CISC 5550 HW4

> Improve “to-do list” application’s design by using web service. To be more specific, leave the application front-end and overall structure unchanged, but provide the data service through a typical RESTful API instead of supporting directly by a database as in the solution posted for the previous homework.

Design

Package

```
from flask import Flask, render_template, redirect, g, request, url_for,
jsonify
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
import os
import requests
import json
```

Workflow

- Using templates to initiate the web application
- Fetching a record from the database and saved as json format
- Updating a record in the database
- Deleting a record in the database

Implementation

Initialization

We import all the modules needed and then create a web application using Flask. And we bind SQLALchemy and Marshmallow into our application.

```
DATABASE = 'todolist.db'

app = Flask(__name__)
app.config.from_object(__name__)
db = SQLAlchemy(app)
ma = Marshmallow(app)
```

After that, we declare our models. This is important, especially when the homework asks us to read data from json format we saved instead of reading HTML from database.

```
class Item(db.Model):
  id = db.Column(db.Integer, primary_key = True)
  what_to_do = db.Column(db.String(80), unique = True)
  due_date = db.Column(db.String(120), unique = True)

  def __init__(self, what_to_do, due_date):
      self.what_to_do = what_to_do
      self.due_date = due_date

class ItemSchema(ma.Schema):
  class Meta:
      fields = ('what_to_do', 'due_date')
```

Get a record and saved as json

I implement this with the Requests package.

```
@app.route("/")
def show_list():
  database = get_db()
  url = 'http://localhost:6000/api/items'
  try:
      r = requests.get(url)
      json_dict = json.loads(json.dumps(r.json()))
      return render_template('index.html', todolist = json_dict)
  except:
      raise ApiError('GET /items/ {}'.format(r.statue_code))
```

Convert the operation of database

This part defined structure of response of our endpoint. We want that all of our endpoint will have JSON response. Here we define that our JSON response will have two keys(what*to*do, and due*date). Also we defined item*schema as instance of ItemSchema, and item_schemas as instances of list of ItemSchema.

```
item_schema = ItemSchema()
items_schema = ItemSchema(many = True)
```

POST function

```
# endpoint to create new item
```

```
@app.route("/add/item", methods = ['POST'])
def create_item():
  what_to_do = request.json['what_to_do']
  due_date = request.json['due_date']

  new_item = Item(what_to_do, due_date)

  db.session.add(new_item)
  db.session.commit()

  return jsonify(new_item)
```

GET function

```
# endpoint to show all items
@app.route("/api/item", methods=["GET"])
def get_items():
    all_items = Item.query.all()
    result = items_schema.dump(all_items)
    return jsonify(result.data)

# endpoint to get item detail by id
@app.route("/api/item/<id>", methods=["GET"])
def get_item(id):
    item = Item.query.get(id)
    return item_schema.jsonify(item)
```

PUT function

```
# endpoint to update item
@app.route("/update/item/<id>", methods=["PUT"])
def update_item(id):
    item = Item.query.get(id)
    what_to_do = request.json['what_to_do']
    due_date = request.json['due_date']

    item.due_date = due_date
    item.what_to_do = what_to_do
```

```
    db.session.commit()
    return item_schema.jsonify(item)
```

DELETE function

```
# endpoint to delete item
@app.route("/delete/item/<id>", methods=["DELETE"])
def item_delete(id):
    item = Item.query.get(id)
    db.session.delete(item)
    db.session.commit()

    return item_schema.jsonify(item)
```

Running the application

```
if __name__ == '__main__':
  app.run(host='0.0.0.0', debug=True)
```

> Using Docker container technology and kubernetes, deploy the 'to-do list' app to a cluster on Google Cloud Platform. You need to describe how you create the Docker image, e.g. using a*Dockerfile*, and how you create the cluster and deploy the containers to it, e.g. using a script (bash script on Linux/Mac, or batch file on Windows) that includes all the relevant commands for this procedure.

Workflow

1. Package the web application into docker image
2. Upload the image into the register
3. Create a cluster for container
4. Deploy the application into the cluster

Preparation

```
gcloud config configurations active my_configuration
gcloud auth active-service-account
gcloud config set project cisc5550
gcloud config set compute/zone us-central1-b
gcloud components install kubectl
gcloud compute components update
```

Package the web application into docker image

```
export PROJECT_ID="$(gcloud config get-value project -q)"
docker build -t gcr.io/${cisc5550}/flask-app:v2
docker image
```

Upload the docker image

```
gcloud auth configure-docker
docker push gcr.io/${cisc5550}/flask-app:v2
docker run --rm -p 6000:6000 gcr.io/${cisc5550}/flask-app:v2
curl http://localhost:6000
```

Create a cluster for container

```
gcloud container clusters create flask-cluster \ --zone us-central1-a \ --num-
nodes=3
gcloud compute instances list
gcloud container clusters get-credentials flask-cluster
```

Deploy the application into the cluster

```
kubectl run flask-web --image=gcr.io/${cisc5550}/flask-app:v2 --port 6000
kubectl get pods
kubectl expose deployment flask-web --type=LoadBalancer --port 80 --target-port
6000
kubectl get service
kubectl scale deployment flask-web --replicas=3
# delete the cluster
gcloud container clusters delete flask-cluster
```