Course: Develop Microservices with Python Flask in 1-Day

Part 1: The Development Environment

Note: These labs are configured on a mac. Please adapt to your operating system

1.1 Install Python 3

Follow the instructions for your operating system

Linux: Python is probably already installed else install python with your OS package management system. On Ubuntu, us apt and on Redhat use yum. You can also download the binary from the link below.

Windows: Need to install. Download windows version from the link below

Mac: Python 2 is installed by default. Need to install Python 3. Download from the link before and follow instructions. Since you now have 2 versions of Python you may want to set alias to python3.

$which python3:

/usr/bin/python3

alias python=/usr/bin/python3

$ python --version

Python 3.8.2

https://www.python.org/downloads/

Once you have installed Python in your OS, check to ensure that Python version 3 is installed.

```
python --version

python 3.8.2
```

If download the latest version of python 3, your output may be slightly different from the above. What is important is that the output begins with Python 3, and not Python 2.

Mac Users: you may also use brew to install python3 by first installing brew

ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

1.2 Install Pip

Pip is the recommended tool for installing Python packages. Here is from the official pip page that can be found here: https://pip.pypa.io/en/stable/installing/#installing-with-get-pip-py

Do I need to install pip?

pip is already installed if you are using Python 2 >=2.7.9 or Python 3 >=3.4 downloaded from python.org or if you are working in a Virtual Environment created by virtualenv or venv. Just make sure to upgrade pip.

```
## we might need to change pip to pip3
```

```
$  pip --version
$ pip 9.0.1 ... (python 3.X)
$ which pip3
/usr/bin/pip3

alias pip = /usr/bin/pip3
pip --version

pip 19.2.3 from /Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.8/lib/
python3.8/site-packages/pip (python 3.8)
```

If the command above produces an output similar to pip 9.0.1 ... (python 3.X), then we are good to go. If we get pip 9.0.1... (python 2.X), then we can try replacing pip with pip3.

1.3 Install Integrated Development Environment(IDE):

You can use any IDE of your choice. If you don't have a favourite, I would recommend any of the following

Install sublime text
https://www.sublimetext.com/3

Install Visual studio code
https://code.visualstudio.com/?wt.mc_id=vscom_downloads

1.4 Python Virtual Environments:

It's always best to create a virtual environment for any python project. The concept of virtual environment is similar to creating a VM machine or container for a different project. Before we install flask we will first learn how to create a virtual environment and then install flask. There are different ways of creating virtual environments and below we will go through a few of them. You can stick to any method you like best.

Method 1: python Virtual environments with virtualenv

```
step 0: Install virtualenv
$ pip3 install virtualenv --user        #may need to use sudo pip3 install virtualenv --user
$ /usr/bin/easy_install virtualenv -user #may need to use sudo  usr/bin/easy_install virtualenv


Step 1: Create your labs directory and change directory to the directory
mkdir labs
cd labs

Step 2: Create a Virtual Environment
$ virtualenv venv

Step 3: Activate the virual environment
$ source venv/bin/activate

Step 4: Install flask using 'pip' which is symlinked to pip2 or pip3 (no sudo needed).Notice how the shell has
changed to (venvu)$
(venv)$ pip install flask --user
......
Installing collected packages: MarkupSafe, Werkzeug, Jinja2, itsdangerous, click, flask
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 flask-1.1.2 itsdangerous-1.1.0

Step 5: Check installed packages
(venv)$ pip show flask
Name: Flask
Version: 1.1.2
Summary: A simple framework for building complex web applications.
Home-page: https://palletsprojects.com/p/flask/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires: click, Jinja2, itsdangerous, Werkzeug
Required-by:

(venv)$ pip show Werkzeug
Name: Werkzeug
Version: 1.0.1
```

```
Summary: The comprehensive WSGI web application library.
Home-page: https://palletsprojects.com/p/werkzeug/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires:
Required-by: Flask
(venv)$ pip show Jinja2
Name: Jinja2
Version: 2.11.2
Summary: A very fast and expressive template engine.
Home-page: https://palletsprojects.com/p/jinja/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires: MarkupSafe
Required-by: Flask

Step 6:  Exit virtual environment
(venv)$ deactivate
```

1.5 Flask Virtual Environments using python -m (only python3)

```
Step 1: Create your labs directory and change directory to the directory
mkdir labs
cd labs

Step 2: Create a Virtual Environment
$ python3 -m venv  venv

Step 3: Activate the virual environment
$ source venv/bin/activate

Step 4: Install flask using 'pip' which is symlinked to pip2 or pip3 (no sudo needed).Notice how the shell has
changed to (venvu)$
(venv)$ pip install flask --user
......
Installing collected packages: MarkupSafe, Werkzeug, Jinja2, itsdangerous, click, flask
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 flask-1.1.2 itsdangerous-1.1.0

Step 5: Check installed packages
(venv)$ pip show flask
Name: Flask
Version: 1.1.2
Summary: A simple framework for building complex web applications.
Home-page: https://palletsprojects.com/p/flask/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires: click, Jinja2, itsdangerous, Werkzeug
Required-by:

(venv)$ pip show Werkzeug
Name: Werkzeug
Version: 1.0.1
Summary: The comprehensive WSGI web application library.
Home-page: https://palletsprojects.com/p/werkzeug/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires:
Required-by: Flask

(venv)$ pip show Jinja2
Name: Jinja2
Version: 2.11.2
Summary: A very fast and expressive template engine.
Home-page: https://palletsprojects.com/p/jinja/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires: MarkupSafe
Required-by: Flask

Step 6:  Exit virtual environment
(venv)$ deactivate
```

1.6 Flask Virtual Environments with pipenv

Virtual Environments (virtualenv)

Although PyPA (the Python Packaging Authority group) on this link https://pip.pypa.io/en/stable/user_guide/#requirements-files recommends pip as the tool for installing Python packages, we will need to use another package to manage our project's dependencies. It's true that pip supports package management through the requirements.txt file, but the tool lacks some features to be used on serious projects running on different production and development machines. Among its issues, the ones that cause the most problems are:

- pip installs packages globally, making it hard to manage multiple versions of the same package on the same machine.
- requirements.txt need all dependencies and sub-dependencies listed explicitly, a manual process that is tedious and error-prone.

To solve these issues, we are going to use Pipenv https://github.com/pypa/pipenv

Pipenv is a dependency manager that isolates projects on private environments, allowing packages to be installed per project. (If you're familiar with NPM or Ruby's bundler, it's similar in spirit to those tools)

pipenv create a virtual environment, where all the dependencies will be installed. On the project directory, you will see 2 files created after executing the pipenv commands.

1. Pipfile, a file that contains details about our project, like the Python version that we are using and the packages that our project needs.
2. Pipenv.lock,a file that contains exactly what version of each package our project depends on, and its transitive dependencies.

```
Step 1: Create your labs directory and change directory to the directory
mkdir labs
cd labs

Step 2: Create a Virtual Environment
$ pip install pipenv
$ pipenv --three

Step 3: Install flask using 'pip' which is symlinked to pip2 or pip3 (no sudo needed).Notice how the shell has
changed to (venvu)$
(venv)$ pipenv install flask
Installing flask...
Adding flask to Pipfile's [packages]...
✔ Installation Succeeded
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Building requirements...
Resolving dependencies...
✔ Success!
Updated Pipfile.lock (ac8e32)!
Installing dependencies from Pipfile.lock (ac8e32)...
      ████████████████████████████████ 0/0 — 00:00:00
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.

Step 4: Activate the virtual environment:
$ pipenv shell

Step 5: Check installed packages
(lab)$ pip show flask
Name: Flask
Version: 1.1.2
Summary: A simple framework for building complex web applications.
Home-page: https://palletsprojects.com/p/flask/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
```

```
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires: click, Jinja2, itsdangerous, Werkzeug
Required-by:

(lab)$ pip show Werkzeug
Name: Werkzeug
Version: 1.0.1
Summary: The comprehensive WSGI web application library.
Home-page: https://palletsprojects.com/p/werkzeug/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires:
Required-by: Flask

(lab)$ pip show Jinja2
Name: Jinja2
Version: 2.11.2
Summary: A very fast and expressive template engine.
Home-page: https://palletsprojects.com/p/jinja/
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/training/env/lib/python3.8/site-packages
Requires: MarkupSafe
Required-by: Flask

Step 6:  Exit virtual environment
(lab)$ deactivate
```

1.7 Test The Installation

Now that our environment has been successfully setup, let is create a hello world program in Flask.

Create a file called hello.py and add the lines below

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "hello world"
#if __name__ == "__main__":
#    app.run(debug=True)
```

These 5 lines of code are everything we need to handle HTTP requests and return a "Hello, World!" message. To run it, we need to export an environment variable called FLASK_APP and then execute flask:

After executing these commands, we can reach our application by opening a browser and navigating to http://127.0.0.1:5000/ or by issuing curl http://127.0.0.1:5000/ from another terminal.

Let us explore different ways to run a flask application.

1.8 Ways to Run a Flask app

You can run a flask app either by calling ‘flask’ or by calling ‘python’. If you call flask, ensure to set/export the environmental variables like FLASK_APP, FLASK_DEBUG, FLASK_ENV. Examples follow:

Method 1: Use the flask command

Here is the code:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "hello world"
#if __name__ == "__main__":
```

```
#    app.run(debug=True)


$ export FLASK_APP=hello_world.py

$ flask run

* Running on http://127.0.0.1:5000/

or

$flask run --host=0.0.0.0 --port=8000
```

Method 2: Using python command

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "hello world"
if __name__ == "__main__":
    app.run(debug=True)

python hello_world.py
```

Method 3:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "hello world"
#if __name__ == "__main__":
#    app.run(debug=True)


export FLASK_APP=hello_world.py

$ python -m flask run

 * Running on http://127.0.0.1:5000/
```

Method 1 and 2 are more common. Just remember to set the environment variables when using the flask command.

Part 2: Flask in Action

url: https://flask.palletsprojects.com/en/1.1.x/

Routing

Flask helps with routing your users to the website URL. This is pretty much easy and straightforward to do in Flask with the **route() decorator.**

Use the route() decorator to bind a function to a URL. Examples include:

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

2.1 Running Flask Applications

We will continue with the Hello-world Python-Flask Webapp written above

Create a project called part1 with a module called hello_world (save as hello_world.py), with this content:

```
"""
hello_world: First Python-Flask webapp
"""
from flask import Flask     # From module flask import class Flask
app = Flask(__name__)       # Construct an instance of Flask class for the webapp

@app.route('/')             # URL '/' to be handled by main() route handler
def main():
    return 'Hello, world!'
if __name__ == '__main__':  # Script executed directly?
    app.run()                 # Launch built-in web server and run this Flask webapp
```

Run the Webapp

Starting from Flask 0.11, there is a new way of running flask app via the 'flask' command or Python's -m switch through an environment variable FLASK_APP, as follows. Using environment is more flexible in changing the configuration (e.g., enable debug mode), without modifying the code itself.

```
$ export FLASK_APP=hello_world.py
$ flask run

 * Running on http://127.0.0.1:5000/
```

Or,

```
$ export FLASK_APP=hello_flask.py
$ python -m flask run

 * Running on http://127.0.0.1:5000/
```

Or,

```
$ python hellow_world.py
```

```
 * Running on http://127.0.0.1:5000/
```

Here, the Flask built-in web server has been started, listening on TCP port 5000. The webapp has also been started. It routes the URL '/' request to main() which returns the hello-world message.
From a web browser, issue URL http://127.0.0.1:5000/ (or http://localhost:5000/) to trigger the webapp (localhost is the domain name for local loop-back IP address (127.0.0.1). You should see the hello-world message.

Flask's "Debug" Mode - Enabling "Reloader" and "Debugger"

In the above examples, you need to restart the app if you make any modifications to the codes. For development, you can turn on the debug mode, which enables the auto-reloader that will provide you with a helpful debugger if things go wrong.

If you enable debug support the server will reload itself on code changes, and it will also provide you with a helpful debugger if things go wrong. You can do that with the command:

$ export FLASK_DEBUG=1

To enable development features you can export the FLASK_ENV environment variable and set it to development before running the server:

$ export FLASK_ENV=development

Within your Flask code, there are two ways to turn on debug mode:

1.Set the debug attribute of the Flask instance app to True:

```
app = Flask(__name__)
app.debug = True      # Enable reloader and debugger
......

if __name__ == '__main__':
   app.run()
```

2.Pass a keyword argument debug=True into the app.run():

```
app = Flask(__name__)
......
if __name__ == '__main__':
   app.run(debug=True)  # Enable reloader and debugger
```

To Try:

1.Change to app.run(debug=True) to enable to auto reloader.
2.Restart the app. Observe the console messages:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: xxx-xxx-xxx
 ......
```

3.Add one more route'/hi' without stopping the app, and try the new route. Check the console messages.

In debug mode, the Flask app monitors your source code, and reload the source code if any modification is detected (i.e., auto-reloader). It also launches the debugger if an error is detected.

IMPORTANT: Debug mode should NOT be used for production, because it impedes the performance, and worse still, lets users execute codes on the server.

FLASK_DEBUG Environment Variable

Starting from Flask 0.11, you can enable the debug mode via environment variable **FLASK_DEBUG** without changing the codes, as follows:

```
$ export FLASK_APP=hello_flask.py
$ export FLASK_DEBUG=1
$ flask run
```

Serving HTML pages

Instead of plain-text, you can respond with an HTML page by returning an HTML-formatted string. For example, modify the hello_world.py as follows:

```
......
@app.route('/')
@app.route('/hello')
@app.route('/hi')
def main():
    """Return an HTML-formatted string and an optional response status code"""
    return """
      <!DOCTYPE html>
      <html>
      <head><title>Hello</title></head>
      <body><h1>Hello, from HTML</h1></body>
      </html>
      """, 200
......
```

MVC Model: Separating the Presentation View from the Controller with Templates

Instead of putting the response message directly inside the route handler (which violates the MVC architecture), you can separate the View (or Presentation) from the Controller by creating an HTML page called "helloworld.html" under a sub-directory called "templates", as follows:

templates/helloworld.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello</title>
</head>
<body>
  <h1>Hello, from templates</h1>
</body>
</html>
```

Modify the hello_world.py as follows:

```
......
from flask import Flask, render_template  # Need render_template() to render HTML pages
```

```
......
@app.route('/')
def main():
   """Render an HTML template and return"""
   return render_template('helloworld.html')  # HTML file to be placed under sub-directory templates
......
```

URL Routes With Variables

You can use variables in the URL routes. For example, create a new Python module called **hello_urlvar** (save as **hello_urlvar.py**) as follows:

```
"""
hello_urlvar: Using URL Variables
"""
from flask import Flask
app = Flask(__name__)


@app.route('/hello')
def hello():
    return 'Hello, world!'


@app.route('/hello/<username>')         # URL with a variable
def hello_username(username):        # The function shall take the URL variable as parameter
    return 'Hello, {}'.format(username)


@app.route('/hello/<int:userid>')       # Variable with type filter. Accept only int
def hello_userid(userid):           # The function shall take the URL variable as parameter
    return 'Hello, your ID is: {:d}'.format(userid)


if __name__ == '__main__':
    app.run(debug=True)                 # Enable reloader and debugger
```

To Try:
1. run the above code.
2. Access the Urls passing parameters of your choice

Functions redirect() and url_for()

- Redirect(): In your route handlers, you can use **"return redirect(url)"** to redirect **the response** to another url.
- url_for(): Instead of hard-coding URLs in your route handlers, you could use **“url_for(route_handler)”** helper function to generate URL based on the route handler's name. The url_for(route_handler) takes a route handler (view function) name, and returns its URL.

The mapping for URLs and view functions are kept in property app.url_map, as shown in the earlier example.
For example, suppose that main() is the route handler (view function) for URL '/', an hello(username) for URL '/hello/<username>':

- url_for('main'): returns the internal (relative) URL '/'.
- url_for('main', _external=True): returns the external (absolute) URL 'http://localhost:5000/'.

- url_for('main', page=2): returns the internal URL '/?page=2'. All the additional keyword arguments are treated as GET request parameters.
- url_for('hello', username='foo', _external=True): returns the external URL 'http://localhost:5000/hello/foo'.

For example,

```
"""
hello_urlredirect: Using functions redirect() and url_for()
"""
from flask import Flask, redirect, url_for

app = Flask(__name__)


@app.route('/')

def main():
    return redirect(url_for('hello', username='foo'))
        # Also pass an optional URL variable


@app.route('/hello/<username>')

def hello(username):

    return 'Hello, {}'.format(username)


if __name__ == '__main__':

    app.run(debug=True)
```

Try:

1.Issue URL http://localhost:5000, and observe that it will be redirected to http://localhost:5000/hello/foo.

2.The console message clearly indicate the redirection:

```
127.0.0.1 - - [15/Jan/2021 17:58:10] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [15/Jan/2021 17:58:10] "GET /hello/foo HTTP/1.1" 200 -
```

3.The original request to '/' has a response status code of "302 Found" and was redirected to '/hello/foo'.

It is strongly recommended to use **url_for(route_handler)**, instead of hardcoding the links in your codes, as it is more flexible, and allows you to easily change your URL.

Other Static Resources: Images, CSS, Javascript

By default, Flask built-in server locates the static resources under the **sub-directory static**. You can create sub-sub-directories such as img, css and js. For example,

```
project-directory
  |
  +-- templates (for Jinja2 HTML templates)
  +-- static
      |
      +-- css
      +-- js
      +-- img
```

- You can use url_for() to reference static resources. For example, to refer to /static/js/main.js, you can use **url_for('static', filename='js/main.js').**
- Static resources are often referenced inside Jinja2 templates (which will be rendered into HTML pages)
- In production, the **static directory** should be served directly by an HTTP server (such as Apache) for performance.

Jinja2 Template Engine

In Flask, the route handlers (or view functions) are primarily meant for the business logic (i.e., Controller in MVC), while the presentation (i.e., View in MVC) is to be taken care by the so-called **templates**. A template is a file that contains the static texts, as well as placeholder for rendering dynamic contents. Flask uses a powerful template engine called Jinja2.

Getting Started: Getting Started in Jinja2 Templates with HTML Forms

Create a sub-directory called 'templates' under your project directory. Create a j2_query.html (which is an ordinary html file) under the '**templates**' sub-directory, as follows:

templates/j2_query.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Entry Page</title>
</head>
<body>
<form action="process" method="post">
  <label for="username">Please enter your name: </label>
  <input type="text" id="username" name="username"><br>
  <input type="submit" value="SEND">
</form>
</body>
</html>
```

Create a templated-html file called j2_response.html under the **'templates'** sub-directory, as follows:
templates/j2_response.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Response Page</title>
</head>
<body>
  <h1>Hello, {{ username }}</h1>
</body>
</html>
```

Create the main script **hello_jinja2.py** under your project directory, as follows:

```
"""
hello_jinja2: Get start with Jinja2 templates
"""
from flask import Flask, render_template, request
app = Flask(__name__)

@app.route('/')
def main():
    return render_template('j2_query.html')

@app.route('/process', methods=['POST'])
def process():
```

```
    # Retrieve the HTTP POST request parameter value from 'request.form' dictionary
    _username = request.form.get('username')  # get(attr) returns None if attr is not present

    # Validate and send response
    if _username:
        return render_template('j2_response.html', username=_username)
    else:
        return 'Please go back and enter your name...', 400  # 400 Bad Request

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

run the webapp

python hello_jinja2.py

Then Try:

1.Issue URL http://localhost:5000, enter the username and "SEND".

2.Issue URL http://localhost:5000, without entering the username and "SEND".

3.Issue URL http://localhost:5000/process. You should receive "405 Method Not Allowed" as GET request was used instead of POST.

Part 3

SQLAlchemy

Reference: SQLAlchemy @ http://www.sqlalchemy.org/.

SQLAlchemy is great for working with relational databases. It is the Python SQL toolkit and Object Relational Mapper (ORM) that gives you the full power and flexibility of SQL. It works with MySQL, PostgreSQL, SQLite, and other relational databases. SQLAlchemy implements the Model of the MVC architecture for Python-Flask webapps.

Installing SQLAlchemy (under virtual environment)

```
$ cd /path/to/project-directory
$ source venv/bin/activate    # Activate the virtual environment

(venv)$ pip install sqlalchemy
(venv)$ pip show sqlalchemy
Name: SQLAlchemy
Version: 1.3.22
Summary: Database Abstraction Library
Home-page: http://www.sqlalchemy.org
Author: Mike Bayer
Author-email: mike_mp@zzzcomputing.com
License: MIT
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires:
Required-by: Flask-SQLAlchemy
```

Using SQLAlchemy with MySQL/PostgreSQL

Installing MySQL Driver for Python

```
$ cd /path/to/project-directory
$ source venv/bin/activate    # Activate the virual environment

# Python 3
```

```
(venv)$ brew install mysql
(venv)$ pip install mysqlclient
Successfully installed mysqlclient-1.3.9
(venv)$ pip show mysqlclient
Name: mysqlclient
Version: 2.0.3
Summary: Python interface to MySQL
Home-page: https://github.com/PyMySQL/mysqlclient
Author: Inada Naoki
Author-email: songofacandy@gmail.com
License: GPL
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires:
Required-by:
```

Installing PostgreSQL Driver for Python

```
$ cd /path/to/project-directory
$ source venv/bin/activate    # Activate the virual environment

(venv)$ pip install psycopg2
(venv)$ brew install  postgres

(venv)$ pip show psycopg2
Name: psycopg2
Version: 2.8.6
Summary: psycopg2 - Python-PostgreSQL Database Adapter
Home-page: https://psycopg.org/
Author: Federico Di Gregorio
Author-email: fog@initd.org
License: LGPL with exceptions
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires:
Required-by:
```

Setting up MySQL

Login to MySQL. Create a test user (called `bill`) and a test database (called `testdb`) as follows:

```
$ mysql -u root -p
mysql> create user 'bill'@'localhost' identified by 'passpass';
mysql> create database if not exists testdb;
mysql> grant all on testdb.* to 'bill'@'localhost';
mysql> quit
```

Setting up PostgreSQL

Create a test user (called `bill`) and a test database (called `testdb` owned by `bill`) as follows:

```
# Create a new PostgreSQL user called bill, allow user to login, but NOT creating databases
# Create a new database called testdb, owned by bill.
```

$ psql postgres

```
(venv) bash-3.2$ psql postgres
psql (13.1)
Type "help" for help.
postgres=#
postgres=# CREATE ROLE bill WITH LOGIN PASSWORD 'passpass';
CREATE ROLE
postgres=# exit
(venv) bash-3.2$ psql postgres -U bill
psql (13.1)
Type "help" for help.

postgres=>
postgres=> createdb testdb --owner bill
postgres-> \connect testdb
You are now connected to database "testdb" as user "bill".
testdb-> \list
                                 List of databases
   Name    |   Owner    | Encoding | Collate | Ctype |     Access privileges
-----------+------------+----------+---------+-------+---------------------------
 postgres  | damianigbe | UTF8     | C       | C     |
 template0 | damianigbe | UTF8     | C       | C     | =c/damianigbe            +
           |            |          |         |       | damianigbe=CTc/damianigbe
 template1 | damianigbe | UTF8     | C       | C     | =c/damianigbe            +
           |            |          |         |       | damianigbe=CTc/damianigbe
```

```
 testdb    | bill       | UTF8     | C       | C     |
(4 rows)

testdb-> \dt
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | cafe | table | bill
(1 row)

testdb=> select * from cafe
```

Restart PostgreSQL server:

```
$ brew services start postgres
```

Example 1: Connecting to MySQL and Executing SQL statements

```
"""
mysql.py: SQLAlchemy Example 1 - Testing with MySQL
"""
from sqlalchemy import create_engine

engine = create_engine('mysql://bill:passpass@localhost:3306/testdb')


engine.echo = True  # Echo output to console


# Create a database connection
conn = engine.connect()

conn.execute('DROP TABLE IF EXISTS cafe')

conn.execute('''CREATE TABLE IF NOT EXISTS cafe (
                          id INT UNSIGNED NOT NULL AUTO_INCREMENT,
                          category ENUM('tea', 'coffee') NOT NULL,
                          name VARCHAR(50) NOT NULL,
                          price DECIMAL(5,2) NOT NULL,
                          PRIMARY KEY(id)
                       )''')


# Insert one record
conn.execute('''INSERT INTO cafe (category, name, price) VALUES
                          ('coffee', 'Espresso', 3.19)''')


# Insert multiple records
conn.execute('''INSERT INTO cafe (category, name, price) VALUES
                          ('coffee', 'Cappuccino', 3.29),
                          ('coffee', 'Caffe Latte', 3.39),
                          ('tea', 'Green Tea', 2.99),
```

```
                        ('tea', 'Wulong Tea', 2.89)''')


# Query table
for row in conn.execute('SELECT * FROM cafe'):
      print(row)


# give connection back to the connection pool
conn.close()
```

Example 2: Connecting to PostgreSQL and Executing SQL statements

```
"""
postgresql.py: SQLAlchemy Example 2 - Testing with PostgreSQL
"""
from sqlalchemy import create_engine
engine = create_engine('postgresql://bill:passpass@localhost:5432/testdb')


engine.echo = True  # Echo output to console

# Create a database connection
conn = engine.connect()
conn.execute('''CREATE TABLE IF NOT EXISTS cafe (
                  id SERIAL,
                  category VARCHAR(10) NOT NULL,
                  name VARCHAR(50) NOT NULL,
                  price DECIMAL(5,2) NOT NULL,
                  PRIMARY KEY(id)
                )''')


# Insert one record
conn.execute('''INSERT INTO cafe (category, name, price) VALUES
                  ('coffee', 'Espresso', 3.19)''')


# Insert multiple records
conn.execute('''INSERT INTO cafe (category, name, price) VALUES
                  ('coffee', 'Cappuccino', 3.29),
                  ('coffee', 'Caffe Latte', 3.39),
                  ('tea', 'Green Tea', 2.99),
                  ('tea', 'Wulong Tea', 2.89)''')


# Query table
for row in conn.execute('SELECT * FROM cafe'):
    print(row)
```

```
# give connection back to the connection pool
conn.close()
```

In this example, we issue raw SQL statements through SQLAlchemy, which is NOT the right way of using SQLAlchemy. Also take note that MySQL and PostgreSQL has a different CREATE TABLE, as they have their own types.

Using SQLALchemy ORM (Object-Relational mapper)

Reference: Object Relational Tutorial @http://docs.sqlalchemy.org/en/latest/orm/tutorial.html.

Instead of writing raw SQL statements, which are tedious, error-prone, inflexible and hard-to-maintain, we can use an ORM (Object-Relational Mapper).

SQLAlchemy has a built-in ORM, which lets you work with relational database tables as if there were native object instances (having attributes and methods). Furthermore, you can include additional attributes and methods to these objects.

Example 1: Using ORM to CREATE/DROP TABLE, INSERT, SELECT, and DELETE

```
"""
mysql_orm.py: SQLAlchemy Example 2 - Using ORM (Object-Relational Mapper)
"""
from sqlalchemy import create_engine, Column, Integer, String, Enum, Numeric
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, scoped_session


# Get the base class of our models
Base = declarative_base()


# Define Object Mapper for table 'cafe'
class Cafe(Base):
    __tablename__ = 'cafe'


    # These class variables define the column properties,
    #   while the instance variables (of the same name) hold the record values.
    id = Column(Integer, primary_key=True, autoincrement=True)
    category = Column(Enum('tea', 'coffee', name='cat_enum'))  # PostgreSQL ENUM type requires a name
    name = Column(String(50))
    price = Column(Numeric(precision=5, scale=2))


    def __init__(self, category, name, price, id=None):
        """Constructor"""
        if id:
            self.id = id   # Otherwise, default to auto-increment
        self.category = category
        self.name = name
        self.price = price
        # NOTE: You can use the default constructor,
        #   which accepts all the fields as keyword arguments
```

```
    def __repr__(self):
        """Show this object (database record)"""
        return "Cafe(%d, %s, %s, %5.2f)" % (
            self.id, self.category, self.name, self.price)

# Create a database engine
engine = create_engine('mysql://bill:passpass@localhost:3306/testdb')
#engine = create_engine('postgresql://bill:passpass@localhost:5432/testdb')
engine.echo = True  # Echo output to console for debugging

# Drop all tables mapped in Base's subclasses
Base.metadata.drop_all(engine)

# Create all tables mapped in Base's subclasses
Base.metadata.create_all(engine)
# -- MySQL --
# CREATE TABLE cafe (
#   id INTEGER NOT NULL AUTO_INCREMENT,
#   category ENUM('tea','coffee'),
#   name VARCHAR(50),
#   price NUMERIC(5, 2),
#   PRIMARY KEY (id)
# )
# -- PostgreSQL --
# CREATE TYPE cat_enum AS ENUM ('tea', 'coffee')
# CREATE TABLE cafe (
#   id SERIAL NOT NULL,
#   category cat_enum,
#   name VARCHAR(50),
#   price NUMERIC(5, 2),
#   PRIMARY KEY (id)
# )

# Create a database session binded to our engine, which serves as a staging area
# for changes to the objects. To make persistent changes to database, call
# commit(); otherwise, call rollback() to abort.
Session = scoped_session(sessionmaker(bind=engine))
dbsession = Session()

# Insert one row via add(instance) and commit
dbsession.add(Cafe('coffee', 'Espresso', 3.19))  # Construct a Cafe object
# INSERT INTO cafe (category, name, price) VALUES (%s, %s, %s)
# ('coffee', 'Espresso', 3.19)
dbsession.commit()

# Insert multiple rows via add_all(list_of_instances) and commit
dbsession.add_all([Cafe('coffee', 'Cappuccino', 3.29),
                   Cafe('tea', 'Green Tea', 2.99, id=8)])  # using kwarg for id
dbsession.commit()

# Select all rows. Return a list of Cafe instances
for instance in dbsession.query(Cafe).all():
    print(instance.category, instance.name, instance.price)
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category,
#   cafe.name AS cafe_name, cafe.price AS cafe_price FROM cafe
# coffee Espresso 3.19
# coffee Cappuccino 3.29
# tea Green Tea 2.99
```

```
# Select the first row with order_by. Return one instance of Cafe
instance = dbsession.query(Cafe).order_by(Cafe.name).first()
print(instance)    # Invoke __repr__()
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category,
#    cafe.name AS cafe_name, cafe.price AS cafe_price
# FROM cafe ORDER BY cafe.name LIMIT %s
# (1,)
# Cafe(2, coffee, Cappuccino,  3.29)

# Using filter_by on column
for instance in dbsession.query(Cafe).filter_by(category='coffee').all():
    print(instance.__dict__)    # Print object as key-value pairs
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category,
#    cafe.name AS cafe_name, cafe.price AS cafe_price
# FROM cafe WHERE cafe.category = %s
$ ('coffee',)

# Using filter with criterion
for instance in dbsession.query(Cafe).filter(Cafe.price < 3).all():
    print(instance)
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category,
#    cafe.name AS cafe_name, cafe.price AS cafe_price
# FROM cafe WHERE cafe.price < %s
# (3,)
# Cafe(8, tea, Green Tea,  2.99)

# Delete rows
instances_to_delete = dbsession.query(Cafe).filter_by(name='Cappuccino').all()
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category,
#    cafe.name AS cafe_name, cafe.price AS cafe_price
# FROM cafe WHERE cafe.name = %s
# ('Cappuccino',)
for instance in instances_to_delete:
    dbsession.delete(instance)
dbsession.commit()
# DELETE FROM cafe WHERE cafe.id = %s
# (2,)

for instance in dbsession.query(Cafe).all():
    print(instance)
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category,
#    cafe.name AS cafe_name, cafe.price AS cafe_price
# FROM cafe
# Cafe(1, coffee, Espresso,  3.19)
# Cafe(8, tea, Green Tea,  2.99)
```

To try:

1. Run the above example and see the entries created

2. connect to mysql database and query the cafe table to see the entries added

The above program works for MySQL and PostgreSQL!!!

Study the log messages produced. Instead of issuing raw SQL statement (of INSERT, SELECT), we program on the objects. The ORM interacts with the underlying relational database by issuing the appropriate SQL statements. Take note that you can include additional attributes and methods in the Cafe class to facilitate your application.

Executing Query:

- get(primary-key-id): execute the query with the primary-key identifier, return an object or None.
- all(): executes the query and return a list of objects.
- first(): executes the query with LIMIT 1, and returns the result row as tuple, or None if no rows were found.
- one(): executes the query and raises MultipleResultsFound if more than one row found; NoResultsFound if no rows found. Otherwise, it returns the sole row as tuple.
- `one_or_none()`: executes the query and raises `MultipleResultsFound` if more than one row found. It return `None` if no rows were found, or the sole row.
- `scaler()`: executes the query and raises `MultipleResultsFound` if more than one row found. It return `None` if no rows were found, or the "first column" of the sole row.

Filtering the Query object, and return a Query object:

- `filter(*criterion)`: filtering criterion in SQL WHERE expressions, e.g., `id > 5`.
- `filter_by(**kwargs)`: filtered using keyword expressions (in Python), e.g., `name = 'some name'`.

Example 2: ORM with One-to-Many Relation

```
"""
sqlalchemy_eg3: SQLAlemcy Example 3 - Using ORM with one-to-many relation
"""
from sqlalchemy import create_engine, Column, Integer, String, Enum, Numeric, ForeignKey

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import relationship

from sqlalchemy.orm import sessionmaker, scoped_session


# Get the base class of our models
Base = declarative_base()


# Define Object Mapper for table 'supplier'
# -- MySQL --
# CREATE TABLE supplier (
#    id INTEGER NOT NULL AUTO_INCREMENT,
#    name VARCHAR(50),
#    PRIMARY KEY (id)
# )
# -- PostgreSQL --
# CREATE TABLE supplier (
#   id SERIAL NOT NULL,
#   name VARCHAR(50),
#   PRIMARY KEY (id)
# )
class Supplier(Base):

    __tablename__ = 'supplier'

    id = Column(Integer, primary_key=True, autoincrement=True)

    name = Column(String(50))

    items = relationship('Cafe', backref='item_supplier')


    def __repr__(self):

        return "Supplier(%d, %s)" % (self.id, self.name)


# Define Object Mapper for table 'cafe'
# -- MySQL --
# CREATE TABLE cafe (
#    id INTEGER NOT NULL AUTO_INCREMENT,
```

```
#    category ENUM('tea','coffee'),
#    name VARCHAR(50),
#    price NUMERIC(5, 2),
#    supplier_id INTEGER,
#    PRIMARY KEY (id),
#    FOREIGN KEY(supplier_id) REFERENCES supplier (id)
# )
# -- PostgreSQL --
# CREATE TYPE cat_enum AS ENUM ('tea', 'coffee')
# CREATE TABLE cafe (
#   id SERIAL NOT NULL,
#   category cat_enum,
#   name VARCHAR(50),
#   price NUMERIC(5, 2),
#   supplier_id INTEGER,
#   PRIMARY KEY (id),
#   FOREIGN KEY(supplier_id) REFERENCES supplier (id)
# )
class Cafe(Base):

    __tablename__ = 'cafe'

    id = Column(Integer, primary_key=True, autoincrement=True)

    category = Column(Enum('tea', 'coffee', name='cat_enum'), default='coffee')

    name = Column(String(50))

    price = Column(Numeric(precision=5, scale=2))

    supplier_id = Column(Integer, ForeignKey('supplier.id'))

    supplier = relationship('Supplier', backref='cafe_items', order_by=id)


    def __repr__(self):

        return "Cafe(%d, %s, %s, %5.2f, %d)" % (

            self.id, self.category, self.name, self.price, self.supplier_id)


# Create a database engine
engine = create_engine('mysql://bill:passpass@localhost:3306/testdb')

#engine = create_engine('postgresql://bill:passpass@localhost:5432/testdb')

engine.echo = True  # Echo output to console for debugging


# Drop all tables
Base.metadata.drop_all(engine)


# Create all tables
Base.metadata.create_all(engine)


# Create a database session binded to our engine, which serves as a staging area
# for changes to the objects. To make persistent changes to database, call
# commit() or rollback().
Session = scoped_session(sessionmaker(bind=engine))

dbsession = Session()


# Insert one row via add(instance)
dbsession.add(Supplier(id=501, name='ABC Corp'))

dbsession.add(Supplier(id=502, name='XYZ Corp'))

dbsession.commit()


# Insert multiple rows via add_all(list_of_instances)
```

```
dbsession.add_all([Cafe(name='Espresso', price=3.19, supplier_id=501),
                   Cafe(name='Cappuccino', price=3.29, supplier_id=501),
                   Cafe(category='tea', name='Green Tea', price=2.99, supplier_id=502)])
dbsession.commit()


# Query table with join
for c, s in dbsession.query(Cafe, Supplier).filter(Cafe.supplier_id==Supplier.id).all():
    print(c, s)
# SELECT cafe.id AS cafe_id, cafe.category AS cafe_category, cafe.name AS cafe_name,
#   cafe.price AS cafe_price, cafe.supplier_id AS cafe_supplier_id,
#   supplier.id AS supplier_id, supplier.name AS supplier_name
# FROM cafe, supplier WHERE cafe.supplier_id = supplier.id
# Cafe(1, coffee, Espresso,  3.19, 501) Supplier(501, ABC Corp)
# Cafe(2, coffee, Cappuccino,  3.29, 501) Supplier(501, ABC Corp)
# Cafe(3, tea, Green Tea,  2.99, 502) Supplier(502, XYZ Corp)


for instance in dbsession.query(Supplier.name, Cafe.name).join(Supplier.items).filter(Cafe.category=='coffee').all():
    print(instance)
# SELECT supplier.name AS supplier_name, cafe.name AS cafe_name
# FROM supplier INNER JOIN cafe ON supplier.id = cafe.supplier_id
# WHERE cafe.category = %s
# ('coffee',)
# ('ABC Corp', 'Espresso')
# ('ABC Corp', 'Cappuccino')
```

Flask-SQLAlchemy

Reference: Flask-SQLAlchemy @ http://flask-sqlalchemy.pocoo.org/2.1/.
Flask-SQLAlchemy is a thin extension that wraps SQLAlchemy around Flask. It allows you to configure the SQLAlchemy engine through Flask webapp's configuration file and binds a database session to each request for handling transactions.

To install Flask-SQLAlchemy (under a virtual environment):

```
$ cd /path/to/project-directory
$ source venv/bin/activate  # Activate the virual environment

$ (venv)pip install flask-sqlalchemy
Successfully installed flask-sqlalchemy-2.1
$ (venv)pip show --files flask-sqlalchemy
Name: Flask-SQLAlchemy
Version: 2.4.4
Summary: Adds SQLAlchemy support to your Flask application.
Home-page: https://github.com/pallets/flask-sqlalchemy
Author: Armin Ronacher
Author-email: armin.ronacher@active-4.com
License: BSD-3-Clause
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires: Flask, SQLAlchemy
Required-by:
Files:
  Flask_SQLAlchemy-2.4.4.dist-info/INSTALLER
  Flask_SQLAlchemy-2.4.4.dist-info/LICENSE.rst
  Flask_SQLAlchemy-2.4.4.dist-info/METADATA
  Flask_SQLAlchemy-2.4.4.dist-info/RECORD
  Flask_SQLAlchemy-2.4.4.dist-info/WHEEL
  Flask_SQLAlchemy-2.4.4.dist-info/top_level.txt
  flask_sqlalchemy/__init__.py
  flask_sqlalchemy/_compat.py
  flask_sqlalchemy/model.py
  flask_sqlalchemy/utils.py
```

Example 1: Running SQLAlchemy under Flask Webapp

In this example, we shall separate the Model from the Controller, as well as the web forms and templates.

flask_sql_alchemy_models.py

```
"""
fsqlalchemy_eg1_models: Flask-SQLAlchemy EG 1 - Define Models and Database Interfaces.
"""
from flask_sqlalchemy import SQLAlchemy  # Flask-SQLAlchemy

# Configure and initialize SQLAlchemy under this Flask webapp.
db = SQLAlchemy()


class Cafe(db.Model):
    """Define the 'Cafe' model mapped to database table 'cafe'."""
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    category = db.Column(db.Enum('tea', 'coffee', name='cat_enum'), nullable=False, default='coffee')
    name = db.Column(db.String(50), nullable=False)
    price = db.Column(db.Numeric(precision=5, scale=2), nullable=False, default=999.99)

    def __repr__(self):
        """Describe itself."""
        return 'Cafe(%d, %s, %s, %5.2f)' % (
            self.id, self.category, self.name, self.price)


def load_db(db):
    """Create database tables and insert records"""
    # Drop and re-create all the tables.
    db.drop_all()
    db.create_all()

    # Insert rows using add_all(list_of_instances).
    # Use the default constructor with keyword arguments.
    db.session.add_all([
        Cafe(name='Espresso', price=3.19),
        Cafe(name='Cappuccino', price=3.29),
        Cafe(name='Green Tea', category='tea', price=2.99)])
    db.session.commit()  # Always commit after insert
```

Hence, there is no need to define a constructor. However, if you need to define a constructor, do it this way:

```
def __init__(self, your-args, **kwargs):
   super(User, self).__init__(**kwargs)  # Invoke the superclass keyword constructor
   # your custom initialization here
```

1.The db.drop_all() and db.create_all() drops and create all tables defined under this db.Model. To create a single table, use ModelName.__table__.create(db.session.bind). You can include checkfirst=True to add "IF NOT EXISTS".

flask_sql_alchemy_controller.py

```
"""
flask_sql_alchemy_controller: Flask-SQLAlchemy EG 1 - Main controller.
"""
from flask import Flask, render_template
from  flask_sql_alchemy_models  import db, Cafe, load_db  # Flask-SQLAlchemy

# Flask: Initialize
app = Flask(__name__)


# Flask-SQLAlchemy: Initialize
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://bill:passpass@localhost:3306/testdb'
#app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://bill:passpass@localhost:5432/testdb'
db.init_app(app)   # Bind SQLAlchemy to this Flask webapp
```

```
# Create the database tables and records inside a temporary test context
with app.test_request_context():
    load_db(db)


@app.route('/')
@app.route('/<id>')
def index(id=None):
    if id:
        _item_list = Cafe.query.filter_by(id=id).all()
    else:
        _item_list = Cafe.query.all()


    return render_template('flask_sql_alchemy_list.html', item_list=_item_list)


if __name__ == '__main__':
    app.debug = True  # Turn on auto reloader and debugger
    app.config['SQLALCHEMY_ECHO'] = True  # Show SQL commands created
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
    app.run()
```

templates/flask_sql_alchemy_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Cafe List</title>
</head>
<body>
<h1>Cafe List</h1>
{% if not item_list %}
  <p>No item found</p>
{% else %}
  <ul>
    {% for item in item_list %}<li>{{ item.name }}, {{ item.category }}, ${{ item.price }}</li>
    {% endfor %}
  </ul>
{% endif %}
</body>
</html>
```

run the controller and then

Try:

- http://127.0.0.1:5000: List all (3) items.
- http://127.0.0.1:5000/1: List item with id=1.
- http://127.0.0.1:5000/4: No such id.

Example 2: Using One-to-Many Relation

```
"""
flask_sql_alchemy_models_2: Flask-SQLAlchemy EG 2 - Define Models and Database Interface
```

```
"""
from flask_sqlalchemy import SQLAlchemy  # Flask-SQLAlchemy

# Flask-SQLAlchemy: Initialize
db = SQLAlchemy()


class Supplier(db.Model):
    """
    Define model 'Supplier' mapped to table 'supplier' (default lowercase).
    """
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50), nullable=False)
    items = db.relationship('Cafe', backref='supplier', lazy='dynamic')
        # Relate to objects in Cafe model
        # backref: a Cafe instance can refer to this as 'a_cafe_instance.supplier'

    def __repr__(self):
        return "<Supplier(%d, %s)>" % (self.id, self.name)


class Cafe(db.Model):
    """
    Define model 'Cafe' mapped to table 'cafe' (default lowercase).
    """
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    category = db.Column(db.Enum('tea', 'coffee', name='cat_enum'), default='coffee')
    name = db.Column(db.String(50))
    price = db.Column(db.Numeric(precision=5, scale=2), default=999.99)
    supplier_id = db.Column(db.Integer, db.ForeignKey('supplier.id'))
        # Via 'backref', a Cafe instance can refer to its supplier as 'a_cafe_instance.supplier'

    def __repr__(self):
        return "Cafe(%d, %s, %s, %5.2f, %d)" % (
                self.id, self.category, self.name, self.price, self.supplier_id)


def load_db(db):
    """Load the database tables and records"""
    # Drop and re-create all the tables
    db.drop_all()
    db.create_all()

    # Insert single row via add(instance) and commit
    db.session.add(Supplier(name='AAA Corp', id=501))
    db.session.add(Supplier(name='BBB Corp', id=502))
    db.session.commit()

    # Insert multiple rows via add_all(list_of_instances) and commit
    db.session.add_all([
            Cafe(name='Espresso', price=3.19, supplier_id=501),
            Cafe(name='Cappuccino', price=3.29, supplier_id=501),
            Cafe(name='Green Tea', category='tea', price=2.99, supplier_id=502)])
    db.session.commit()

    # You can also add record thru relationship
    _item = Cafe(name='latte', price=3.99)  # without the supplier_id
    _supplier = Supplier(name='ABC Corp', id=503, items=[_item])
        # OR:
        # _supplier = Supplier(name='XXX Corp', id=503)
```

```
        # _supplier.items.append(_item)
    db.session.add(_supplier)  # also add _item
    db.session.commit()
```

flask_sql_alchemy_controller-2.py

```
"""
fsqlalchemy_eg2_controller: Flask-SQLAlchemy EG 2 - Using one-to-many relation.
"""
from flask import Flask, render_template, redirect, flash, abort
from flask_sql_alchemy_models-2 import db, Supplier, Cafe, load_db  # Flask-SQLAlchemy

# Flask: Initialize
app = Flask(__name__)


# Flask-SQLAlchemy: Initialize
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://bill:passpass@localhost:3306/testdb'
#app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://bill:passpass@localhost:5432/testdb'
db.init_app(app)   # Bind SQLAlchemy to this Flask app

# Create the database tables and records inside a temporary test context
with app.test_request_context():
    load_db(db)


@app.route('/')
@app.route('/<id>')
def index(id=None):
    if id:
        _items = Cafe.query.filter_by(id=id).all()
    else:
        _items = Cafe.query.all()


    if not _items:
        abort(404)


    return render_template('flask_sql_alchemy_list-2.html', item_list=_items)


if __name__ == '__main__':
    app.config['SQLALCHEMY_ECHO'] = True
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
    app.debug = True
    app.run()
```

templates/flask_sql_alchemy_list-2.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Cafe List</title>
</head>
<body>
<h1>Cafe List</h1>
<ul>
  {% for item in item_list %}
  <li>{{ item.name }}, {{ item.category }}, ${{ item.price }}, {{ item.supplier.name }}</li>
  {% endfor %}
```

```
</ul>
</body>
</html>
```

Two queries was issued for each item in Cafe:

```
SELECT cafe.id AS cafe_id, cafe.category AS cafe_category, cafe.name AS cafe_name,
  cafe.price AS cafe_price, cafe.supplier_id AS cafe_supplier_id
FROM cafe
WHERE cafe.id = %s
('1',)

SELECT supplier.id AS supplier_id, supplier.name AS supplier_name
FROM supplier
WHERE supplier.id = %s
(501,)
```

run the controller and then

Try:

- http://127.0.0.1:5000: List all (3) items.
- http://127.0.0.1:5000/1: List item with id=1.
- http://127.0.0.1:5000/4: No such id.

Part 4: RESTful Web Services

An API (Application Programming Interface) is a set of routines, protocols and tools for building software applications. It exposes a software component in terms of its operations, input and output, that are independent of their implementation. A Web Service is a web application that is available to your application as if it was an API.

There are several protocols that clients can use to communicate with a web service or remote API, such as RPC (Remote Procedure Call), SOAP (Simplified Object Access Protocol) and REST (Representational State Transfer). REST has emerged as the favourite nowadays.

REST is a programming pattern which describes how data should be transferred between client and server over the network. REST specifies a set of design constraints that leads to higher performance and better maintainability. These constraints are: client-server, stateless, cacheable, layer system, uniform interface and code-on-demand.
If your web services conform to the REST constraints, and can be used in standalone (i.e., does not need an UI), then you have a RESTful Web Service API, or RESTful API in short. RESTful API works like a regular API, but delivers through a web server.

RESTful API is typically accessible through a URI. The most common scheme is to use the various HTTP request method to perform **CRUD (Create-Read-Update-Delete)** database operations. For example,

- GET request to /api/user/ to list ALL the users (Database READ).
  - Request data: NIL
  - Response data: a list of users (in JSON)
  - Response Status Code for success: 200 OK
  - Response Status Code for failure: 401 Unauthorized (login required), 403 No Permissions (role required) - these status codes are applicable to all requests.
- POST request to /api/user/ to create a new user (Database CREATE).
  - Request data: a new user (in JSON)
  - Response data: URL of the created item, or auto-increment ID, or the created item (in JSON)

    - Response Status Code for success: 201 Created
    - Response Status Code for failure: 400 Bad Request (invalid or missing input), 401 Unauthorized, 403 No Permissions
- GET request to /api/user/<id> to list ONE user with <id> (Database READ).
    - Request data: NIL
    - Response data: a user (in JSON)
    - Response Status Code for success: 20O OK
    - Response Status Code for failure: 404 Not Found, 401 Unauthorized, 403 No Permissions
- PUT or PATCH request to /api/user/<id> to update ONE user with <id> (Database UPDATE).
    - Request data: selected fields of a user (in JSON)
    - Response data: URL of the updated item, or the updated item (in JSON)
    - Response Status Code for success: 200 OK
    - Response Status Code for failure: 400 Bad Request (invalid or missing input), 404 Not Found, 401 Unauthorized, 403 No Permissions
- DELETE request to /api/user/<id> to delete ONE user with <id> (Database DELETE).
    - Request data: NIL
    - Response data: NIL
    - Response Status Code for success: 204 No Content
    - Response Status Code for failure: 404 Not Found, 401 Unauthorized, 403 No Permissions
- GET request to /api/user/me to list the current user (Database READ).
- GET request to /api/course/<code>/student/ to list all students of the given course code.
- POST request to /api/course/<code>/student/<id> to add a student to the given course code.

The request data (for POST, PATCH or PUT) and the response data (for GET) could use 'transport format' of text, HTML/XML, JSON, or other formats. **JSON has become the most common data format**, for its simplicity in representing objects (over XML) and its close ties to the client-side JavaScript programming language.

Flask- API Library extensions

- **flask-marshmallow:** the de facto library for serialization/deserialization. Data stored in the database are not stored in json but REST uses json so we need a library to help with converting data to and json formats as and when needed.
- Restless – To speed writing API's
- Flask-RESTful – to speed writing RESTFUL APIS

Marshmallow

We will be using Python package marshmallow (@https://marshmallow.readthedocs.org/en/latest/) for object serialization/deserialization and field validation.

To install marshmallow:

```
# Activate your virtual environment
(venv)$ pip install marshmallow

(venv)$ pip show --files marshmallow
Name: marshmallow
Version: 3.10.0
Summary: A lightweight library for converting complex datatypes to and from native Python datatypes.
Home-page: https://github.com/marshmallow-code/marshmallow
Author: Steven Loria
Author-email: sloria1@gmail.com
License: MIT
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires:
Required-by:
Files:
```

```
marshmallow-3.10.0.dist-info/INSTALLER
marshmallow-3.10.0.dist-info/LICENSE
marshmallow-3.10.0.dist-info/METADATA
marshmallow-3.10.0.dist-info/RECORD
marshmallow-3.10.0.dist-info/WHEEL
```

Read "marshmallow: quick start" @https://marshmallow.readthedocs.io/en/latest/quickstart.html for an excellent introduction to marshmallow.

Writing RESTFUL APIS

You can write RESTful APIs in Flask with no additional dependencies but the code would be longer and takes much effort. This is like comparing using Frameworks like Flask or writing everything in Python. To speed up the process of writing APIs, you can use FLASK Extensions. 2 of the most popular extensions are Flask-restful and Flask-restless. We will discuss each one of them:

- Roll your own API
- APIs with Flask-restful
- APIs with Flask-restless

Flask RESTful API – Roll Your Own

Flask supports RESTful API easily, by using URL variable in the route, e.g.,@app.route('/api/<version>/users/<user_id>').

Example 1: Handling GET Request

```
"""
resteg1_get.py: HTTP GET request with JSON response
"""
import simplejson as json  # Needed to jsonify Numeric (Decimal) field
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy  # Flask-SQLAlchemy
from marshmallow import Schema


app = Flask(__name__)
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Needed for CSRF
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://bill:passpass@localhost:3306/testdb'
db = SQLAlchemy(app)

# Define Model mapped to table 'cafe'
class Cafe(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    category = db.Column(db.Enum('tea', 'coffee', name='cat_enum'), nullable=False, default='coffee')
    name = db.Column(db.String(50), nullable=False)
    price = db.Column(db.Numeric(precision=5, scale=2), nullable=False)
        # 'json' does not support Numeric; need 'simplejson'

    def __init__(self, category, name, price):
        """Constructor: id is auto_increment"""
        self.category = category
        self.name = name
        self.price = price

# Drop, re-create all the tables and insert records
db.drop_all()
db.create_all()
```

```
db.session.add_all([Cafe('coffee', 'Espresso', 3.19),
                    Cafe('coffee', 'Cappuccino', 3.29),
                    Cafe('coffee', 'Caffe Latte', 3.39),
                    Cafe('tea', 'Green Tea', 2.99),
                    Cafe('tea', 'Wulong Tea', 2.89)])
db.session.commit()


# We use marshmallow Schema to serialize our database records
class CafeSchema(Schema):
    class Meta:
        fields = ('id', 'category', 'name', 'price')  # Serialize these fields

item_schema = CafeSchema()            # Single object
items_schema = CafeSchema(many=True)  # List of objects


@app.route('/api/item/', methods=['GET'])
@app.route('/api/item/<int:id>', methods=['GET'])
def query(id = None):
    if id:
        item = Cafe.query.get(id)


        if item is None:
            return jsonify({'err_msg': ["We could not find item '{}'".format(id)]}), 404
        else:
            result = item_schema.dump(item)  # Serialize object
                # dumps() does not support Decimal too
                # result: MarshalResult(data={'name': 'Espresso', 'id': 1, 'price': Decimal('3.19'), 'category':
'coffee'},
                #          errors={})
            return jsonify(result)  # Uses simplejson


    else:
        items = Cafe.query.limit(3)  # don't return the whole set
        result = items_schema.dump(items)  # Serialize list of objects
            # Or, item_schema.dump(items, many=True)
        return jsonify(result) #was result.data


if __name__ == '__main__':
    # Turn on debug only if launch from command-line
    app.config['SQLALCHEMY_ECHO'] = True
    app.debug = True
    app.run()
```

To try:

Notes: In order to jsonify Decimal (or Numeric) field, we need to use simplejson to replace the json of the standard library. To install simplejson, use "pip install simplejson". With "import simplejson as json", the jsonify() invokes simplejson. Marshmallow's dumps() does not support Decimal field too.
Try these URLs and observe the JSON data returned. Trace the request/response messages using web browser's developer web console.

1. GET request: http://localhost:5000/api/item/
2. GET request: http://localhost:5000/api/item/1
3. GET request: http://localhost:5000/api/item/6

Flask-RESTful Extension

Reference: Flask-RESTful @ http://flask-restful-cn.readthedocs.org/en/0.3.4/.
Flask-RESTful is an extension for building REST APIs for Flask app, which works with your existing ORM.

Installing Flask-RESTful

```
# Activate your virtual environment
(venv)$ pip install flask-restful
Successfully installed aniso8601-1.2.0 flask-restful-0.3.5 python-dateutil-2.6.0 pytz-2016.10
```

```
(venv)$ pip show flask-restful
Name: Flask-RESTful
Version: 0.3.8
Summary: Simple framework for creating REST APIs
Home-page: https://www.github.com/flask-restful/flask-restful/
Author: Twilio API Team
Author-email: help@twilio.com
License: BSD
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires: aniso8601, six, Flask, pytz
Required-by:
```

Flask-Restful Example 1: Using Flask-Restful Extension

```
"""
flask_restful.py: Flask-Restful Example 1 - Using Flask-Restful Extension
"""
from flask import Flask, abort
from flask_restful import Api, Resource


class Item(Resource):
    """
    For get, update, delete of a particular item via URL /api/item/<int:item_id>.
    """
    def get(self, item_id):
        return 'reading item {}'.format(item_id), 200


    def delete(self, item_id):
        return 'delete item {}'.format(item_id), 204  # No Content


     def put(self, item_id):  # or PATCH
        """Request data needed for update"""
        return 'update item {}'.format(item_id), 200


class Items(Resource):
    """
    For get, post via URL /api/item/, meant for list-all and create new.
    """
    def get(self):
        return 'list all items', 200


    def post(self):
        """Request data needed for create"""
        return 'create a new post', 201  # Created


app = Flask(__name__)
api_manager = Api(app)
# Or,
#api_manager = Api()
#api_manager.init_app(app)

api_manager.add_resource(Item, '/api/item/<item_id>', endpoint='item')
api_manager.add_resource(Items, '/api/item/', endpoint='items')
    # endpoint specifies the view function name for the URL route



if __name__ == '__main__':
```

```
    app.run(debug=True)
```

http://localhost:5000/api/item/4
http://localhost:5000/api/item

Flask-Restless Extension

After Note: Although Flask-Restless requires fewer codes, but it is not as flexible as Flask-Restful.
Flask-Restless is an extension capable of auto-generating a whole RESTful API for your SQLAlchemy models with support for GET, POST, PUT, and DELETE.

To install Flask-Restless Extension:

```
# Activate your virtual environment
(venv)$ pip install Flask-Restless
Successfully installed Flask-Restless-0.17.0 mimerender-0.6.0 python-mimeparse-1.6.0

(venv)$ pip show Flask-Restless
Name: Flask-Restless
Version: 0.17.0
Summary: A Flask extension for easy ReSTful API generation
Home-page: http://github.com/jfinkels/flask-restless
Author: Jeffrey Finkelstein
Author-email: jeffrey.finkelstein@gmail.com
License: GNU AGPLv3+ or BSD
Location: /Users/damianigbe/Library/Python/3.8/lib/python/site-packages
Requires: python-dateutil, sqlalchemy, flask, mimerender
Required-by:
```

Example: Using Flask-Restless Extension

```
"""
flask_restless.py: Testing Flask-Restless to generate RESTful API
"""
import simplejson as json
from flask import Flask, url_for
from flask_sqlalchemy import SQLAlchemy  # Flask-SQLAlchemy
from flask_restless import APIManager    # Flask-Restless


app = Flask(__name__)
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Needed for CSRF
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://bill:passpass@localhost:3306/testdb'
db = SQLAlchemy(app)


# Define Model mapped to table 'cafe'
class Cafe(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    category = db.Column(db.Enum('tea', 'coffee', name='cat_enum'), nullable=False, default='coffee')
    name = db.Column(db.String(50), nullable=False)
    price = db.Column(db.Numeric(precision=5, scale=2), nullable=False)


    def __init__(self, category, name, price):
        """Constructor: id is auto_increment"""
        self.category = category
        self.name = name
        self.price = price


# Drop, re-create all the tables and insert records
db.drop_all()
db.create_all()
db.session.add_all([Cafe('coffee', 'Espresso', 3.19),
                    Cafe('coffee', 'Cappuccino', 3.29),
                    Cafe('coffee', 'Caffe Latte', 3.39),
                    Cafe('tea', 'Green Tea', 2.99),
```

```
                    Cafe('tea', 'Wulong Tea', 2.89)])
db.session.commit()


# Create the Flask-Restless API manager
manager = APIManager(app, flask_sqlalchemy_db=db)


# Create API endpoints, which will be available at /api/<tablename>,
# by default. Allowed HTTP methods can be specified as well.
manager.create_api(Cafe, methods=['GET', 'POST', 'PUT', 'DELETE'])


if __name__ == '__main__':
    # Turn on debug only if launch from command-line
    app.config['SQLALCHEMY_ECHO'] = True
    app.debug = True
    app.run()
```

To send POST/PUT/DELETE requests, you can use the command-line curl (which is rather hard to use); or browser's extension such as Firefox's HttpRequester, or Chrome's Advanced REST client or postman.
You can use curl to try the RESTful API (See http://flask-restless.readthedocs.org/en/latest/requestformat.html on the request format):

```
# GET request to list all the items
$ curl --include --header "Accept: application/json" --header "Content-Type: application/json" --request GET
http://127.0.0.1:5000/api/cafe

HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 609
Link: <http://127.0.0.1:5000/api/cafe?page=1&results_per_page=10>; rel="last"
Link: <http://127.0.0.1:5000/api/cafe?page=1&results_per_page=10>; rel="last"
Vary: Accept
Content-Type: application/json
Server: Werkzeug/1.0.1 Python/3.8.2
Date: Sat, 16 Jan 2021 03:38:43 GMT

{
  "num_results": 5,
  "objects": [
    {
      "category": "coffee",
      "id": 1,
      "name": "Espresso",
      "price": 3.19
    },
    {
      "category": "coffee",
      "id": 2,
      "name": "Cappuccino",
      "price": 3.29
    },


# GET request for one item
$ curl --include --header "Accept: application/json" --header "Content-Type: application/json" --request GET
http://127.0.0.1:5000/api/cafe/3

HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 83
Vary: Accept
Content-Type: application/json
Server: Werkzeug/1.0.1 Python/3.8.2
Date: Sat, 16 Jan 2021 03:39:33 GMT

{
  "category": "coffee",
  "id": 3,
  "name": "Caffe Latte",
  "price": 3.39
}


# POST request to add one item
$ curl --include --header "Accept: application/json" --header "Content-Type: application/json" --request POST --
data '{"category":"coffee", "name":"coffee xxx", "price":1.01}'  http://127.0.0.1:5000/api/cafe
```

```
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 82
Location: http://127.0.0.1:5000/api/cafe/6
Vary: Accept
Content-Type: application/json
Server: Werkzeug/1.0.1 Python/3.8.2
Date: Sat, 16 Jan 2021 03:40:33 GMT

{
  "category": "coffee",
  "id": 6,
  "name": "coffee xxx",
  "price": 1.01
}


# DELETE request to remove one item
$ curl --include --header "Accept: application/json" --header "Content-Type: application/json" --request DELETE
http://127.0.0.1:5000/api/cafe/4

HTTP/1.0 204 NO CONTENT
Content-Type: application/json
Vary: Accept
Content-Type: application/json
Server: Werkzeug/1.0.1 Python/3.8.2
Date: Sat, 16 Jan 2021 03:41:47 GMT

# PUT (or PATCH) request to update one item
$ curl --include --header "Accept: application/json" --header "Content-Type: application/json" --request PUT --
data '{"price":9.99}' http://127.0.0.1:5000/api/cafe/1
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 80
Vary: Accept
Content-Type: application/json
Server: Werkzeug/1.0.1 Python/3.8.2
Date: Sat, 16 Jan 2021 03:42:36 GMT

{
  "category": "coffee",
  "id": 1,
  "name": "Espresso",
  "price": 9.99
}
```

Flask-Restless generates web services that are RESTful API. They are client-server (HTTP), stateless (HTTP is stateless), cacheable (browser), uniform interface (JSON input, JSON output, consistent URLs for GET (read), POST (insert or create), PUT (update), DELETE (delete), supporting CRUD (create-read-update-delete) operations).

To try:

From the web browser you can do:
http://127.0.0.1:5000/api/cafe
http://127.0.0.1:5000/api/cafe/1
http://127.0.0.1:5000/api/cafe/2

Part 5:

Docker-rize and run a Microservice in Kubernetes

In this lab we will create a restful api microservice offering books. We will create the API, Create a Docker Image, tag the image and upload to DockerHub, create a deployment file for it in Kubernetes and then run the microservice using Kubernetes.

Step 1: Create the Directory for the API

mkdir FlaskBookApi/

cd FlaskBookApi/

Step 2: Clone the API repository

git clone https://github.com/igbedo/rest-api-microservice-docker.git

cd rest-api-microservice-docker/

Step 3: Create the Docker Image using Dockerfile

The Dockerfile is already provided so now you need to build the Docker image

docker build -t flaskbookapi:1.0 .

Step 4: Test the microservice by running the docker container

Now, finally! Fire up a container with the image we just built!

```
$ docker run -p 5000:5000 --name FlaskBookAPI flaskbookapi:1.0
* Serving Flask app "api" (lazy loading)
* Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
  * Debug mode: off
  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Step 5: Test that the API is running as expected

Open another terminal on your laptop and curl the api just deployed on a docker container. You can query for all the books and you can query individual books as follows.

```
controlplane $ curl localhost:5000/books
{"books": [{"id": 1, "title": "Zero to One", "author": "Peter Thiel", "length": 195, "rating": 4.17}, {"id": 2,
"title": "Atomic Habits ", "author": "James Clear", "length": 319, "rating": 4.35}]}

controlplane $ curl localhost:5000/books/1
{"book": {"id": 1, "title": "Zero to One", "author": "Peter Thiel", "length": 195, "rating": 4.17}}
controlplane $ curl localhost:5000/books/2
{"book": {"id": 2, "title": "Atomic Habits ", "author": "James Clear", "length": 319, "rating": 4.35}}
```

If you try to query a book that does not exists it will give an error

controlplane $ curl localhost:5000/books/3

Run the API using Kubernetes deployment Object

Step 1: tag the image and upload to docker registry

docker tag flaskbookapi:1.0 igbedo/flaskbookapi:1.0

Step 2: Login to DockerHub and upload the image to DockerHub

docker login

docker push igbedo/flaskbookapi:1.0

Step 3: Create Deployment file and test that the Pod is running correctly

Create a deployment file to create the api

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flaskbookapi
  labels:
    app: flaskbookapi
spec:
  replicas: 3
  selector:
    matchLabels:
      app: flaskbookapi
  template:
    metadata:
      labels:
        app: flaskbookapi
    spec:
      containers:
      - name: flaskbookapi
        image: igbedo/flaskbookapi:1.0
        ports:
        - containerPort: 5000
```

```
Kubectl create -f deploy.yaml

ontrolplane $ kubectl get pods -o wide
NAME                             READY   STATUS    RESTARTS   AGE   IP           NODE     NOMINATED NODE
READINESS GATES
flaskbookapi-8478f749c9-dvhd9   1/1     Running   0          48s   10.244.1.5   node01   <none>           <none>
flaskbookapi-8478f749c9-h6hzg   1/1     Running   0          48s   10.244.1.3   node01   <none>           <none>
flaskbookapi-8478f749c9-txjmt   1/1     Running   0          48s   10.244.1.4   node01   <none>           <none>

controlplane $ curl 10.244.1.5:5000/books
{"books": [{"id": 1, "title": "Zero to One", "author": "Peter Thiel", "length": 195, "rating": 4.17}, {"id": 2,
"title": "Atomic Habits ", "author": "James Clear", "length": 319, "rating": 4.35}]}

controlplane $ curl 10.244.1.5:5000/books/1
{"book": {"id": 1, "title": "Zero to One", "author": "Peter Thiel", "length": 195, "rating": 4.17}}
controlplane $ curl 10.244.1.5:5000/books/2
{"book": {"id": 2, "title": "Atomic Habits ", "author": "James Clear", "length": 319, "rating": 4.35}}

Step4: create a service to point to the deployment and test that the service is working as expected

apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: flaskbookapi
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000

Kubectl create -f service.yaml

controlplane $ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    18m
my-service   ClusterIP   10.107.228.226   <none>        5000/TCP   3s

controlplane $ kubectl get ep
NAME         ENDPOINTS                                  AGE
kubernetes   172.17.0.9:6443                            18m
```

```
my-service   10.244.1.3:5000,10.244.1.4:5000,10.244.1.5:5000   10s

ontrolplane $ curl 10.107.228.226:5000/books
{"books": [{"id": 1, "title": "Zero to One", "author": "Peter Thiel", "length": 195, "rating": 4.17}, {"id": 2,
"title": "Atomic Habits ", "author": "James Clear", "length": 319, "rating": 4.35}]}

controlplane $ curl 10.107.228.226:5000/books/1
{"book": {"id": 1, "title": "Zero to One", "author": "Peter Thiel", "length": 195, "rating": 4.17}}
controlplane $ curl 10.107.228.226:5000/books/2
{"book": {"id": 2, "title": "Atomic Habits ", "author": "James Clear", "length": 319, "rating": 4.35}}
```