INFO3180 Tutorial 3

In this tutorial we will take our first look at Flask add-ons and specifically two add-ons that can be used for creating forms and working with databases. We'll also look at how to use "**`pip freeze`**" to manage your **`requirements.txt`** file to ensure that your extensions are available to the production version of your code.

Flask Add-ons

The Flask micro-framework can be extended using packages (or add-ons). Flask calls these add-ons "extensions" and they are usually installed using the "pip" command. For a list of supported Add-ons visit http://flask.pocoo.org/extensions/ .

For example, if you wanted to use the really useful extension called "Flask-DebugToolbar", you would run the following command from within your activated project:

```
source venv/bin/activate
pip install Flask-DebugToolbar
```

Note: Before installing new extensions using **`pip`**, always remember to activate your **`virtualenv`** for your project.

A Quick Refresher on Databases

1. A DataBase Management Server (sometimes DBMS for short) is a type of software that can manage one or more databases.
2. Database management servers are broadly categorized as being SQL Databases or being NoSQL/NewSQL based

3. Examples of SQL Databases include Mysql/Mariadb, Postgresql, Oracle, IBM DB2, Microsoft SQL Server.
4. Examples of NoSQL based servers include MongoDB, Redis, ZODB, CouchDB, Apache Cassandra and Riak.

A tour relational databases

Let's look at 3 open source relational databases, PostgreSQL, MySQL and Sqlite3. We'll use Cloud9 for this exercise.

Cloud9 ships with PostgreSQL and MySQL already installed. This is how you start them:

```
# start MySQL. Will create an empty database on first start
$ mysql-ctl start

# stop MySQL
$ mysql-ctl stop

# run the MySQL interactive shell
$ mysql-ctl cli
```

```
# Start PostgreSQL
sudo service postgresql start

# Run the PostgreSQL interactive shell
psql
```

Now let us look at how to list and create databases in each of the three servers.

Task	MySQL	Sqlite3	PostgreSQL
Show available databases	`show database;`	not applicable (keeps databases .db files)	`\l or \list`
select a database by name.	`use test;`	`sqlite3 library.db`	`\c databasename`
show tables in database	`show tables;`	`.tables`	`\dt`
create a table	`create table book( id INT NOT NULL AUTO_INCREMENT, title VARCHAR(100) NOT NULL, author VARCHAR(40) NOT NULL, publication_date DATE, PRIMARY KEY ( id ) );`	`CREATE TABLE book(Id INTEGER PRIMARY KEY, author TEXT, title TEXT, publication_date TEXT);`	`CREATE TABLE book ( id SERIAL, author text NOT NULL, title text NOT NULL, publication_date DATE, PRIMARY KEY(id) );`

As you can see from this table, there is some variation in the syntax of different database systems.

Discussion Questions:

1. If you started a project using sqlite3 and wanted to continue it with PostgreSQL, what are some of things that might need to change in your code?
2. What if someone told you that they needed your code to always work, no matter what database they used, how would you write code that would work with any database?

SQLAlchemy (Database Independent Code)

There is a library for Python called SQLAlchemy which acts as an abstraction layer on top of various databases. Using SQLAlchemy's Object Relational Mapper (ORM) it

is possible to interact with a database and it's tables and rows as if they were Python objects.

We've provided a demo of sqlalchemy (https://github.com/uwi-info3180/sqlalchemy_demo) that works with SQLite3 (in the lab you will be asked to make this same demo work with PostgreSQL and MySQL).

Exploring SQLAlchemy's ORM

Let's take a look at SQLAlchemy's Object Relational Mapper (ORM). To get started clone the repository, activate a **`virtualenv`** and install the requirements using **`pip`**.

```
git clone https://github.com/uwi-info3180/sqlalchemy_demo
cd sqlalchemy_demo
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt
```

Note:

Compare the **`sqlite_ex.py`** (https://github.com/uwi-info3180/sqlalchemy_demo/blob/master/sqlite_ex.py) to **`sqlalchemy_declarative.py()`**. Notice that both files create a database containing "persons" and "addresses", however one presents the information as python objects.

For your convenience we've made it possible to interactively work with sqlalchemy, run the following commands to get started (make sure you're still in the virtualenv)

```
python sqlalchemy_create.py
python sqlalchemy_interactive.py
```

We import all the code and objects for you, you'll see a prompt similar to this:

What happens when you run the following command at the prompt?

```
session.query(Person).all()
```

Now try this

```
person = session.query(Person).first()
person.name
```

Note:

The **`.first()`** method returns the first result from the "query". In the example above this will select the first person retrieved from the query.

You can also find the person's address:

```
address = session.query(Address).filter(Address.person ==
person).one()
address.post_code
```

Note:

The **`.one()`** at the end is important here, if you leave it out it returns a "query" object. Adding the **`.one()`** method returns a single item from the query, while **`.all()`** returns a list of items.

Finally, try adding a new person:

```
new_person = Person(name='john brown')
session.add(new_person)
session.commit()
```

And now run the query for all persons again

```
session.query(Person).all()
```

To close the interactive session press **`Ctrl + d`** or type '**`q`**'.

Databases and Flask

Let us look at the Flask add-on called **Flask-SQLAlchemy**.

As you know, SQLAlchemy maps tables and relations to representative objects. This makes it possible to switch between different relational database backends without having to rewrite your backend code.

To get a better understanding let's look at a simple database application.

Preparation

Before starting we will need to install a database and the **`flask-sqlalchemy`** add-on. Now let us start the PostgresSQL server on Cloud 9 and connect to a database, and run the **`\dt`** command to show tables and rows (relations):

```
$ sudo service postgresql start
$ sudo sudo -u postgres psql
postgres=# CREATE DATABASE db_app;
postgres=# \c db_app;
postgres=# \dt
No relations found.
```

We haven't created any tables in our database as yet, hence the "No relations found" message when we tried to show our tables.

Let us also create a password for the **postgres** user that way we can connect to it from our Flask application. To do this:

```
postgres=# \password postgres
Enter new password:
```

Note: Remember this password as you will need it in the connection string for Flask-SQLAlchemy.

To quit the postgres command line type the following:

```
postgres=# \q
```

Now we'll use the flask_starter (https://github.com/uwi-info3180/flask_starter) and we'll extend it. Start by cloning the flask_starter code.

```
$ git clone https://github.com/uwi-info3180/flask_starter db_app
$ cd db_app
$ virtualenv venv
$ source venv/bin/activate
```

Add "**Flask-SQLAlchemy**" and "**psycopg2**" to the **requirements.txt** file.

Note: psycopg2 for postgresql

psycopg2 is a database connector for PostgreSQL. SQLAlchemy depends on psycopg2 when connecting to PostgreSQL.

Now install the requirements by running the following command

```
$ pip install -r requirements.txt
```

Customizing the app

Firstly we'll add relevant sqlalchemy code to '**app/__init__.py**'. The highlighted lines show the additional below.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SECRET_KEY'] = 'some$3cretKey'
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://
postgres:yourpasswordfromabove@localhost/db_app'
db = SQLAlchemy(app)

from app import views, models
```

In the '**app**' folder we'll add an additional file called '**models.py**' to our project, this is a good place to define our database object models, we'll start with a single model called "**User**":

```
from . import db

class User(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

Each database field is considered to be a column, note the use of **`db.Column()`**.

For discussion

Look at the **`SQLALCHEMY_DATABASE_URI`** configuration option above, this is where we configure the specific database engine.

1. Try to identify the username, the servername and the database in the URI string. **Ans.** **`dialect+driver://username@host:port/database`**
2. There is no password specified in our URI, do you know how to specify a password? **Ans.** **`dialect+driver://username:password@host:port/database`**
3. How would you switch this from postgresql to mysql? **Ans.** **`mysql://username:password@host/database`**

Initializing the database

You must initialize the database. You can do so using the following commands at the python prompt:

```
>>> from app import db
>>> db.create_all()
```

Note: If you would like to remove/drop all your tables, you can use **`db.drop_all()`**. After this go back to '**`psql`**' and run **`\dt`** again, this time you'll see evidence of your "**`user`**" table, looking something like this:

```
sudo sudo -u postgres psql
postgres=# \c db_app
postgres=# \dt
        List of relations
 Schema | Name | Type  | Owner
--------+------+-------+--------
 public | user | table | postgres
(1 row)
```

Populating the database

Let's quickly create a few users (we'll just type this on the python prompt)

Note: Activate venv

Remember to have your **`virtualenv`** activated so you're using the right python environment.

Launch the python prompt and run the following (or place it in a file and run it as a standalone script):

```
>>> from app.models import User
```

```
>>> from app import db
>>> admin = User('admin', 'admin@example.com')
>>> guest = User('guest', 'guest@example.com')
# add and commit the new users to the database
>>> db.session.add(admin)
>>> db.session.add(guest)
>>> db.session.commit()
# You can check that these were added doing
>>> db.session.query(User).all()
```

You should see something like this:

```
>>> db.session.query(User).all()
[<User u'admin'>, <User u'guest'>]
```

You should now have two new users in your database. Let's create a new route and custom view in the **app/views.py** file that shows our first user, call the new route "**/person**" and view method called "**person**":

```
from app import app
from flask import render_template, request, redirect, url_for
from app import db
from app.models import User


##
# Routing for your application.
##


@app.route('/')
```

```
def home():
    """Render website's home page."""
    return render_template('home.html')


@app.route('/person')
def person():
    first_user = db.session.query(User).first()
    return "username: {}, email: {}".format(first_user.username,
first_user.email)


@app.route('/about/')
def about():
    """Render the website's about page."""
    return render_template('about.html')


...
```

Now run the app

```
$ python run.py
```

and visit **http://yourwebsiteurl/person** to see the output of your new route.

For discussion

In our example we have only used "String" types. What other types are available?

(hint: look at the documentation https://pythonhosted.org/Flask-SQLAlchemy/models.html?highlight=date#simple-example)

How would you add an additional field to the user table?

For example a "high_score" field?

Note: Migrations

Flask-SQLAlchemy doesn't automatically alter the database schema, therefore if you change your model and run **`db.create_all()`** again, it will wipe all your data. In the world of object relational mappers (ORMs) the idea of updating your database schema non-destructively is referred to as "**migration**". We will discuss this in a future tutorial.

Webforms

Let's extend our database application to support a webform. We do this by using an add-on called **Flask-WTF**, which is a wrapper around the python wtforms library. Add it to your requirements.txt.

Now install using **`pip`**

```
pip install -r requirements.txt
```

Next we'll create a "**`app/forms.py`**" file.

```
from flask_wtf import Form
from wtforms.fields import TextField
# other fields include PasswordField
from wtforms.validators import Required, Email

class EmailPasswordForm(Form):
    username = TextField('Username', validators=[Required()])
    email = TextField('Email', validators=[Required(), Email()])
```

And import forms into **`views.py`**.

```
from app import app
from flask import render_template, request, redirect, url_for
from app import db
from app.models import User

from .forms import EmailPasswordForm

@app.route('/theform', methods=["GET", "POST"])
def theform():
    form = EmailPasswordForm()
    return render_template('theform.html', form=form)
```

Finally create a template called **theform.html** in the templates section.

```
{% extends "base.html" %}

{% block js %}
{% endblock %}

{% block main %}
<div class="container">
  <form action="{{ url_for('theform') }}" method="post">
    <label for="username">username:</label> {{ form.username }}<br
/>
     <label for="email">email:</label> {{ form.email }}<br />
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
</div>
{% endblock %}
```

Visiting your app at **`urlforyourapp/theform`**, will now show the form.

Discussion:

There's a lot that is still left to do, for example, you'll want to

1. Verify the fields and show errors
2. Connect the input of this form to your database
3. Making the form secure

We'll be looking at this in the future.