

Trabajo Practico#5

Desarrollo - Estructura del proyecto CryptoAppMongo

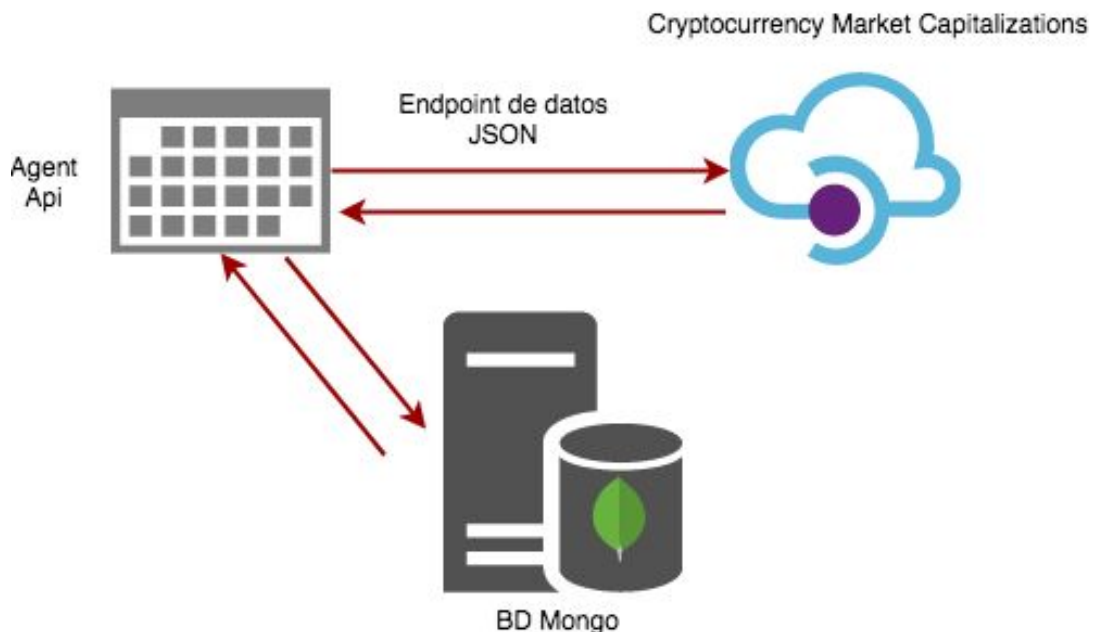
Requerimientos:

Una empresa de finanzas que opera con cryptomonedas, nos solicita un modulo de desarrollo para poder incorporar a sus sistemas de gestión a clientes financieros y reportar las cotizaciones de las criptomonedas en tiempo real.

Los requerimientos del mismo constará en desarrollar una API-REST para alojarlos en una base de datos MongoDB y poder consumirlos desde la aplicación que deberá realizarse en Python.

Esta aplicación tiene dos componentes, el componente de agente, que es la aplicación que extrae la información de [CoinMarketCap](https://coinmarketcap.com/api/) y deberá guardarla en la base de datos, y una API Rest que va a leer esa información de la base de datos y mostrarla a través del navegador.

API: <https://coinmarketcap.com/api/>



Librerías que usaremos en nuestra app y cómo instalarlas

En CryptoAppMongo utilizaremos tres librerías: PyMongo, Requests y Flask.

Flask - Es un microframework de Python.

Documentación: <http://flask.pocoo.org/>

Para instalar cada librería usaremos **pip3**

Instalación de Requests y Flask

Al momento de instalar es importante conocer si está pip con Python 3, para eso debes ejecutar:

```
$ pip3 -V  
pip 8.1.1 from /usr/lib/python3/dist-packages (python 3.5)
```

Si aparece (**python 3.x**) significa que está instalado con Python 3, en caso que de error podrías probar con pip3.

Ya con la versión de Python 3 en **pip** hay que ejecutar la instalación de **Flask y Requests**.

```
$ pip install requests  
$ pip install flask
```

Verificar la instalación

En la consola de Python escribir

```
>>> import requests  
>>> import flask
```

Si no aparece ningún error significa que ya las librerías están disponibles.

Licenciatura en Sistemas de Información Bases de Datos NSQL

Modelo del desarrollo

Estructura base del agente:

```
main.py — agent      main.py — api      readme.py      redmy.py

1  import requests #Librería para capturar peticiones rest
2  import pymongo #Librería para comunicarse con la bd mongodb
3
4  API_URL = "https://api.coinmarketcap.com/v1/ticker/" #Url desde donde se obtiene los datos a guardar en mongodb
5
6  #Función para crear conexión a la bd
7  def get_db_connection(uri):
8      pass
9      #Crear una conexión a la bd
10     #Devolver bd cryptoApp
11
12 #Función para obtener datos del api externa
13 def get_cryptocurrencies_from_api():
14     r = requests.get(API_URL) #Obtener datos de la url del api externa
15     if r.status_code == 200: #Si devuelve 200
16         result = r.json() #Formatear resultados en formato json
17         return result #Devolver resultado
18
19     raise Exception('Api Error'); #En caso de no recibir 200 del api, devolver un error forzado
20
21 def first_element(elements):
22     pass #Devolver el key de elements
23
24 def get_hash(value):
25     #Función para encriptar
26     #El string codificado en utf-8, requisito de la librería hash
27     #Convertir encriptado en un string para poder ser almacenado posteriormente en bd
28
29 def get_ticker_hash(ticker_data): #Función para retornar un solo gran string con todos los datos de los items
30     #Permite ordenar una colección bajo un criterio
31     pass
32     sorted( #Función para ordenar cualquier tipo de conjunto de elementos con un criterio
33         #Se le pasa la lista de items a ordenar
34         #Sorted manda a llamar a la función first_element, pasándole una tupla (key, value).
35         #Y lo que retorne, será el valor usado para ordenar
```

Licenciatura en Sistemas de Información Bases de Datos NSQL

Estructura base del API-REST

```
1 import pymongo #Librería para comunicarse con la bd mongodb
2 from flask import Flask, jsonify, request #Flask es para levantar un servidor web. jsonify para poder retornar un js
3
4 #Función para crear conexión a la bd
5 def get_db_connection(uri):
6     client = pymongo.MongoClient(uri) #Crear una conexión a la bd
7     return client.cryptooapp #Devolver bd cryptongo
8
9 app = Flask(__name__) #Crear un servidor web a partir del nombre del archivo ejecutado
10 db_connection = get_db_connection('mongodb://localhost:27017/') #Conexión a la bd
11
12 #Función para obtener lista de documentos a devolver
13 def get_documents():
14     params = {} #Defino un diccionario de datos
15     name = request.args.get('name', '') #Obtengo el campo "name", recibido por get. Si no existe, devuelve vacío
16     limit = int(request.args.get('limit', 0)) #Obtengo el campo "limit", recibido por get. Si no existe, devuelve 0,
17
18     if name: #Si existe "name"
19         params.update({'name': name}) #Lo agrego al diccionario de datos
20
21     cursor = db_connection.tickers.find( #Busco tickers
22         params, #, según lo que recibí por get
23         {'_id': 0, 'ticker_hash': 0} #Campos omitidos
24     ).limit(limit)
25
26     return list(cursor) #convertir resultados de bd en una lista y devolverlo como respuesta a la petición
27
```

Creación del endpoint

Vamos a definir los endpoints para poder llamar las funciones definidas en el API, se utiliza un patrón de diseño, decorator, recordando, que este patrón lo que realiza es la agregación de funcionalidades a los objetos o funciones.

Estos endpoint nos permitirá conectarnos a nuestra base de datos, junto con Postman, realizaremos las funciones de insert y delete.

Requerimientos generales del desarrollo

Modulo del API - REST

import pymongo #Librería para comunicarse con la bd mongodb
from flask import Flask, jsonify, request #Flask es para levantar un servidor web. jsonify para poder retornar un json como respuesta al cliente. request permite capturar peticiones rest.

#Función para crear conexión a la bd

#Crear una conexión a la bd

app = Flask(__name__) #Crear un servidor web a partir del nombre del archivo ejecutado

#Funciones para obtener lista de documentos a devolver

Licenciatura en Sistemas de Información
Bases de Datos NSQL

```
#Definir un diccionario de datos
#Obtener el campo "name", recibido por get. Si no existe, devuelve vacío
#Obtener el campo "limit", recibido por get. Si no existe, devuelve 0, omitiendo
limit. Fuerzo conversión a entero
```

```
cursor = db_connection.tickers.find(
#Buscar tickers
#convertir resultados de bd en una lista y devolverlo como respuesta a la petición.
```

#Función para obtener el top20 de documentos

```
def get_top20():
#Obtener el campo "name", recibido por get. Si no existe,
devuelve vacío.
#Obtener el campo "limit", recibido por get. Si no existe,
devuelve 0, omitiendo limit. Fuerzo conversión a entero
return list(cursor) #convertir resultados de bd en una lista y devolverlo
como respuesta a la petición
```

#Función para eliminar un documento

```
def remove_currency():
    params = {} #Defino un diccionario de datos
    name = request.args.get('name', '') #Obtengo el campo "name",
recibido por get. Si no existe, devuelve vacío
```

Nuestro modelo del patrón para los endpoint podrían tener este formato:

```
@app.route("/") #Defino la ruta raíz para recibir peticiones
def index(): #Cuando alguien pida esta ruta, se ejecutará esta función
    return jsonify(#Retornará una respuesta en formato json
        {
            'name': 'CryptoApp API'
        }
    )
```

```
@app.route("/top20", methods=['GET']) #Defino ruta GET para top20
def top20(): #Se ejecutará esta función cuando alguien entre a esa ruta
    return jsonify(get_top20()) #Devuelvo un json con la lista de top20
```

Uso de Postman

Postman es un entorno de desarrollo para poder ejecutar estructuras de API-Rest.

<https://www.getpostman.com/>

En este entorno podremos ejecutar los endpoint de nuestra api para poder realizar el GET y DELETE de cada colección que queramos eliminar.

Desplegando el proyecto en Dockerfile

Para desplegar CryptoAppMongo se necesita tener instalado y configurado Docker en el entorno dónde se desea desplegar CryptoAppMongo

Una vez instalado Docker debemos empezar a crear las imágenes de Docker con dos Dockerfiles uno para el agente y otro para nuestra **API REST**.

Desplegando nuestro agente

Debemos comenzar por ejecutar un contenedor que contendrá MongoDB para guardar toda la información de las criptomonedas. MongoDB 3.4 se encuentra disponible en la imagen oficial de MongoDB para Docker.

Para ejecutar mongo solo debemos ejecutar desde la consola:

```
$ docker run -d --name=mongo-crypto mongo:3.4
```

Con el comando run le indicamos a Docker que va a ejecutar un contenedor, con el parámetro -d indicamos que se ejecutará en segundo plano, con --name asignamos un nombre personalizado al contenedor y por último especificamos el nombre de la imagen y la etiqueta.

En este caso ejecutamos la imagen mongo con la etiqueta 3.4 para especificar que de la imagen de MongoDB deberá Docker ejecutar la versión 3.4 que está disponible en el repositorio oficial.

Antes de desplegar nuestro agente haremos una modificación al código para agregar un sleep y un while para ejecutar de manera infinita nuestro agente y que espere 4 minutos para volver a pedir información del API. La modificación se debe hacer al principio y al final del archivo.

Al principio del archivo hay que importar la librería time dónde encontraremos la función de sleep. Para eso colocamos import time.

Licenciatura en Sistemas de Información Bases de Datos NSQL

Y al final de nuestro archivo main.py, vamos a reemplazar todo el contenido que está por debajo de la línea que contiene `if __name__ == "__main__":` por el siguiente código:

```
if __name__ == "__main__":  
    while True:  
        print("Guardando información en Cryptongo")  
        connection = get_db_connection('mongodb://mongo-crypto:27017/')  
        tickers = get_cryptocurrencies_from_api()  
  
        for ticker in tickers:  
            save_ticker(connection, ticker)  
        time.sleep(240)
```

Una forma válida y sencilla para ejecutar infinitamente un programa es agregando un `while` para que no se detenga la ejecución.

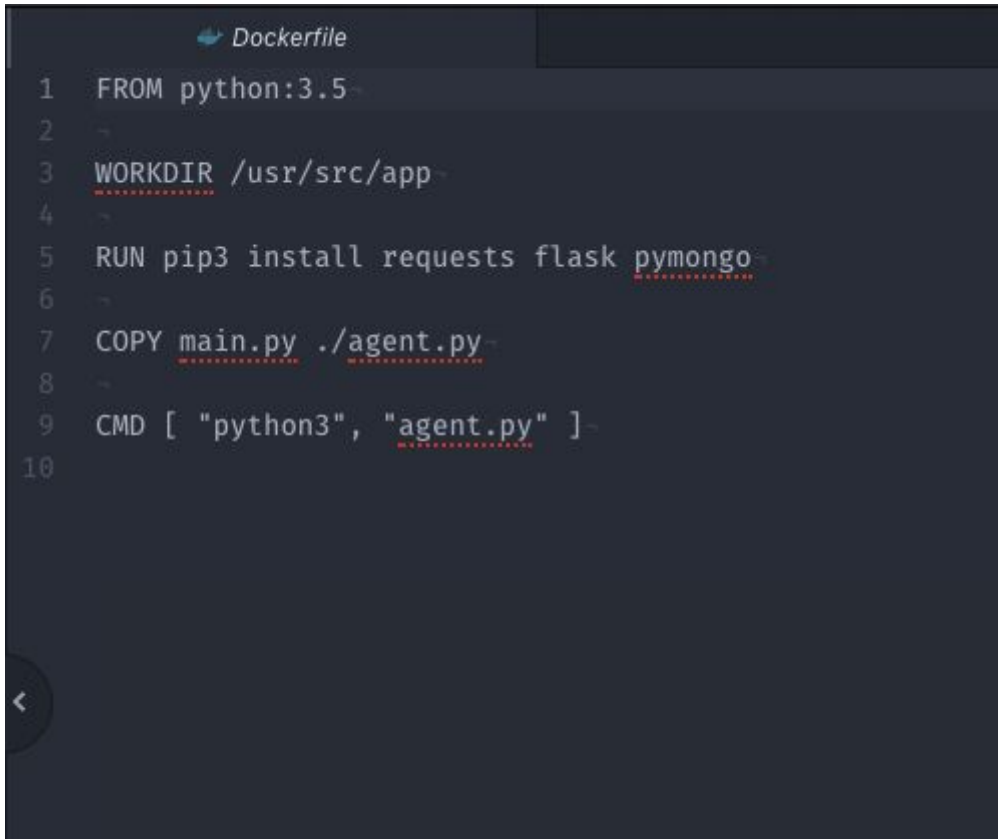
También cambiamos `localhost` por `mongo-crypto` para que enlacemos el contenedor de MongoDB con el del agente aprovechando las características de Docker que me permite especificar un alias como `mongo-crypto` que apuntará a la IP interna del contenedor de MongoDB.

También se agrega `time.sleep(240)` que indica a Python que espere 240 segundos, tras ese tiempo se ejecutará de nuevo el `while`.

El `print` es solo para mostrar algo en sola y verificar que todo se ejecute correctamente.

Una vez se hagan todos esos cambios procederemos a crear nuestro `Dockerfile` en la misma carpeta donde se encuentra `main.py` de nuestro agente.

El contenido del Dockerfile será el siguiente:



```
1 FROM python:3.5
2
3 WORKDIR /usr/src/app
4
5 RUN pip3 install requests flask pymongo
6
7 COPY main.py ./agent.py
8
9 CMD [ "python3", "agent.py" ]
10
```

La primera línea del FROM se indica la imagen que usará como base, en este caso python:3.5.

WORKDIR se usa para especificar que carpeta se usará para ejecutar todos los comandos, es equivalente a hacer cd carpeta en *NIX.

En la tercera línea se usa el comando RUN para instalar las librerías con pip. Con COPY copiamos de nuestra carpeta del agente el archivo main.py a la carpeta que tenemos especificada en WORKDIR además renombramos a agent.py

Por último especificamos el comando que tendrá nuestra imagen
CMD ["python", "agent.py"].

Construimos y ejecutamos nuevamente docker run enlazando nuestro contenedor de MongoDB con el parámetro --link de Docker.

Construir la imagen del agente

```
$ docker build -t="cryptongo-agent" .
```

Este comando construirá la imagen con el Dockerfile que está en la carpeta y le colocará el nombre de "cryptongo-agent".

Ejecutar la imagen del agente

```
$ docker run -it --link=mongo-crypto:mongo-crypto cryptongo-agent  
Guardando información en Cryptongo
```

Con docker run especificamos -it para tener una consola interactiva de nuestro contenedor y de esta forma se ejecuta en primer plano, si deseamos que sea en segundo plano cambiamos -it por -d como se hizo en el contenedor de MongoDB. En --link especificamos el nombre de nuestro contenedor de MongoDB y el alias que tendrá dentro de nuestro contenedor del agente.

Ya en este punto está en ejecución nuestro agente de CryptoAppMongo y cada 4 minutos pedirá información del API.

Desplegando nuestro API REST

Para nuestra API solo cambiaremos la línea que contiene:

```
db_connection = get_db_connection('mongodb://localhost:27017/')
```

por

```
db_connection = get_db_connection('mongodb://mongo-crypto:27017/')
```

Luego creamos el Dockerfile en la carpeta donde se encuentra el archivo main.py de nuestra API y colocamos el siguiente contenido:

Licenciatura en Sistemas de Información Bases de Datos NSQL

```
Dockerfile
1 FROM python:3.5
2
3 ENV FLASK_APP api.py
4
5 WORKDIR /usr/src/app
6
7 RUN pip3 install requests flask pymongo
8
9 COPY main.py ./api.py
10
11 EXPOSE 5000
12
13 CMD ["flask" , "run"]
14
```

El archivo es muy similar al Dockerfile del agente, pero tiene dos variaciones.

La primera crea una variable de entorno necesaria por Flask para saber que archivo ejecutar, eso se realiza con el comando ENV seguido del nombre de la variable y el contenido.

La segunda modificación es el uso del comando EXPOSE para publicar el puerto 5000 de nuestra imagen, ese es el puerto que usa Flask para escuchar peticiones.

Construcción de la imagen del API

Para construir la imagen ejecutamos docker build cambiando el nombre a cryptongo-api.

```
$ docker build -t="cryptongo-api" .
```

Ejecutando el API de CryptoAppMongo

La ejecución se realiza con docker run y de nuevo enlazamos el contenedor de MongoDB para poder leer la base de datos:

```
$ docker run -it --link=mongo-crypto:mongo-crypto -p 5000:5000 cryptongo-api
```

La diferencia esta vez es que abrimos el puerto 5000 del contenedor para enlazarlo con el puerto 5000 del servidor host, de esta forma se puede ingresar desde el navegador a la dirección IP del servidor con el puerto 5000.

Requerimiento II

Genere una interfaz donde se puede desplegar los datos de una mejor manera UX/UI