Queen Mary
University of London

ECS781P: Cloud Computing
Lab Instructions for Week 8

Basics of App development, Designing RESTful APIs

Dr. Arman Khouzani

Feb 28, 2017

Preparation[1]

1. Open a terminal. Make sure you are logged in as part of the `ECS781P` group by running `id`. If not, execute: `newgrp ECS781P`. For better visuals, modify your `ls` as follows: `alias ls='ls --color=auto'` (now your `ls` command for seeing a list of files and directories always return a colourful display!).

2. If you haven't already, create a folder for this lab: call it `ECS781P` for example. Change the directory into it.

3. If you haven't done already, run `rhc setup`. When prompted for hostname, enter `openshift.eecs.qmul.ac.uk`. Then your user-name and password if prompted.

4. Run `rhc apps` to see a list of your apps. (You can also run `rhc account` to see the number of "gears" you are currently using.) If there is any (unless you are working on a cool app already!), delete the ones you don't need using the following command:

   ```
   rhc app-delete <name-of-the-app>
   ```

   You will be prompted with a warning that the deletion is irreversible, confirm with `yes`!

5. If you don't need any of your previous apps, delete the whole domain name: `rhc domain-delete -f` (Note: `-f` is to force delete, even if there are applications inside the domain name).

[1]The preparation part is adapted from the following tutorial: How to Install and Configure a Python Flask Dev & Deploy Environment, September 10, 2012, By Angel Rivera, Accessed on Feb 14, 2016.

6. Now that you have a clean slate, we can create our first proper app, which we call it `apitest1` using the following command (all in one line, and replace the `<your-user-name>` with your username, or some unique name!):

   ```
   rhc app-create -n ecs781pdev -a <your-user-name>apitest1
   -t python-2.7
   --from-code git://github.com/openshift/flask-example.git
   ```

   Note: `-n ecs781pdev` determines the name of the environment (which is `ecs781pdev`, where we will develop our apps), `-a <your-user-name>apitest1` sets the name of the app, `-t python-2.7` sets the type (cartridge) of our app (the programming language and web-server that we will be using is Python 2.7), and `--from-code git://github.com/openshift/flask-example.git` provides a link to a sample (hello-world) template based on `flask`.

7. Open your app in a browser (the URL is provided to you at the end of creating your app. If you missed it, you can check it again by using `rhc app-show`.

Creating a local dev-box (for testing our app locally, before pushing it for deployment on the cloud):

1. First, make sure your active python is version 2.7, by running: `python --version`. Then, check if you have `virtualenv` installed (e.g. run `virtualenv --version` from the terminal). If not, install it using the following command:

   ```
   pip install --user virtualenv
   ```

   The `--user` parameter means that it will be installed in your own home directory for you. Then execute: `export PATH=${HOME}/.local/bin:${PATH}`. This is to add `virtualenv` to your `PATH`. Now to ensure it went well, run `virtualenv --version` again.

   (Note: you can even install a different version of `python` locally without `sudo` privilege, but we don't need it as Python 2.7 is good enough for our purposes. Ask the instructor if you are interested!)

2. Change the directory into your app `cd apitest1`. Then change into `wsgi` subdirectory. Here is where we will create our virtual python environment (which we will name `venv`!):

   ```
   virtualenv venv --python=python2.7
   ```

3. "Activate" your virtual environment by running: `. venv/bin/activate`. Notice the change in your console prompt (you should see the prefix of `venv`) in parentheses. This means we can install python packages locally for our app (for testing in before pushing it to the cloud).

4. Install a few useful python packages: `flask`, `flask-wtf`, `flask-babel`, `markdown` and `flup`, by running:

   ```
   pip install flask flask-wtf flask-babel markdown flup
   ```

Note that we DO NOT push these packages to the PaaS. The PaaS (openshift) will automatically install these for us, as long as we put them either in `setup.py` or `requirement.txt` (as we will do in the next step!).

5. Navigate to the root folder of the app: `cd ..$`. Then using `gedit`, or `vim` (or any text editor like `sublime` or `emacs`), edit the line in the `setup.py` file for `install_requires` for the following (all in one line):

   ```
   install_requires=['Flask','flask-wtf','flask-babel',
   'markdown','flup'],
   ```

   This tells the PaaS cloud (openshift), which dependencies and packages should be installed. Note: you should also edit the rest of the entries of the `setup.py` appropriately, but it is not critical.

6. Next, we need to make sure we do not push these packages that we install for our own local testing to the cloud, which is our `venv` subdirectory (and also any other garbage, like `.pyc` and `.pyo` and backup files that are generated when running an app locally). We do this by adding them to the list of files that `git` should ignore (and not add them). We can do this by editing the `.gitignore` file (Note that the dot means it is a hidden file. It is located in the root directory of your app (where you currently are!). To see it use `ls -a`. But it can also be passed with the dot to be opened in the text editor, e.g. type in: `gedit .gitignore`). Add the following lines to `.gitignore`:

   ```
   venv/
   .project
   *.pyc
   *.pyo
   *~
   ```

Hello-World!

1. Ok, now we are ready to develop our app! Let's create some standard subfolders (standard web app layout): change your directory to `wsgi`. It should now have a subdirectory called `venv` where our local python packages are installed (and is NOT part of the git pushes). Create the following subfolders:

   ```
   mkdir app
   mkdir app/static
   mkdir app/templates
   mkdir tmp
   ```

   - ▷ The `app` folder is where we put our (python) application package.
   - ▷ `app/static` holds static files like images, javascripts, and cascading style sheets (CSS files).
   - ▷ `app/templates` is where we store our html templates.

2. Inside the `app` folder, create a file with the name of `__init__.py`, and save the following python commands in it:

```
from flask import Flask
app = Flask(__name__)
from app import views
```

Note that the existence of the file `__init__.py` in a folder is the standard way that python announces that this directory contain packages. `__init__.py` can just be an empty file in the simplest case, or it can contain initialization code for the package. Here, it simply creates the application object, which is of class `Flask` and calls it `app`. The script then imports the `views` module from the `app` directory (which we will write next). Just keep in mind that the first `app` refers to object of class `Flask`, while the second `app` is the package folder. If it confuses you, then assign different names to each (although, once you justified yourself about the difference, this is common to use the same name for both of them). Next, we write the `views` module.

3. From the same folder of `app`, create the file `views.py` with the following content:

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

The `views` are simple functions that return a string to be displayed into the browser. Here, the returned string is embarrassingly simple: "Hello, World!". The two route decorators above the `index()`, map the urls `/` and `/index` to the `index()` function (the name `index()` is arbitrary, but the route decorators are of course not!).

4. In the main (`wsgi`) folder, create the following python file, named `run.py`:

```
from app import app
if __name__ == "__main__":
        app.run(debug = True)
#Note: the debug MUST BE set to false in production for security!
```

5. You are now ready to test the application locally! (With your python virtual environment active:) Run the application in the background: `python run.py &`. The open a browser (also in the background): `firefox &`, and navigate to the local url address of the app: (`http://127.0.0.1:5000/`). You should see the Hello-World!

6. In your home folder, there is a file named `application`. Change its content to the following:

```
#!/usr/bin/python
import os
virtenv = os.environ['APPDIR'] + '/virtenv/'
```

```
os.environ['PYTHON_EGG_CACHE'] = os.path.join(virtenv,\
 'lib/python2.7/site-packages')
virtualenv = os.path.join(virtenv, 'bin/activate_this.py')
try:
        execfile(virtualenv, dict(__file__=virtualenv))
except IOError:
        pass
from run import app as application
```

This is just to let our PaaS provider (openshift) know it has to call `run.py`.

7. We are now ready to deploy the app in the cloud. First let us add and commit to all the changes. From the root folder of the app, (so `cd ..`), enter the following commands:

   ```
   git add .
   ```

   To add everything to the git index (note that our local packages and garbage files will not be added (will be ignored!) as we set it in `.gitignore`). Before, we commit, change the argument of `app.run` in `run.py` from `debug = True` to `debug = False`. Then, commit to all the changes, with a proper message:

   ```
   git commit -a -m "my initial deployment"
   ```

   Finally:

   ```
   git push
   ```

   Note that our openshift is setup so that the app is automatically deployed upon each push. To see if everything worked properly, open the public url of the app (again, to see the url you can use the `rhc app-show` command).

First API with No Cloud Storage

1. This is just to have you run your first API. The data is simply stored in memory as a python dictionary.
2. Change the content of the to the following:

```
from flask import Flask, jsonify

app = Flask(__name__)

mytasks = [
{
        'id': 1,
        'task': u'Finish Hello World',
        'session': u'Week 7',
        'done': True
},
{
```

```
        'id': 2,
        'task': u'Finish first API',
        'session': u'Week 7',
        'done': False
}
]

@app.route('/todo/api/tasks', methods=['GET'])
def get_tasks():
return jsonify({'tasks': mytasks})
```

3. Commit to changes and deploy the app. Can you see how to use your first API?