

ADMINISTRACIÓN DE SISTEMAS
INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

Proyecto Individual - CouchDB

8 de diciembre de 2020



Alvaro Luzuriaga

Índice de Contenidos

Introducción a CouchDB	1
Tareas obligatorias	3
Aplicación cliente y Dockerfile	3
DockerHub	6
Docker Compose	7
Tareas opcionales	10
Kubernetes	10
Volumen persistente	13
Docker volume	13
Kubernetes volume	16
Vagrant	19
Enlaces	21
Bibliografía	22

Introducción a CouchDB

La aplicación asignada a este proyecto ha sido CouchDB. Como todos sabemos es importante tener un buen sistema gestor de bases de datos (SGBD) para la correcta administración de nuestras las mismas. CouchDB se presenta como una opción sencilla y fácil de manejar en el mundo de las SGBD.

Las bases de datos más conocidas tradicionalmente son las llamadas relacionales. Esta clase de BD, guarda sus registros en función de relaciones comunes. En este caso, CouchDB se nos da a ver como una opción distinta a las ya familiares bases de datos relacionales. Ahora los datos se guardarán en base a documentos, no en tablas o filas, sino en notas independientes cerradas.

Este enfoque centrado en documentos hace en gran medida, más fácil el proceso de desarrollo en. Además, permite que los registros semánticamente similares (por ejemplo, con formatos de archivo iguales), pero que se diferencian sintácticamente unos de otros (en cuanto a estructura externa e interna), se recopilen agrupados.

Estos documentos incluyen uno o más pares campo/valor expresados en JSON. Los valores de los campos pueden ser datos simples como cadenas de caracteres, números o fechas. Aunque también pueden ser listados ordenados o vectores asociativos. Cada uno de estos documentos en una base de datos CouchDB tienen un identificador único y no requieren un esquema determinado.

Es posible interactuar con una BD de CouchDB de diversas maneras. La primera y más sencilla de ellas es por comandos en el terminal. Con el comando *curl* podremos acceder a la IP y puerto de nuestra base de datos, pudiendo en el mismo ejecutar acciones con esta.

Dependiendo de cómo esté configurada nuestra base de datos, deberemos introducir nuestro usuario y contraseña dentro del propio comando. Tenemos que tener en cuenta también en todas ellas que CouchDB utiliza por defecto el puerto 5984.

- Si queremos mostrar todas las bases de datos, el comando sería el siguiente, teniendo que introducir usuario y contraseña:

```
curl -X GET http://admin@admin127.0.0.1:5984/_all_dbs
```

- Si queremos introducir una nueva tabla llamada prueba, el comando sería así, sin introducir usuario y contraseña:

```
curl -X PUT http://127.0.0.1:5984/prueba
```

- Si deseamos actualizar esta última con algún dato nuevo, sin introducir usuario y contraseña::

```
curl -X PUT http://127.0.0.1:5984/reviews/01 -d '{"_nombre":"prueba",  
"fecha":"03-12-2020", "_rev":"1-8ce1d23b7455705c3c2cbeeb86d8ccf5"}'
```

Se ha querido probar el funcionamiento de la BD en un pequeño *script* de *bash*. En este *script*, se querían poder seleccionar, crear y borrar bases de datos a voluntad. Introduciendo una única vez el usuario y contraseña como inicio de sesión, por cada operación.

```
#!/bin/bash

read -p "Introduzca su usuario de CouchDB: " usr
read -p "Introduzca su contraseña de CouchDB: " psw
echo ""
echo "Bienvenido a CouchDB"
echo ""
curl http://127.0.0.1:5984/
echo ""

while true; do
    echo "Opciones:"
    echo "1) GET"
    echo "2) PUT"
    echo "3) DELETE"
    echo ""
    read -p "Seleccione una opcion " op
    case $op in
        #GET
        1) echo "Opción Seleccionada GET"
            echo ""
            curl -X GET http://$usr:$psw@127.0.0.1:5984/_all_dbs
            echo ""
            read -p "Seleccione una base de datos a leer: " get
            curl -X GET http://$usr:$psw@127.0.0.1:5984/$get
            break;;
        #PUT
        [2]* ) echo "Opción Seleccionada PUT"
                echo ""
                read -p "Indique el nombre de la base de datos a crear: " put
                curl -X PUT http://$usr:$psw@127.0.0.1:5984/$put
                break;;
        #DELETE
        [3]* ) echo "Opción Seleccionada DELETE"
                echo ""
                curl -X GET http://$usr:$psw@127.0.0.1:5984/_all_dbs
                echo ""
                read -p "Seleccione una base de datos a leer: " del
                curl -X DELETE http://$usr:$psw@127.0.0.1:5984/$del
                break;;
        * ) echo "Seleccione una opcion .";;
    esac
done

echo "¡Hasta otra!"
```

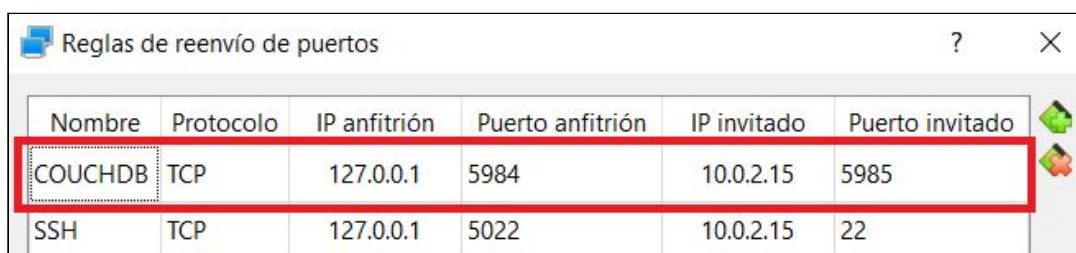
Tareas obligatorias

Aplicación cliente y Dockerfile

Según se ha explicado previamente, es posible acceder y usar CouchDB de distintas maneras. A parte de por comandos, es posible realizarlo por la interfaz que ofrece la propia herramienta, usando comandos o aplicaciones de *Python*, o usando *Java*.

Este proyecto se ha desarrollado principalmente con la interfaz y *Python*. Ya que combinando ambas, puede realizarse todas las operaciones sin problema.

Como primer paso, se ha creado un nuevo reenvío de puertos, ya que este trabajo se está realizando en una máquina virtual de ubuntu. Si no se llega a realizar, no podríamos visualizar la interfaz de la BD por el navegador de nuestro equipo anfitrión. El reenvío de puertos sería el siguiente:



Nombre	Protocolo	IP anfitrión	Puerto anfitrión	IP invitado	Puerto invitado
COUCHDB	TCP	127.0.0.1	5984	10.0.2.15	5985
SSH	TCP	127.0.0.1	5022	10.0.2.15	22

A continuación se ha realizado una aplicación en *Python*, la cual deberá realizar ciertas operaciones sencillas en la BD. Primero hemos realizado unos comandos sencillos en la consola de *Python* para comprobar su funcionamiento. Pero antes de eso tendremos que poner CouchDB en ejecución, con su puerto, usuario y contraseña correspondientes:

```
sudo docker run -p 5985:5984 -e COUCHDB_USER=admin -e COUCHDB_PASSWORD=admin couchdb
```

Al probar `import couchdb` en *Python* vemos que hay problemas de compatibilidad con *python3* y la sintaxis de los comandos. Después de investigar se ha llegado a la conclusión de que se debe utilizar la versión *couchdb2* del cliente, a la vez que los comandos correspondientes. Una vez sabido esto probamos los comandos:

```
python3
>>> import couchdb2
>>> server = couchdb2.Server(href='http://127.0.0.1:5985/',
username='admin', password='admin', use_session=True, ca_file=None)
>>> db = server.create('python')
```

Una vez comprobado su correcto funcionamiento, se ha realizado la siguiente aplicación de cliente:

```
import couchdb2

server = couchdb2.Server(href='http://127.0.0.1:5985/',
username='admin', password='admin', use_session=True, ca_file=None)
#1.
db = server.create('test')
#2.
doc1 = {'_id': 'myid', 'name': 'mydoc', 'level': 4}
db.put(doc1)
doc = db['myid']
assert doc == doc1
#3.
doc2 = {'name': 'another', 'level': 0}
db.put(doc2)
print(doc2)
#4.
db.put_design('mydesign',
              {"views":
               {"name":
                {"map": "function (doc) {emit(doc.name, null);}"}}
              })
result = db.view('mydesign', 'name', key='another', include_docs=True)
assert len(result) == 1
print(result[0].doc)
```

En esta aplicación configuramos que el servidor que se va a utilizar tiene el siguiente enlace y puerto, usuario y contraseña. Después se realizan cuatro acciones.

1. Se crea una tabla *test*.
2. Se añade la tabla *test* una nueva entrada o documento.
3. Se añade la tabla *test* otra nueva entrada o documento, con características diferentes.
4. Se añade la tabla *test* otra nueva entrada o documento, con una estructura diferente.

Mediante esta prueba, a parte del correcto funcionamiento de la aplicación, se quiere demostrar como funciona una base de datos no relacional basada en documentos. Así como mostrar la carencia de dependencias a la hora de introducir o borrar datos en la misma.

Teniendo ya una aplicación funcional, solo nos queda crear el *Dockerfile* para encapsularla en una imagen *Docker*. Antes de ello tendremos que crear un fichero de requisitos para la aplicación del cliente. El cual tendrá únicamente escrito:

```
couchdb2
```

Este *Dockerfile* tiene la siguiente estructura:

```
FROM python:3
WORKDIR /couchdb
RUN apt-get update
RUN apt-get install python3
COPY $pwd/requirements.txt /couchdb/requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5984
COPY $pwd/app-client.py /couchdb/app-client.py
CMD [ "python", "./app-client.py" ]
```

- **FROM:** Imagen base de *python3*
- **WORKDIR:** Cambiar directorio a una carpeta nueva llamada *couchdb*
- **RUN:** Ejecutar la instalación de *python3* y los requisitos de la aplicación
- **COPY:** Copiar la aplicación de cliente y los requisitos al directorio nuevo
- **EXPOSE:** Exponer puerto 5984
- **CMD:** Ejecutar la aplicación de cliente

Lo montamos, y una vez hecho esto lo ejecutamos con el ID que tenga el contenedor. El parámetro `--net='host'` es necesario para poder ver la red anfitrión y que se ejecute correctamente:

```
sudo docker build .
sudo docker run --net='host' <ContainerID>
```

Una vez ejecutado, podremos comprobar los cambios en la siguiente ruta: http://localhost:5984/_utils/#. Cabe recordar activar el modo “Single Node” si el comando `docker run` se queda en bucle dando error, entrando en la pestaña de la “llave inglesa” dentro de la propia interfaz de *Couchdb*.

Databases					Database name	Create Database	{ }JSON
	Name	Size	# of Docs	Partitioned	Actions		
	<code>_replicator</code>	2.3 KB	1	No			
	<code>_users</code>	2.3 KB	1	No			
	<code>test</code>	0 bytes	0	No			

DockerHub

Ya teniendo creada la imagen de la aplicación cliente, vamos a proceder a subirla a *DockerHub*.

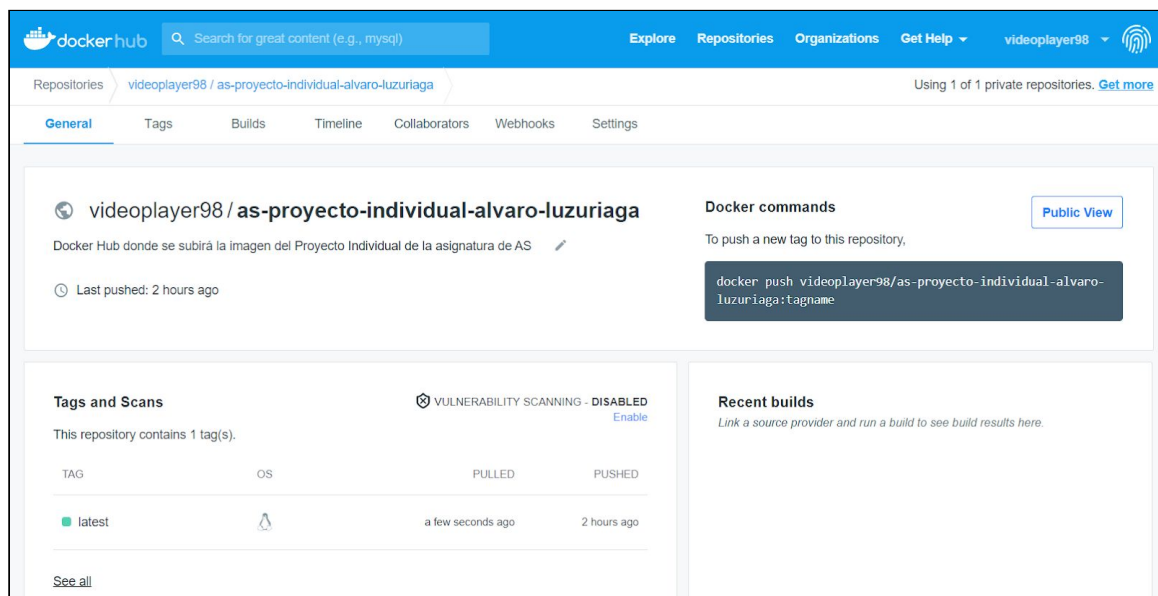
Nos dirigimos a la *web* → *Repositories* → *Create Repository*. Creamos un repositorio con el nombre “as-proyecto-individual-alvaro-luzuriaga” de tipo público e iniciamos sesión:

```
sudo docker login --username=videoplayer98
```

Ahora procederemos a etiquetar la imagen, construirla y subirla a nuestro repositorio:

```
sudo docker build -t videoplayer98/as-proyecto-individual-alvaro-luzuriaga .  
sudo docker push -t videoplayer98/as-proyecto-individual-alvaro-luzuriaga .
```

Como se puede comprobar, se ha subido sin problemas al repositorio: **videoplayer98/as-proyecto-individual-alvaro-luzuriaga**



Docker Compose

En el último apartado de las tareas obligatorias debemos generar un entorno *Docker Compose*, el cual estará ejecutándose tanto la aplicación asignada como la aplicación de cliente. Para ello deberemos realizar algunos cambios en esta última.

Antes de nada, sería prudente limpiar todas las imágenes de *couchdb* activas, para no sufrir conflictos a la hora de construir el *Docker Compose*:

```
sudo docker system prune
sudo docker images
sudo docker image rm couchdb
sudo docker image rm couchdb:2.1.1
```

Pasamos a crear nuestro *Compose*, que tendrá la siguiente estructura:

```
version: "3"
services:
  couchdb:
    container_name: couchdb
    image: couchdb:latest
    restart: always
    ports:
      - 5985:5984
    environment:
      - 'COUCHDB_USER=admin'
      - 'COUCHDB_PASSWORD=admin'
  app-client:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 5986:5984
    depends_on:
      - couchdb
```

- **versión:** usaremos la versión 3 de *Compose*.
- **services:** tendremos dos servicios que estarán encapsulados en el *Compose*. El primero, *couchdb*, será la aplicación asignada, siendo *app-client* nuestra aplicación de *python* creada, lanzada desde el *Dockerfile*. Ambos podrán verse uno al otro ya que están lanzados desde el mismo *Docker Compose*.
- **couchdb:**
 - **image:** usaremos la imagen más reciente de *couchdb*.
 - **restart:** siempre que iniciemos el servicio de *couchdb* desde el *Compose*, reiniciamos el estado del mismo.
 - **ports:** redirige el puerto 5984 en el contenedor al 5985 del anfitrión.
 - **environment:** asignamos el usuario y contraseña de inicio de sesión para *couchdb*, en nuestro caso *admin* para el usuario y *admin* para la contraseña.
- **couchdb:**
 - **build:** construiremos el *Dockerfile* alojado en el mismo directorio que el *Compose*, con el nombre "*Dockerfile*"
 - **ports:** redirige el puerto 5984 en el contenedor al 5986 del anfitrión.
 - **depends_on:** haremos que este servicio, dependa de *couchdb*. Es decir, que no iniciará su ejecución hasta que el anterior se ejecute. Debe ser así, ya que necesitamos la base de datos en marcha antes de realizar cambios en ella mediante la aplicación de cliente.

Una vez ejecutado el *Docker Compose*, vemos que *CouchDB* se inicia correctamente, pero no que los cambios realizados por la aplicación de cliente se procesan. Esto es debido a que de alguna manera *depends_on* no funciona como debería.

Sabiendo esto, hemos añadido un bucle en la aplicación de cliente, con el cual hacemos que el acceso a base de datos no proceda hasta que esta esté finalmente en marcha:

```
import couchdb2
import time

retries = 10
while True:
    try:
        couchdb_conex()
    except couchdb.exceptions.ConnectionError as exc:
        if retries == 0:
            raise exc
        retries -= 1
        time.sleep(10)
```

```
def couchdb_conex():
    server = couchdb2.Server(href='http://couchdb:5984/', username='admin',
                             password='admin', use_session=True, ca_file=None)
    db = server.create('test')

    doc1 = {'_id': 'myid', 'name': 'mydoc', 'level': 4}
    db.put(doc1)
    doc = db['myid']
    assert doc == doc1

    doc2 = {'name': 'another', 'level': 0}
    db.put(doc2)
    print(doc2)

    db.put_design('mydesign',
                  {"views":
                   {"name":
                    {"map": "function (doc) {emit(doc.name, null);}"
                   }
                  })
    result = db.view('mydesign', 'name', key='another', include_docs=True)
    assert len(result) == 1
    print(result[0].doc)
```

Habiendo solucionado este problema, vemos que la base de datos se pone en marcha y los cambios se suceden de manera esperada.

Databases				Database name	Create Database	{ } JSON
Name	Size	# of Docs	Partitioned	You have been logged in.		
_replicator	2.3 KB	1	No			
_users	2.3 KB	1	No			
test	1.1 KB	3	No			

Tareas opcionales

Kubernetes

En esta aplicación de *Kubernetes* usaremos objetos *Deployment* para ejecutar la aplicación asignada y la aplicación cliente. Usaremos el mismo *Dockerfile* y *app-client.py* que hemos usado anteriormente en el despliegue de *Docker*.

Para realizar este apartado deberemos tener compilado el *Dockerfile* con nuestro usuario y el “tag” que queramos:

```
sudo docker build -t videoplayer98/as-proyecto-individual-alvaro-luzuriaga .
```

Una vez compilado lo subimos a Docker Hub con el “tag” completo:

```
sudo docker push videoplayer98/as-proyecto-individual-alvaro-luzuriaga
```

Ahora creamos el fichero de *Kubernetes* para el despliegue de la aplicación. De tipo *Deployment*, llamado *client-deployment.yml* :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: couchdb
spec:
  replicas: 1
  selector:
    matchLabels:
      component: web
  template:
    metadata:
      labels:
        component: web
    spec:
      containers:
        - name: client
          image: videoplayer98/as-proyecto-individual-alvaro-luzuriaga
          ports:
            - containerPort: 5984
```

- **apiVersion:** indicamos una versión de la API a utilizar, en este caso *apps/v1*
- **kind:** indicamos el tipo de objeto, como se especifica en el enunciado, será de tipo *Deployment*.
- **spec:**
 - **replica:** *Kubernetes* usa este controlador para asegurarse de que se ejecutan el número exacto de *Pod* indicados en ejecución. En este caso únicamente hay uno.
 - **selector:** en el selector se agrupan los elementos básicos de agrupación en *Kubernetes*. Se utilizan para seleccionar un conjunto de objetos. En nuestro caso el componente “web”.
 - **template:** describimos la configuración de los *Pod* que se van a crear.
 - **metadata:** utilizamos este apartado para comprender la forma en que se organizan los contenedores en sus numerosos servicios, máquinas etc. En este caso, solo tendremos el componente “web”
 - **spec:** especificamos qué nombre tendrá el contenedor (*client*) que imagen usará (la definida anteriormente en *DockerHub*) y el puerto del contenedor (5984).

Aplicamos la nueva configuración del fichero *Deployment* :

```
kubectl apply -f client-deployment.yml
```

Creamos un *NodePort*, para poder configurar manualmente el reenvío de puertos. Así poder exponer un puerto del contenedor al exterior del cluster. Tendrá el nombre *client-node-port.yml* :

```
apiVersion: v1
kind: Service
metadata:
  name: client-node-port
spec:
  type: NodePort
  ports:
    - port: 3050
      targetPort: 5985
      nodePort: 31000
  selector:
    component: web
```

- **apiVersion:** indicamos una versión de la API a utilizar, en este caso *v1*
- **kind:** indicamos el tipo de objeto, como se especifica en el enunciado, será de tipo *Service*.
- **spec:**
 - **type:** especificamos el tipo de servicio, siendo en este caso un *NodePort*
 - **ports:**
 - **port:** el puerto que dentro del *cluster* expone el servicio, es el 3050.
 - **targetPort:** indicamos el puerto de conexión con los pods, aquí será el 5985.
 - **nodePort:** indicamos el puerto de conexión con el exterior, es decir, que nodo ofrece el servicio fuera del POD. En este caso el 31000.
 - **selector:** en el selector se agrupan los elementos básicos de agrupación en Kubernetes. Se utilizan para seleccionar un conjunto de objetos. En nuestro caso el componente “web”.

Aplicamos la nueva configuración del fichero *NodePort*.

```
kubectl apply -f client-node-port.yaml
```

Accedemos al siguiente enlace para comprobar el correcto funcionamiento del despliegue de Kubernetes, http://localhost:31000/_utils/#.

Volumen persistente

Vamos a hacer que tanto *Docker* como *Kubernetes* ejecuten la aplicación cliente. Esta aparte de su función original, realizará también unas escrituras en ficheros.

Docker volume

Docker provee tres maneras de montar datos en un contenedor: volúmenes, *bind mounts* y *tmpfs*. Estos últimos al no ser persistentes, hemos decidido utilizar volúmenes para realizar este apartado.

Los volúmenes pueden ser creados tanto por el comando `docker volume create`, como inicializando un contenedor.

Como primer paso, crearemos un volumen llamado *virtualenv* que sirva de ruta para guardar entornos virtuales.

```
docker volume create virtualenv
[{
  "CreatedAt": "2020-12-05T22:57:48Z",
  "Driver": "local",
  "Labels": {},
  "Mountpoint": "/var/lib/docker/volumes/virtualenv/_data",
  "Name": "virtualenv",
  "Options": {},
  "Scope": "local"
}]
```

Una vez realizado, procedemos a crear el contenedor, el cliente y los scripts necesarios. Al fichero *app-client.py*, le añadiremos unas líneas que escriban como se ha mencionado anteriormente, texto en unos ficheros. En nuestro caso también comprobará si el entorno virtual “my_env” existe, y sino lo creará. Vamos a montar el volumen creado más arriba como la carpeta `~/.virtualenv` en el contenedor.

El *Dockerfile* sería similar al anteriormente usado, pero instalando *virtualenv*.

```
FROM python:3
WORKDIR /couchdb
ADD . /couchdb
RUN apt-get update
RUN apt-get install python3
RUN pip install virtualenv
RUN pip install -r requirements.txt
EXPOSE 5984
CMD [ "python", "./app-client.py" ]
```

La aplicación *app-client.py* quedaría de esta forma.

```
import couchdb2
import os
import subprocess

server = couchdb2.Server(href='http://127.0.0.1:5985/', username='admin',
password='admin', use_session=True, ca_file=None)

db = server.create('test')

doc1 = {'_id': 'myid', 'name': 'mydoc', 'level': 4}
db.put(doc1)
doc = db['myid']
assert doc == doc1

doc2 = {'name': 'another', 'level': 0}
db.put(doc2)
print(doc2)

db.put_design('mydesign',
              {"views":
               {"name":
                {"map": "function (doc) {emit(doc.name, null);}"}}
              })
result = db.view('mydesign', 'name', key='another', include_docs=True)
assert len(result) == 1
print(result[0].doc)
volumen()

def volumen():

    # Creacion de volumen persistente
    if os.path.exists('/root/.virtualenv/my_env'):
        print('my_env ya existe')
    else:
        subprocess.run(['bash', 'create_env.sh'])
        print('my_env creado')

    # Escritura de ficheros
    f= open("fichero1.txt","w+")
    for i in range(2):
        f.write("Este es el fichero 1, linea numero %d\r\n" % (i+1))
    f.close()
    f = open("fichero2.txt", "a")
    f.write("Este es el fichero 2 \n")
    f.close()
```

Con este código, generamos dos ficheros que en cada uno se escribirán distintas líneas, así como comprobar que el volumen ha sido creado. El script que se ejecuta sería el siguiente.

```
cd ~/.virtualenv/ && virtualenv my_env
```

También hará falta un nuevo fichero con los requisitos para instalar las nuevas dependencias.

```
couchdb2  
os  
subprocess
```

Ahora que ya tenemos todos los archivos necesarios, construimos la imagen Python.

```
docker build -t docker-data-persistence .
```

Montamos el volumen usando el argumento `--mount`.

```
docker run --mount source=virtualenv,target=/root/.virtualenv  
docker-data-persistence
```

Dependiendo de si se ejecuta por primera vez o no, el mensaje devuelto por pantalla cambiará. Ahora podemos mirar dentro de los ficheros del volumen, para verificar el contenido.

```
docker run -it --mount source=virtualenv,target=/root/.virtualenv  
docker-data-persistence find /root/.virtualenv/my_env/bin
```

Si deseamos que el volumen se borre, lo haremos con el siguiente comando. Sin embargo, no podrás borrarlo cuando haya un contenedor que lo use.

```
docker volume rm virtualenv
```

Kubernetes volume

Para realizar el volumen persistente en *Kubernetes*, haremos unos cambios tanto en el *app-client.py* como en el *client-deployment.yml*. A su vez tendremos que crear un fichero llamado *persistent-volume-claim.yml* para configurar el volumen persistente.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: persistent-volume-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2G
  hostPath:
    path: "/couchdb/volumen"
```

- **apiVersion:** indicamos una versión de la API a utilizar, en este caso *v1*
- **kind:** indicamos el tipo de objeto, como se especifica en el enunciado, será de tipo *PersistentVolumeClaim*.
- **metadata:** decimos que sea de creación manual.
- **spec:**
 - **storageClassName:** especificamos el tipo de servicio, que será un *NodePort*.
 - **accessModes:** acceso específico a un nodo.
 - **resources:** la capacidad de almacenamiento será de 2 GB.
 - **hostPath:** Montaje en la ruta local “/couchdb/volumen”.

Aplicamos la creación del volumen.

```
kubectl apply -f persistent-volume-claim.yml
```

Ahora añadiremos la solicitud de volumen persistente a la configuración del fichero *client-deployment.yml*. Este tendrá el siguiente aspecto.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: couchdb
spec:
  replicas: 1
  selector:
    matchLabels:
      component: web
  template:
    metadata:
      labels:
        component: web
    spec:
      containers:
        - name: client
          image: videoplayer98/as-proyecto-individual-alvaro-luzuriaga
          ports:
            - containerPort: 5984
          volumeMounts:
            - name: couchdb-storage
              mountPath: /couchdb/volumen
              subPath: couchdb
      volumes:
        - name: couchdb-storage
          persistentVolumeClaim:
            claimName: persistent-volume-claim
```

- **spec:**
 - **template:**
 - **spec:**
 - **volumeMounts:** aquí indicamos la configuración del volumen persistente. Señalamos el nombre del volumen (*couchdb-storage*), el punto de montaje en el contenedor (*/couchdb/volumen*) y la ruta donde guardar los datos en el volumen (*couchdb*).
 - **volumes:** en este apartado está la configuración de la solicitud de volumen persistente. Señalamos el nombre del volumen (*couchdb-storage*) y el nombre de la solicitud (*couchdb-storage*).

Añadimos la configuración al cluster.

```
kubectl apply -f client-deployment.yml
```

El fichero `app-client.py` será similar al usado en `Docker`, pero retirando las líneas de comprobación del volumen `Docker` creado, ya que eso lo haremos manualmente.

```
import couchdb2

server = couchdb2.Server(href='http://127.0.0.1:5985/', username='admin',
password='admin', use_session=True, ca_file=None)

db = server.create('test')

doc1 = {'_id': 'myid', 'name': 'mydoc', 'level': 4}
db.put(doc1)
doc = db['myid']
assert doc == doc1

doc2 = {'name': 'another', 'level': 0}
db.put(doc2)
print(doc2)

db.put_design('mydesign',
              {"views":
               {"name":
                {"map": "function (doc) {emit(doc.name, null);}"}}
              })
result = db.view('mydesign', 'name', key='another', include_docs=True)
assert len(result) == 1
print(result[0].doc)
volumen()

def volumen():
    # Escritura de ficheros
    f= open("fichero1.txt","w+")
    for i in range(2):
        f.write("Este es el fichero 1, linea numero %d\r\n" % (i+1))
    f.close()
    f = open("fichero2.txt", "a")
    f.write("Este es el fichero 2 \n")
    f.close()
```

Finalmente, verificamos la creación del volumen persistente, listando los volúmenes.

```
kubectl get pv
```

Vagrant

En esta tarea opcional procederemos a realizar un despliegue equivalente al usado con contenedores, pero empleando *Vagrant*. Tendremos que crear una máquina virtual en la que se instale la aplicación asignada y la aplicación cliente, sin utilizar contenedores.

Antes de nada comprobaremos que tenemos *Vagrant* instalado en nuestra máquina anfitrión, en mi caso en Windows.

```
vagrant version
```

Añadimos la 'box' requerida a *Vagrant*, para así poder configurarla y poner en marcha la aplicación. En nuestro caso la añadiremos a *Virtualbox*. Así que pulsamos la opción dos para instalarlo ahí.

```
vagrant box add ubuntu/trusty64
```

Creamos un entorno propio en el directorio que ejecutemos el comando.

```
vagrant init
```

Aquí se generará el *Vagrantfile*, en el cual configuraremos nuestro entorno de *Vagrant*. Este tendrá el siguiente contenido.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
Vagrant.configure("2") do |config|

  config.vm.box = "ubuntu/trusty64"
  config.vm.network "forwarded_port", guest: 5984, host: 5985, host_ip: "127.0.0.1"
  config.vm.synced_folder "C:/Users/Alvaro/Desktop", "/home/vagrant/shared_folder"

  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = 2
  end

  config.vm.provision "shell", inline: <<-SHELL
    sudo apt-get update
    sudo apt-get install -y couchdb
    sudo service couchdb restart
  SHELL
end
```

En este fichero definimos qué configuración y funciones tendrá *Vagrant*.

- **Vagrant.configure:** decimos qué configuración y funcione de *Vagrant* a ejecutar.
- **config.vm.box:** indicamos el “box” a utilizar, es decir la imagen de la consola virtual, en este caso *ubuntu/trusty64*.
- **config.vm.network:** indicamos el reenvío de puertos de invitado y anfitrión, así como la IP de anfitrión.
- **config.vm.synced_folder:** indicamos la carpeta compartida entre el anfitrión y la máquina.
- **config.vm.provider:** seteamos las especificaciones de la máquina, así como la cantidad de memoria a utilizar y el número de núcleos.
- **config.vm.provision:** indicamos los comandos en shell que queremos que se ejecuten en *Vagrant*. En nuestro caso queremos instalar e iniciar el servicio de *Couchdb*.

Arrancamos la consola virtual anteriormente creada desde consola.

```
vagrant up
```

Nos conectamos a la máquina virtual por medio de *ssh*.

```
vagrant ssh
```

Nos metemos en la carpeta compartida y creamos el fichero de la aplicación cliente, en este caso el mismo *app-client.py* que hemos utilizado en el primer apartado de *Dockerfile*. Lo ejecutamos y al tener en marcha el servicio *couchdb* los cambios proceden sin problemas.

```
cd shared_folder/  
vim app-client.py  
python app-client.py
```

Accedemos al enlace del anfitrión para comprobarlo, http://localhost:5984/_utils/#.

Enlaces

Aquí dejamos los enlaces referentes a los archivos, ficheros e imágenes usadas en el proyecto. Se han subido los archivos tanto a GitHub como a Drive por si surgen problemas al acceder a alguna. En DockerHub se encuentra la imagen de la aplicación cliente.

- Drive:

https://drive.google.com/drive/folders/1JsNTPqkAiiOu_Rg5YX4XS7C4q6iHQyIM?usp=sharing

- GitHub:

<https://github.com/AlvaroLuzu/as-proyecto-individual-alvaro-luzuriaga>

- DockerHub:

<https://hub.docker.com/repository/docker/videoplayer98/as-proyecto-individual-alvaro-luzuriaga>

Bibliografía

CouchDB

Docker Official Images - docker.com

https://hub.docker.com/_/couchdbino

Presentación de CouchDB - ionos.es

<https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/presentacion-de-couchdb/>

Working With CouchDB From the Command Line - ionos.es

<https://www.ionos.com/community/hosting/couchdb/working-with-couchdb-from-the-command-line/>

CouchDB not running on Docker image - stackoverflow.com

<https://stackoverflow.com/questions/57989678/couchdb-not-running-on-docker-image>

Python

Python en CouchDB - couchdb-python.readthedocs.io

<https://couchdb-python.readthedocs.io/en/latest/>

Interfacing CouchDB with Python - opensourceforu.com

Dr Kumar Gaurav - Amit Doegar - July 8, 2015

<https://www.opensourceforu.com/2015/07/interfacing-couchdb-with-python-2/>

CouchDB2 1.9.3 - pypi.org

<https://pypi.org/project/CouchDB2/>

JavaScript

Building an offline-first app with React and CouchDB - manifold.co

<https://manifold.co/blog/building-an-offline-first-app-with-react-and-couchdb>

Python CouchDB Connectivity - javatpoint.com

<https://www.javatpoint.com/python-couchdb>

How to develop your first CouchDB application with CouchApp and JS - medium.com

Jaquie - Julio 23, 2019

<https://medium.com/ksquare-inc/how-to-develop-your-first-couchdb-application-with-couchapp-and-js-e078271eb455>

Bash Script

How to read two input values in bash? - stackoverflow.com

<https://stackoverflow.com/questions/54730123/how-to-read-two-input-values-in-bash>

Bash Script con Menú de Opciones - medium.com

Samuel Vargas - Oct 29, 2019

<https://medium.com/linux-tips-101/bash-script-con-menu-de-opciones-4371e05f4e0f>

Kubernetes

deployment-couchdb.yaml - gist.github.com

<https://gist.github.com/kocolosk/d4bed1a993c0c506b1e58274352b30df>

Volúmenes

How to persist data in docker container - dev.to

Jibin Liu - Sept 15, 2018

<https://dev.to/jibinliu/how-to-persist-data-in-docker-container-2m72>

Persistent Volumes - kubernetes.io

<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>