Readme.md

This is a simple application which is aimed at understanding and putting together various pieces of the puzzle.

**Overview:**

In this application, we use flask for creating the web app with connexion and swagger-ui for API documentation. We create RESTful API for requests and use a fast in-memory Redis server for database needs. JSON is used for the data interchange and is serialized and deserialized using python json module. Nginx acts as a reverse proxy for this app and Docker with user defined networks are used to containerize the application.

Let’s build this project in bottom up approach:

The Flask, Python to build a RESTful API which can make HTTP calls to server to GET/PUT data to populate the various dynamic portions of application. Swagger is used in addition to provide useful documentation. In this we also have a small single paged web application which is used to demonstrate the API usage with Javascript HTML and CSS.

The idea behind the API is to isolate data from the application which uses it and encapsulating the implementation details of data from application.

**Flask**:

Flask is a lightweight [WSGI](https://wsgi.readthedocs.io/) web application framework (WSGI is the Web Server Gateway Interface. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request [3]). It is designed to make getting started quick and easy, with the ability to scale up to complex applications. [2]

**RESTful API**: It is one which is based on client-server architecture and is stateless. The main abstraction is RESOURCE. A resource can be anything from a document, an image, a collection of resources, a non-virtual object (e.g. a person), etc. REST uses a **resource identifier** to identify the particular resource involved in an interaction between components.

State of a resource at a given time is called resource representation. A representation can have data, metadata and **hypermedia** links. Then there are resource methods which are used to perform the desired transition. Majority of people use HTTP **GET/PUT/POST/DELETE** methods, however, any uniform interface can also be RESTful.

In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs). The resources are acted upon by using a set of simple, well-defined operations. The clients and servers exchange representations of resources by using a standardized interface and protocol – typically HTTP. [1]

**People REST API:**

For the demo application, we have a REST API providing access to a collection of people with CRUD access to an individual person within that collection. Here’s the API design for the people collection:

| **Action** | **HTTP Verb** | **URL Path** | **Description** |
| --- | --- | --- | --- |
| Create | POST | /api/people | Defines a unique URL to create a new person |
| Read | GET | /api/people | Defines a unique URL to read a collection of people |
| Read | GET | /api/people/Farrell | Defines a unique URL to read a particular person in the people collection |
| Update | PUT | /api/people/Farrell | Defines a unique URL to update an existing order |
| Delete | DELETE | /api/orders/Farrell | Defines a unique URL to delete an existing person |

Now our server.py file looks like this:

*from flask import render\_template*

*import connexion*

*# Create the application instance*

*app = connexion.App(\_\_name\_\_, specification\_dir='./')*

*# Read the swagger.yml file to configure the endpoints*

*app.add\_api('swagger.yml')*

*# Create a URL route in our application for "/"*

*@app.route('/')*

*def home():*

*return render\_template('home.html')*

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

*app.run(host='0.0.0.0', port=5000, debug=True)*

Here, we have imported the Flask module, giving the application access to the Flask functionality. We then created a Flask application instance, the app variable. Next, it is connected to the URL route '/' to the home() function using @app.route('/'). This function calls the Flask render\_template() function to get the *home.html*file from the templates directory and returns it to the client browser.

Further Connexion is used to add REST API endpoints. The Connexion module allows a Python program to use the [Swagger](https://swagger.io/) specification. This provides: validation of input/output data to and from your API, easy configuration of the API URL endpoints and the expected parameters, and a really nice UI interface to work with the created API and explore it. app.add\_api('swagger.yml') tells to read the file swagger.yml from the specification directory and configure the system to provide the Connexion functionality.

The file swagger.yml is a YAML or JSON file containing all of the information necessary to configure the server to provide input parameter validation, output response data validation, URL endpoint definition, and the Swagger UI. Here is the *swagger.yml*file defining the GET /api/people endpoint your REST API will provide:

Portion of the *swagger.yml* looks like:

swagger: "2"

info:

description: This is the swagger file that goes with our server code

version: "1.0"

title: Swagger ReST Article

consumes:

- application/json

produces:

- application/json

basePath: /api

# Paths supported by the server application

paths:

/people:

get:

operationId: people.read\_all

tags:

- People

summary: Read the entire list of people

description: Read the list of people

parameters:

- name: length

in: query

type: integer

description: Number of people to get from people

required: false

- name: offset

in: query

type: integer

description: Offset from beginning of list where to start gathering people

required: false

responses:

200:

description: Successfully read people list operation

schema:

type: array

items:

properties:

fname:

type: string

lname:

type: string

timestamp:

type: string

paths tell the beginning of where all the API URL endpoints are defined. The /people value indented under that defines the start of where all the /api/people URL endpoints will be defined. The get: indented under that defines the section of definitions associated with an HTTP GET request to the /api/people URL endpoint. This goes on for the entire configuration and the file is quite self-explanatory.

Now we need to have a handler for people endpoint. In connexion configuration we have people module and the read function within the module when the API gets an HTTP request for GET /api/people. This means a people.py module must exist and contain a read() function.

We also have a single page web application demonstrating the use of the API. This will all be handled by AJAX calls from JavaScript to the people API URL endpoints. Home.html file pulls in the external [normalize.min.css](https://necolas.github.io/normalize.css/) file, which is a CSS reset file to normalize the formatting of elements across browsers. It also pulls in the external [jquery-3.3.1.min.js](https://jquery.com/) file to provide the jQuery functionality you’ll use to create the single-page application interactivity.

Redis: As described on its website:

*Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets.*

The app makes use of Redis as in memory data structure store in which information of persons can be stored. It can also be written to disk using command bgsave(). The python module redis-py is used for making interaction between our app and redis possible.

Redis is preferred because it is open source, very fast in memory database and provides several features over other NoSQL databases.

**Docker:**

As described officially:

***Docker****is a set of coupled*[*software-as-a-service*](https://en.wikipedia.org/wiki/Software-as-a-service)*and*[*platform-as-a-service*](https://en.wikipedia.org/wiki/Platform-as-a-service)*products that use*[*operating-system-level virtualization*](https://en.wikipedia.org/wiki/Operating-system-level_virtualization)*to develop and deliver software in packages called*[*containers*](https://en.wikipedia.org/wiki/Container_(virtualization)).

Containers are isolated from one another and bundle their own software, [libraries](https://en.wikipedia.org/wiki/Library_(computing)) and configuration files; they can communicate with each other through well-defined channels.

We have used docker containers for the application. One docker container consist of nginx image pulled from already existing ones, the next container has the flask app which basically renders the content on the web browser. The next container is the database container which is only connected to the flask container and has no access to the nginx directly. The nginx container listens to the requests from outside world.

**The project app outline: -**

User defined bridge networks

web\_nw

db\_nw

REST API

Port 5000

Port 6379

Port 80

REDIS

Flask App

Nginx

Containers

Port 8080

The Nginx is acting as the reverse proxy for the flask server and the external requests are handled by it. There are 2 user defined networks namely web\_nw and db\_nw which are used to maintain the communication within the clusters. The user defined networks provide a feature of IP address mapping from names to respective IP address.

In the nginx configuration file we have:

server {

listen 80;

server\_name localhost;

location / {

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

proxy\_set\_header Host $http\_host;

proxy\_pass http://flaskapp:5000;

}

}

The nginx listens on port 80 of the user defined bridge network “web\_nw” and it is mapped to 8080 to the outside world (- "8080:80"). The flaskapp is also connected to the web\_nw network and listens on port 5050 and the IP assigned to flaskapp is (172.20.0.3/16). The user defined networks provide the functionality to resolve IP address by container name.

So the incoming requests @ port 8080 from nginx travel to port 80 on web\_nw to <http://flaskapp:5000>.

Now the flask app needs to connect to the redis which is on another user defined network db\_nw and listens on port 6379 and has a dedicated IP assigned to container named m-nginx\_db\_1which is built using already available redis image: redis-alpine. In the people.py file defining the functionality of REST API we have connection established to redis container:

r=redis.StrictRedis(db=1,host=”m-nginx\_db\_1”, port=6379…)

So the RESTAPI (people.py) has all the information regarding IP and port to connect to the redis listening @port 6379 of db\_nw bridge network.

Docker compose is used to build and run the application:

The following commands are needed to be run if you have docker and docker compose installed:

$ docker-compose up -d db

$ docker-compose run --rm flaskapp /bin/bash -c "cd /opt/services/flaskapp/src && python server.py"

$ docker-compose up -d

Now, browse to localhost:8080 to see the app in action.

[1] <https://restfulapi.net/>

[2] <https://palletsprojects.com/p/flask/>

[3] <https://wsgi.readthedocs.io/en/latest/what.html>

[4] <https://realpython.com/flask-connexion-rest-api/>

[5] <https://github.com/juggernaut/nginx-flask-postgres-docker-compose-example>

[6] <http://www.ameyalokare.com/docker/2017/09/14/docker-migrating-legacy-links.html>