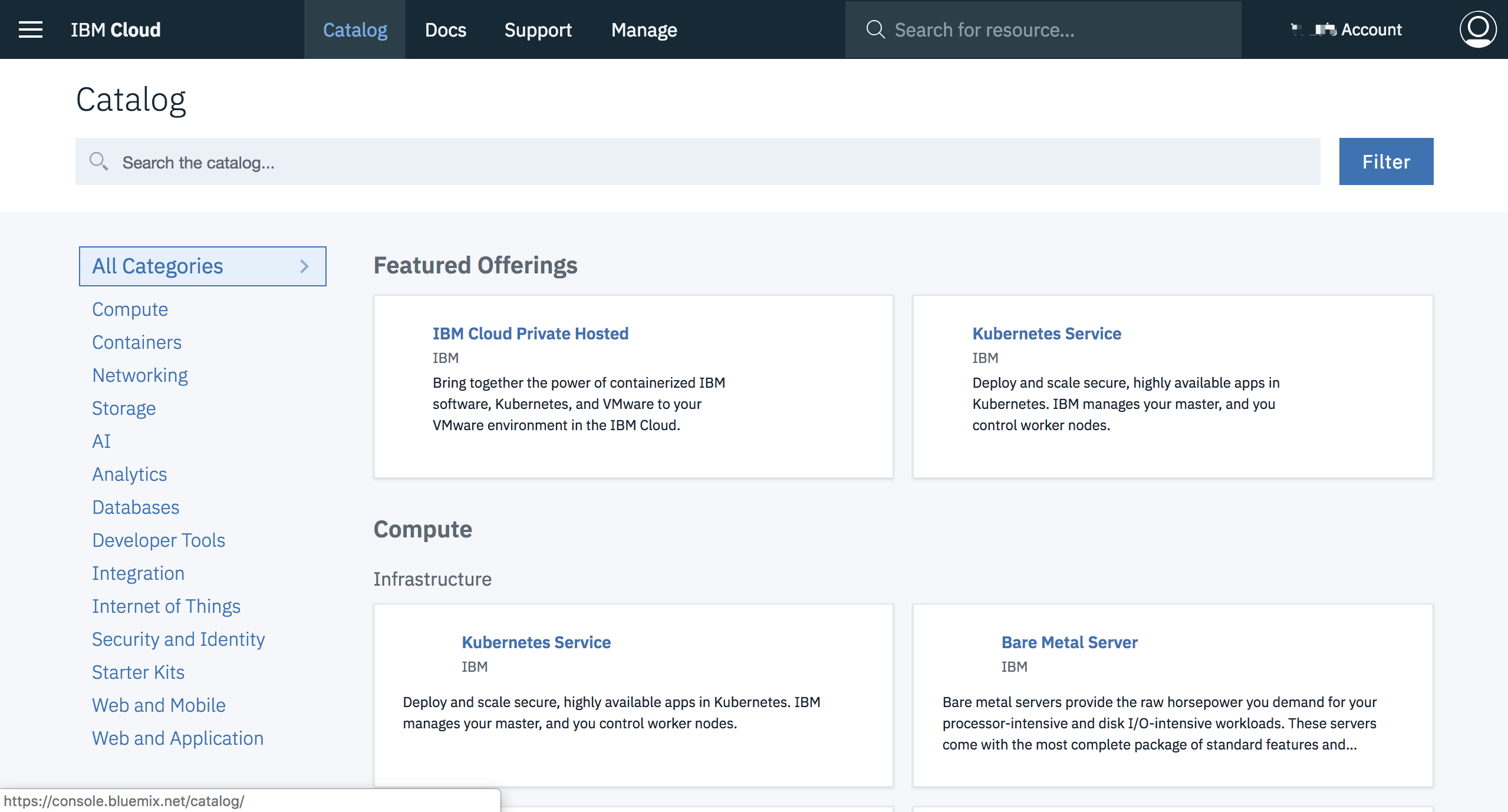


Figure 12. Create Cloud Foundry Application Program

When uploading application, we need IBM Cloud CLI. It can be downloaded in the application introduction page which just created.

1. First tell Cloud Foundry the url by command:

$ bluemix api https://api.ng.bluemix.net.

1. Then login to IBM Cloud environment by command. Use command:

$ bluemix login -u username -o organizationname

to open the program’s root directory to ensure there’s a ‘manifest.yml’ file in the current directory.

1. Then, use command $ bluemix app push to upload application and configure according to the content in ‘manifest.yml’.

‘manifest.yml’ should be placed under the program’s root directory as List 6. File contents represent hierarchical relationships by YAML indentation:

List 6. manifest.yml demo

---

command: pip install --upgrade pip

applications:

- name: GMTDEV

  memory: 512M

  instances: 1

  command: gunicorn run:app --workers=4

- name: GMTCeleryDEV

  memory: 512M

  instances: 1

  no-route: true

  command: celery -A tasks.celery worker --loglevel=info

  health-check-type: process

After finishing upload, we can check the upload status in dashboard. When there shows the follow figure, the deployment is successfully:

Figure 13. Application Program Management Dashboard

# Summary

This paper described the whole process of building a Web project rapidly through ‘Flask+Celery+Redis+Cloudant’. It first introduced Flask, Redis, MQ, Celeery, Cloudant, Pandas, etc technologies and the way to install and use them. Then introduced the project’s architecture and showed a project’s establishment process. Finally showed how to deploy a project in IBM Cloud. Hope that this paper can help you create a DMS Web system quickly and efficiently.

# Reference

* Flask Tutorial
* Pandas Tutorial
* MQ Tutorial
* Cloudant Tutorial
* Celery Tutorial
* Register IBM Cloud