Python

Developing Web Applications with Flask

1. Introduction to Python-Flask Webapp Framework

References:

1. Flask @ http://flask.pocoo.org/; Flask Documentation & User Guide @ http://flask.pocoo.org/docs/; Flask API @http://flask.pocoo.org/docs/0.12/api/.
2. Werkzeug @ http://werkzeug.pocoo.org/.
3. Jinja2 @ http://jinja.pocoo.org/.
4. Italo Maia, Building Web Applications with Flask, Packt Pub., 2015.
5. Miguel Grinberg, Flask Web Development, O'Reilly, 2014.

Why Framework?

To build a complex webapp, you could *roll-your-own* (RYO) from scratch or build it over a *framework* (which defines the structure and provides a set of libraries for common tasks). Rolling-your-own means that you need to write ten-thousand lines of boiler-plate codes, that are already provided by a framework. Worse still, your codes are most likely messy, buggy, un-tested and un-maintainable - you can't write better codes than those who could build framework. On the other hand, using a framework means that you need to spend weeks or even months reading and understanding the framework, as each framework has it own "syntax" and, hence, requires a steep learning curve. Furthermore, there are just too many frameworks available and choosing the *right* framework turns out to be a difficult decision.

TABLE OF CONTENTS (HIDE)

There are many Python frameworks available, e.g., full-stack frameworks like Djiango, TurboGears, web2py; non-full-stack frameworks like Flask, Pyramid. You need to make your own decision to select a framework, which could be a hard choice.

This article describe the Python's flask framework.

Python-Flask Framework

Flask (@ http://flask.pocoo.org/) was created by Armin Ronacher in 2000 as a small, minimalistic and light-weight Python Webapp framework. It is so small to be called a micro-framework. Flask is actually a glue that sticks together two popular frameworks:

1. Werkzeug (@ http://werkzeug.pocoo.org/): a WSGI (Web Server Gateway Interface) library for Python, which includes a URL routing system, fully featured request and response objects and a powerful debugger. (WSGI is a specification for simple and universal interface between web servers and Python web applications.)
2. Jinja2 (@ http://jinja.pocoo.org/): a full-feature template engine for Python.

High-level tasks like database access, web form and user authentication are supported through "extensions".

Within the scope of MVC (Model-View-Controller) architecture, Werkzeug covers the Controller (C) and Jinja2 covers the View (V). Flask does not provide an integrated Model (M) layer, and lets you pick your database solution. A popular choice is Flask-SQLAlchemy with a ORM (Object-Relational Mapper) over a relational database such as MySQL or PostgreSQL.

In summary, the Flask framework provides:

- a WSGI compliance interface.
- URL routing and Request dispatching.
- Secure cookies and Sessions.
- a built-in Development Web Server and Debugger.
- Unit test client for unit testing that facilitates write-test-first.
- Jinja2 templates (tags, filters, macros, etc).

Via Flask, you can handle HTTP and AJAX requests, user sessions between requests, route requests to controllers, evaluate and validate the request data, and response with HTML or JSON, and so on.

Flask Documentation and Tutorials

Read the Flask documentation, quick start guide, and tutorials available at the Flask mother site @ http://flask.pocoo.org/.

2. Getting Started with Python-Flask

2.1 Installing Flask (under a Virtual Environment)

It is strongly recommended to develop Python application under a virtual environment. See "Virtual Environment".

We shall install Flask under a virtual environment (called `myflaskvenv`) under our project directory. You need to choose your own project directory and pick your Python version.

```
$ cd /path/to/project-directory      # Choose your project directory
$ virtualenv -p python3 venv  # For Python 3, or
$ virtualenv venv             # For Python 2

$ source venv/bin/activate    # Activate the virual environment
(venv)$ pip install flask     # Install flask using 'pip' which is symlinked to pip2 or pip3 (no sudo needed)
......
Successfully installed Jinja2-2.9.5 MarkupSafe-0.23 Werkzeug-0.11.15 click-6.7 flask-0.12 itsdangerous-0.24
(venv)$ pip show flask        # Check installed packages
Name: Flask
Version: 0.12
Summary: A microframework based on Werkzeug, Jinja2 and good intentions
Location: .../venv/lib/python3.5/site-packages
Requires: Werkzeug, itsdangerous, click, Jinja2
(venv)$ pip show Werkzeug
Name: Werkzeug
Version: 0.11.15
Summary: The Swiss Army knife of Python web development
Location: .../venv/lib/python3.5/site-packages
Requires:
(venv)$ pip show Jinja2
Name: Jinja2
Version: 2.9.5
Summary: A small but fast and easy to use stand-alone template engine written in pure python.
Location: .../venv/lib/python3.5/site-packages
Requires: MarkupSafe
(venv)$ deactivate  # Exit virutal environment
```

2.2 Using Eclipse PyDev IDE

Using a proper IDE (with debugger, profiler, etc.) is CRITICAL in software development.

Read HERE on how to install and use Eclipse PyDev for Python program development. This section highlights how to use Eclipse PyDev to develop Flask webapp.

Configuring Python Interpreter for virtualenv

To use the virtual environment created in the previous section, you need to configure a new Python Interpreter for your virtual environment via: Window ⇒ Preferences ⇒ PyDev ⇒ Interpreters ⇒ Python Interpreter ⇒ New ⇒ Enter a name and select the Python Interpreter for the virtual environment (e.g., `/path/to/project-directory/venv/bin/python`).

Build Import-List for Flask's Extension

To use Flask's extensions, you need to build the import list "`flask.ext`" via: Window ⇒ Preferences ⇒ PyDev ⇒ Interpreters ⇒ Python Interpreter ⇒ Select your Python interpreter ⇒ Forced Builtins ⇒ New ⇒ enter "`flask.ext`".

Write a Hello-world Python-Flask Webapp

As usual, create a new PyDev project via: New ⇒ Project ⇒ PyDev ⇒ PyDev Project. Then, create a PyDev module under the project via: New ⇒ PyDev Module.

As an example, create a PyDev project called `test-flask` with a module called `hello_flask` (save as `hello_flask.py`), as follows:

```
# -*- coding: UTF-8 -*-
"""
hello_flask: First Python-Flask webapp
"""
from flask import Flask  # From module flask import class Flask
app = Flask(__name__)    # Construct an instance of Flask class for our webapp

@app.route('/')   # URL '/' to be handled by main() route handler
def main():
    """Say hello"""
    return 'Hello, world!'

if __name__ == '__main__':  # Script executed directly?
    app.run()  # Launch built-in web server and run this Flask webapp
```

I shall explain how this script works in the next section.

Run the Webapp

To run the Python-Flask Webapp, right-click on the Flask script ⇒ Run As ⇒ Python Run. The following message appears on the console:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

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.

Debugging Python-Flask Webapps (Running on Flask's Built-in Web Server)

As usual, set a breakpoint by double-clicking on the left margin of the desired line ⇒ Debug As ⇒ Python Run. You can then trace the program, via Step-Over, Step-Into, Step-Out, Resume, or Terminate.

Try:

1. Set a breakpoint inside the `main()` function (e.g., at the `return` statement).
2. Start debugger. You shall see the message "pydev debugger: starting (pid: xxxx)" on the console.
3. From a web browser, issue URL `http://127.0.0.1:5000/` to trigger `main()` and hit the breakpoint.
4. Switch over to Eclipse, and Step-Over.... Resume to finish the function.

NOTE: If you encounter the message "warning: Debugger speedups using cython not found. Run '...command...' to build". Copy the command and run this command (under your virtual environment, if applicable).

PyDev's Interactive Python Console

To open an interactive console, press Ctrl-Alt-Enter (in the PyDev Editor); or click the "Open Console" button (under the "Console" Pane). Choose "Python console" (`PYTHONPATH` includes all projects in the workspace) or "Console for the currently active editor" (`PYTHONPATH` includes only this project in the workspace). Choose your Python Interpreter.

You can now run Python statements from the command prompt. For example,

```
# Check sys.path (resulted from PYTHONPATH)
>>> import sys
>>> sys.path

>>> import Hello
>>> runfile(......)
```

PyDev's Debug Console

To connect the interactive console to the debug session, goto "Window" ⇒ Preferences ⇒ PyDev ⇒ Interactive Console ⇒ Check "Connect console to debug session".

To use the "Debug Console", set a breakpoint somewhere in your Python script ⇒ Debug As ⇒ Python Run. When the breakpoint is reached, you can issue Python statements at the command prompt.

You might need to "select a frame to connect the console", choose the desired "stack frame" (from the method chain of the stack trace) of the "Debug" pane (under the "Debug" perspective).

(Advanced) Debugging Python-Flask Webapp Remotely (Running on Apache WSGI)

Reference: Remote Debugger @ http://www.pydev.org/manual_adv_remote_debugger.html.

Read this section only if you have setup Flask to run on Apache Web Server.

The steps are:

1. Start the Remote Debug Server: Switch into "Debug" perspective (Click the "Debug" perspective button; or Window menu ⇒ Open Perspective ⇒ Other ⇒ Debug), and click the "PyDev: Start the pydev server" (or select from the "PyDev" menu). You shall receive a message "Debug Server at port: 5678", in the Console-pane. In addition, in the Debug-pane, you should see a process "Debug Server".
2. Include the module `pydevd.py` into your `sys.path` (or `PYTHONPATH`). First, locate the module path (which is installation-dependent) under `eclipse/plugins/org.python.pydev_x.x.x/pysrc/pydevd.py`. There are several ways to add this module-path to `sys.path`:
    - In your `.wsgi`: Include

      ```
      sys.path.append('/path/to/eclipse/plugins/org.python.pydev_x.x.x.x/pysrc')
      ```

    - In your apache configuration file: Include

      ```
      WSGIDaemonProcess myflaskapp user=flaskuser group=www-data threads=5 python-path=/path/to/eclipse/plugins/org.python.pydev_x.x.x.x/pysrc
      ```

    - You might be able to set a system-wide environment variable for all processes in `/etc/environment`:

      ```
      PYTHONPATH='/path/to/eclipse/plugins/org.python.pydev_x.x.x.x/pysrc'
      ```

      Take note that variable expansion does not work here. Also there is no need to use the `export` command.
      Also take note that exporting the `PYTHONPATH` from a bash shell or under `~/.profile` probably has no effects on the WSGI processes.

    NOTES: You should also include this module-path in your Eclipse PyDev (to avoid the annoying module not found error): Window ⇒ Preferences ⇒ PyDev ⇒ Interpreters ⇒ Python Interpreters ⇒ Libraries ⇒ Add Folder.
3. Call `pydevd.settrace()`: Insert "`import pydevd; pydevd.settrace()`" inside your program. Whenever this call is made, the debug server suspend the execution and show the debugger. This line is similar to an initial breakpoint.
   NOTES: You need to remove `pydevd.settrace()` if your debug server is not running.

(Advanced) Reloading Modified Source Code under WSGI

Reference: `modwsgi` Reloading Source Code @ https://code.google.com/p/modwsgi/wiki/ReloadingSourceCode.

If you modify your source code, running under WSGI, you need to restart the Apache server to reload the source code.

On the other hand, if you your WSGI process is run in so-called daemon mode (to verify, check if `WSGIDaemonProcess` is used in your apache configuration), you can reload the WSGI process by touching the `.wsgi` file.

By default, reloading is enabled and a change to the WSGI file will trigger the reloading. You can use the `WSGIScriptReloading On|Off` directive to control the reloading.

2.3 Web Browser's Developer Console

For developing webapp, it is IMPORTANT to turn on the developer's web console to observe the request/response message.

For Firefox: Choose "Settings" ⇒ Developer ⇒ Web Console. (Firebug, which I relied on for the past years, is no longer maintained. You should use Web Console instead.)

For Chrome: Choose "Settings" ⇒ More Tools ⇒ Developer Tools.

For IE: ??.

My experience is that you CANNOT write a JavaScript without the developer console, as no visible message will be shown when an error is encountered. You have to find the error messages in the developer console!!!

3. Routes and View Functions

Read "Flask - Quick Start" @ http://flask.pocoo.org/docs/0.12/quickstart/.

Flask uses Werkzeug package to handle URL routing.

Flask comes with a built-in developmental web server, i.e., you do not need to run an external web server (such as Apache) during development.

3.1 Getting Started

Write a Hello-world Flask Webapp

Let us begin by creating our first hello-world webapp in Flask. Create a Python-Flask module called `hello_flask` (save as `hello_flask.py`), under your chosen project directory (or PyDev's project), as follows:

```
# -*- coding: UTF-8 -*-
"""
hello_flask: First Python-Flask webapp to say hello
"""
from flask import Flask  # From 'flask' module import 'Flask' class
app = Flask(__name__)    # Construct an instance of Flask class for our webapp

@app.route('/')   # URL '/' to be handled by main() route handler (or view function)
def main():
    """Say hello"""
    return 'Hello, world!'

if __name__ == '__main__':  # Script executed directly (instead of via import)?
    app.run()  # Launch built-in web server and run this Flask webapp
```

Run the webapp

To run the flask app under PyDev: Right-click on `hello_flask.py` ⇒ Run As ⇒ Python Run.

To run the flask app under Command-Line:

```
$ cd /path/to/project-directory
$ python hello_flask.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Flask launches its built-in developmental web server and starts the webapp. You can access the webapp from a browser via URL `http://127.0.0.1:5000` (or `http://localhost:5000`). The webapp shall display the hello-world message.

Check the console for the HTTP requests and responses:

```
127.0.0.1 - - [15/Nov/2015 12:46:42] "GET / HTTP/1.1" 200 -
```

You can stop the server by pressing `Ctrl-C`.

Run the webapp (Since Flask 0.11)

Starting from Flask 0.11, there is a new way of running flask app via the `'flask'` command or Python's `-m` switch thru an environment variable `FLASK_APP`, as follows:

```
$ export FLASK_APP=hello_flask.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/
```

Using environment is more flexible in changing the configuration (e.g., enable debug mode), without modifying the codes.

How It Works

1. Line 1 sets the source code encoding to UTF-8, recommended for internationalization (i18n).
2. Lines 2-4 are the doc-string for the Python module.
3. In Line 5, we import the `Flask` class from the `flask` module. The `from-import` statement allows us to reference the `Flask` class directly (without qualifying with the module name as `flask.Flask`). In Line 6, we create a new instance of `Flask` class called `app` for our webapp. We pass the Python's global variable `__name__` into the `Flask`'s constructor, which would be used to determine the root path of the application so as to locate the relevant resources such as templates (HTML) and static contents (images, CSS, JavaScript).
4. Flask handles HTTP requests via the so-called *routes*. The decorator `@app.route(url)` registers the decorated function as the route handler (or view function) for the `url`.

5. In Line 8-11, Flask registers the function `main()` as the route handler for the root url `'/'`. The return value of this function forms the response message for the HTTP request. In this case, the response message is just a plain text, but it would be in HTML/XML or JSON.
6. In Flask, the route handling function is called a *view function* or *view*. It returns a view for that route (quite different from the presentation view in MVC)?! I rather call it *route handler*.
7. The `if __name__ == '__main__'` ensures that the script is executed directly by the Python Interpreter (instead of being loaded as an imported module, where `__name__` will take on the module name).
8. The `app.run()` launches the Flask's built-in development web server. It then waits for requests from clients, and handles the requests via the routes and route handlers.
9. You can press Ctrl-C to shutdown the web server.
10. The built-in developmental web server provided by `Flask` is not meant for use in production. I will show you how to run a Flask app under Apache web server later.

3.2 Multiple Routes for the same Route Handler

You can register more than one URLs to the same route handler (or view function). For example, modify `hello_flask.py` as follows:

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

Restart the webapp. Try issuing URL `http://127.0.0.1:5000/` and `http://127.0.0.1:5000/hello`, and observe the hello-world message.

To inspect the url routes (kept under property `app.url_map`), set a breakpoint inside the `main()` and debug the program. Issue the following command in the console prompt:

```
>>> app.url_map
Map([<Rule '/hello' (GET, HEAD, OPTIONS) -> main>,
 <Rule '/' (GET, HEAD, OPTIONS) -> main>,
 <Rule '/static/<filename>' (GET, HEAD, OPTIONS) -> static>])
```

The urls `'/hello'` and `'/'` (for HTTP's `GET`, `HEAD` and `OPTIONS` requests) are all mapped to function `main()`. We will discuss `'/static'` later.

3.3 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, we can turn on the debug mode, which enables the auto-reloader as well as the debugger.

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
```

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 (Since Flask 0.11)

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
```

3.4 Serving HTML pages

Instead of plain-text, you can response with an HTML page by returning an HTML-formatted string. For example, modify the `hello_flask.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
......
```

How It Works

1. The second optional return value is the HTTP response status code, with default value of 200 (for OK).

3.5 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), we separate the View (or Presentation) from the Controller by creating an HTML page called "`hello.html`" under a sub-directory called "`templates`", as follows:

`templates/hello.html`

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

Modify the `hello_flask.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('hello.html')  # HTML file to be placed under sub-directory templates
......
```

How It Works

1. Flask uses Jinja2 template engine for rendering templates.
2. In the above example, the function `render_template()` (from module `flask`) is used to render a Jinja2 template to an HTML page for display on web browser. In this example, the template is simply an HTML file (which does not require rendering). A Jinja2 template may contain expression, statement, and other features. We will elaborate on the powerful Jinja2 templates later.

3.6 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:

```
# -*- coding: UTF-8 -*-
"""
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
```

How It Works

1. Try issuing URL `http://localhost:5000/hello` to trigger the first route.
2. The URL of the second route contains a variable in the form of `<url-varname>`, which is also bound to a function parameter. Try issuing URL `http://localhost:5000/hello/peter`.
3. You can also apply a type converter to the variable (as in the 3rd route), in the form of `<type-converter:url-varname>` to filter the type of the URL variable. The available type-converters are:
    - `string`: (default) accept any text without a slash (/). You can apply addition arguments such as `length`, `minlength`, `maxlength`, e.g., `'<string(length=8):username>'`, `'<string(minlength=4, maxlength=8):username>'`.
    - `int`: you can apply additional argument such as `min` and `max`, e.g., `'<int(min=1, max=99):count>'`
    - `float`
    - `path`: accept slash (/) for URL path, e.g., `'<path:help_path>'`
    - `uuid`
    - `any(converter1,...)`: matches one of the converter provided, e.g., `any(int, float)`

3.7 Functions redirect() and url_for()

In your route handlers, you can use "`return redirect(`*`url`*`)`" to redirect the response to another *url*.

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 `'/'`, and `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='peter', _external=True)`: returns the external URL `'http://localhost:5000/hello/peter'`.

For example,

```
# -*- coding: UTF-8 -*-
"""
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='Peter'))
        # 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/Peter`.
2. The console message clearly indicate the redirection:

   ```
   127.0.0.1 - - [15/Mar/2017 14:21:48] "GET / HTTP/1.1" 302 -
   127.0.0.1 - - [15/Mar/2017 14:21:48] "GET /hello/Peter HTTP/1.1" 200 -
   ```

   The original request to `'/'` has a response status code of "302 Found" and was redirected to `'/hello/peter'`.
3. You should also turn on the "Developer's Web console" to observe the request/response.

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 change your URL.

3.8 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
```

A url-map for `'static'` is created as follows:

```
>>> app.url_map
Map([.....,
 <Rule '/static/<filename>' (GET, HEAD, OPTIONS) -> static>])
```

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) - to be discussed later.

In production, the `static` directory should be served directly by an HTTP server (such as Apache) for performance.

4. 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.

4.1 Getting Started

Example: 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:

```
# -*- coding: UTF-8 -*-
"""
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)
```

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.

How It Works

1. The `j2_query.html` is an ordinary html file. But the `j2_response.html` is a Jinja2 template file. It contains static texts as well as placeholders for dynamic contents. The placeholders are identified by `{{ jinja2-varname }}`.
2. To render a template, use `render_template(template_filename, **kwargs)` function (of the `flask` package). This function takes a template filename as the first argument; and optional keywords arguments for passing values from Flask's view function into Jinja2 templates. In the above example, we pass Flask's variable `_username` from view function into Jinja2 as template variable `username`.
3. This example also illustrates the handling of form data. We processes the POST request parameters, retrieved via `request.form.get(param-name)`. (The `get(attr-name)` returns `None` if `attr-name` is not present; while `request.form[param-name]` throws a `KeyError` exception.)
4. There are several ways to retrieve the HTTP request parameters in Flask:
    - `request.args`: a key-value dictionary containing the GET parameters in the URL. For example, in `http://localhost/?page=10`, the `request.args['page']` returns 10.
    - `request.form`: a key-value dictionary containing the POST parameters sent in the request body as in the above example.
    - `request.values`: combination of `request.args` and `request.form`.
    - `request.get_json`: for loading the JSON request data.

4.2 Jinja2 Template Syntaxes

Reference: Template Designer Documentation @ http://jinja.pocoo.org/docs/dev/templates/.

A Jinja2 template is simply a text file embedded with Jinja2 statements, marked by:

- `{# ...comment... #}`: a comment not included in the template output.
- `{{ ...expression... }}`: an expression (such as variable or function call) to be evaluated to produce the template output.
- `{% ...statement... %}`: a Jinja2 statement.

You can use Jinja2 template to create any formatted text, including HTML, XML, Email, or MarkDown.

Jinja2 template engine is powerful. It supports variables, control constructs (conditional and loop), and inheritance. Jinja2 syntaxes closely follow Python.

Variables

Template variables are bound by the so-called *context dictionary* passed to the template. They can be having a simple type (such as string or number), or a complex type (such as list or object). You can use an expression `{{ jinja2-varname }}` to evaluate and output its value.

The following flask's variables/functions are available inside the Jinja2 templates as:

- `config`: the `flask.config` object.
- `request`: the `flask.request` object.
- `g`: the `flask.g` object, for passing information within the current active request only.
- `session`: the `flask.session` object, for passing information from one request to the next request for a particular user.
- `url_for()`: the `flask.url_for()` function.
- `get_flashed_message()`: the `flask.get_flashed_message()` function.

To inject new variables/functions into all templates from flask script, use `@app_context_processor` decorator. See ...

Testing Jinja2 Syntaxes

You can test Jinja2 syntaxes without writing a full webapp by constructing a `Template` instance and invoking the `render()` function. For example,

```
# Import Template class
>>> from jinja2 import Template

# Create a Template containing Jinja2 variable
>>> t = Template('Hello, {{ name }}')

# Render the template with value for variable
>>> t.render(name='Peter')
'Hello, Peter'
```

Jinja2's Filters

You can modify a value by piping it through so called *filter*, using the pipe symbol (`'|'`) in the form of `{{ varname|filter-name }}`, e.g.,

```
>>> from jinja2 import Template
>>> t1 = Template('Hello, {{ name|striptags }}')
>>> t1.render(name='<em>Peter</em>')
'Hello, Peter'  # HTML tags stripped

>>> t2 = Template('Hello, {{ name|trim|title }}')
>>> t2.render(name='  peter and paul  ')
'Hello, Peter And Paul'  # Trim leading/trailing spaces and initial-cap each word
```

The piping operation `{{ name|trim|title }}` is the same as function call `title(trim(name))`. Filters can also accept arguments, e.g. `{{ mylist|join(', ') }}`, which is the same as `str.join(', ', mylist)`.

The commonly-used built-in filters are:

- `upper`: to uppercase
- `lower`: to lowercase
- `capitalize`: first character in uppercase, the rest in lowercase
- `title`: initial-capitalize each word
- `trim`: strip leading and trailing white-spaces
- `striptags`: remove all HTML tags
- `default('default-value')`: provides the default value if the value is not defined, e.g.

```
>>> from jinja2 import Template
>>> t = Template('{{ var|default("some default value") }}')

>>> t.render()
'some default value'
>>> t.render(var='some value')
'some value'
```

- `safe`: output without escaping HTML special characters. See the section below.
- `tojson`: convert the given object to JSON representation. You need to coupled with `safe` to disable autoescape, e.g., `{{ username|tojson|safe }}`.

Auto-Escape and safe filter

To protect against Code Injection and XSS (Cross-Site Scripting) attack, it is important to replace special characters that are used to create markups, especially `'<'` and `'>'`, to their escape sequences (known as HTML entities), such as `&lt;` and `&gt;`, before sending these characters in the response message to be rendered by the browser.

In Jinja2, if you use `render_template()` to render a `.html` (or `.htm`, `.xml`, `.xhtml`), auto-escape is enabled by default for all the template variables, unless the filter `safe` is applied to disable the escape.

For example, run the above `hello_jinja2` example again. Try entering `'<em>Peter</em>'` into the box. Choose "view source" to view the rendered output. Clearly, the special characters are replaced by their HTML entities (i.e., `'&lt;em&gt;Peter&lt;/em&gt;'`).

Then, modify `templates/response.html` to apply the `safe` filter to `username` (i.e., `{{ username|safe }}`), and repeat the above.

You can also explicitly enable/disable autoescape in your Jinja2 template using `{% autoescape %}` statement, e.g.,

```
{% autoescape false %}
  <p>{{ var_no_escape }}</p>
{% endautoescape %}
```

Assignment and set Statement

You can create a variable by assigning a value to a variable via the `set` statement, e.g.,

```
{% set username = 'Peter' %}
{% set navigation = [('index.html', 'Index'), ('about.html', 'About')] %}
{% set key, value = myfun() %}
```

Conditionals

The syntax is:

```
{% if ...... %}
   ......
{% elif ...... %}
   ......
{% else %}
   ......
{% endif %}
```

For example,

```
# -*- coding: UTF-8 -*-
from jinja2 import Template
t = Template('''
{% if name %}
  Hello, {{ name }}
{% else %}
  Hello, everybody
{% endif %}''')

print(t.render(name='Peter'))  # 'Hello, Peter'
print(t.render())              # 'Hello, everybody'
```

Loops

Jinja2 support `for-in` loop. The syntax is:

```
{% for item in list %}
   ......
{% endfor %}
```

For example, to generate an HTML list:

```
# -*- coding: UTF-8 -*-
from jinja2 import Template
t = Template('''
<ul>
{% for message in messages %}
<li>{{ message }}</li>
{% endfor %}
</ul>''')

print(t.render(messages=['apple', 'orange', 'pear']))
```

The output is:

```
<ul>
<li>apple</li><li>orange</li><li>pear</li>
</ul>
```

`break` and `continue`

Jinja2 does not support `break` and `continue` statements for loops.

`for-in` loop with `else`

You can add a `else`-clause to the `for` loop. The `else`-block will be executed if `for` is empty.

```
{% for item in list %}
   ......
{% else %}
   ......      # Run only if the for-loop is empty
{% endfor %}
```

The Variable `loop`

Inside the loop, you can access a special variable called `loop`, e.g.,

- `loop.first`: first iteration?

- `loop.last`: last iteration?
- `loop.cycle('str1', 'str2', ...)`: print `str1`, `str2`,... alternating
- `loop.index`: 1-based index
- `loop.index0`: 0-based index
- `loop.revindex`: reverse 1-based index
- `loop.revindex0`: reverse 0-based index

For example:

```
# -*- coding: UTF-8 -*-
from jinja2 import Template
t = Template('''
<ul>
{% for item in items %}<li>
  {% if loop.first %}THIS IS ITEM 1{% endif %}
  {{ loop.index }}: {{ item }} ({{ loop.cycle('odd', 'even') }})</li>
{% endfor %}
</ul>''')

print(t.render(items=['apple', 'banana', 'cherry']))
```

The output is:

```
<ul>
<li>
  THIS IS ITEM 1
  1: apple (odd)</li>
<li>

  2: banana (even)</li>
<li>

  3: cherry (odd)</li>

</ul>
```

`with` block-scope variable

You can create block-scope variable using the `with` statement, e.g.,

```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class='flashes'>
    {% for message in messages %}<li>{{ message }}</li>{% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

The variable `message` is only visible inside the `with`-block.

Take note that `with` belongs to Jinja2 extension, and may not be available by default.

`macro` and `import`

A Jinja2 macro is similar to a Python function. The syntax is:

```
{% macro macro-name(args) %}
   ......
{% endmacro %}
```

For example,

```
{% macro gen_list_item(item) %}
   <li>{{ item }}</li>
{% endmacro %}

<ul>
  {% for message in messages %}
    {{ gen_list_item(message) }}
  {% endfor %}
</ul>
```

You can also store all the macros in a standalone file, and import them into a template. For example,

```
{% import 'macro.html' as macros %}

<ul>
  {% for message in messages %}
    {{ macros.gen_list_item(message) }}
  {% endfor %}
</ul>
```

The `import` statement can take several forms (similar to Python's `import`)

```
{% import 'macro.html' as macros %}
{% from 'macro.html' import gen_list_item as gen_list %}
```

4.3 Building Templates with include and Inheritance

Three Jinja2 statements, namely, `include`, `block` and `extends`, can be used to build templates from different files. The `block` and `extends` statements are used together in inheritance.

Include

In a template file, you can include another template file, via the `include` statement. For example,

```
{% include 'header.html' %}
```

Blocks

You can define a block via the `block` statement, as follows:

```
{% block block-name %}
.....
{% endblock %}
```

Blocks can be used as placeholders for substitution, as illustrated in template inheritance below.

Template Inheritance via extends and block Statements

You can derive a template from a base template through template inheritance (similar to Python class inheritance), via the `extends` and `block` statements.

As an example, we shall first create a base template (called `j2_base.html`) as follows:

`templates/j2_base.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  {% block head_section %}
  <link rel="stylesheet" href="/static/css/my_style.css" />
  <title>{% block title %}{% endblock %} - ABC Corp.</title>
  {% endblock %}
</head>
<body>
  <header id="header">
```

```
    {% block header %}{% endblock %}
  </header>

  <div id="content">
    {% block content %}{% endblock %}
  </div>

  <footer id="footer">
    {% block footer %} © Copyright 2015 by ABC Corp.{% endblock %}
  </footer>
</body>
</html>
```

The base template contains 5 blocks, defined via `{% block block-name %}...{% endblock %}`. We can derive child templates from the base template by filling in these blocks. For example, let's create a `j2_derived.html` that extends from `j2_base.html` as follows:

`templates/j2_derived.html`

```
{% extends "j2_base.html" %}
{% block title %}Main Page{% endblock %}
{% block head_section %}
  {{ super() }}{# Render the contents of the parent block #}
  <style type="text/css">
    .red { color: red; }
  </style>
{% endblock %}
{% block header %}<h1>Hello</h1>{% endblock %}
{% block content %}
  <p class="red">hello, world!</p>
{% endblock %}
```

1. The `{% extends .... %}` specifies the parent template.
2. Blocks are defined, that replaces the blocks in the parent template.
3. The `{{ super() }}` renders the contents of the parent block.

To test the template, write a script, as follows:

```
"""
j2_template_inheritance: Test base and derived templates
"""
from flask import Flask, render_template
app = Flask(__name__)
```

```
@app.route('/')
def main():
    return render_template('j2_derived.html')

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

Jinja2's `url_for()` Function

The `url_for(`*`view_function_endpoint`*`)` helper function, which returns the URL for the given view function (route handler), works in Jinja2 template as well.

In Flask, the static resources, such as images, css, and JavaScripts are stored in a sub-directory called `static`, by default, with sub-sub-directories such as `img`, `css`, `js`. The routes for these static files are `/static/<filename>` (as shown in the `app.url_map`).

- `url_for('static', filename='css/mystyles.css', _external=True)`: return the external URL `http://localhost:5000/static/css/mystyles.css`.

For example, to include an addition CSS in the `head_section` block:

```
{% block head_section %}
{{ super() }}
<link rel="stylesheet" href="{{ url_for('static', filename='css/mystyles.css') }}">
{% endblock %}
```

5. Web Forms via Flask-WTF and WTForms Extensions

References

1. Flask-WTF @ https://flask-wtf.readthedocs.org/en/latest/.
2. WTForms Documentation @ https://wtforms.readthedocs.org/en/latest/.

After Notes: Flask-WTF is somehow not quite compatible with the AngularJS client-side framework, which I used for building SPA (Single Page Architecture). The server-side input validation supported by Flask-WTF can also be carried out via Marshmallow, which is needed for serialization/deserialization of data models.

Recall that we can use the `request.form`, `request.args`, `request.values` and `request.get_json()` to retrieve the request data. However, there are many repetitive and tedious task in handling form input data, in particular, input validation. The Flask-WTF extension greatly simplifies form processing, such as generating HTML form codes, validating submitted data, cross-site request forgery (CSRF) protection, file upload, and etc.

Read "Flask-WTF Quick Start" @ http://flask-wtf.readthedocs.io/en/stable/quickstart.html.

Installing Flask-WTF (under virtual environment)

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

(venv)$ pip install flask-wtf
Successfully installed WTForms-2.1 flask-wtf-0.14.2
(venv)$ pip show flask-wtf
Name: Flask-WTF
Version: 0.14.2
Location: .../venv/lib/python3.5/site-packages
Requires: Flask, WTForms
(venv)$ pip show WTForms
Name: WTForms
Version: 2.1
Location: .../venv/lib/python3.5/site-packages
Requires:
```

5.1 WTForms

Flask-WTF uses WTForms for form input handling and validation. Installing Flask-WTF will automatically install WTForms, as shown above.

In WTForms,

- `Form` (corresponding to an HTML `<form>`) is the core container that glue everything together. A `Form` is a collection of `Field`s (such as `StringField`, `PasswordField` corresponding to `<input type="text">`, `<input type="password">`). You create a form by sub-classing the `wtforms.Form` class.
- A `Field` has a data type, such as `IntegerField`, `StringField`. It contains the field data, and also has properties, such as `label`, `id`, `description`, and `default`.
- Each field has a `Widget` instance, for rendering an HTML representation of the field.
- A field has a list of `validators` for validating input data.

You can explore these object in the console, e.g.,

```
>>> from wtforms import Form, StringField, PasswordField
>>> from wtforms.validators import InputRequired, Length

# Derive a subclass of 'Form' called 'LoginForm', containing Fields
>>> class LoginForm(Form):
       username = StringField('User Name:', validators=[InputRequired(), Length(min=3, max=30)])
       passwd = PasswordField('Password:', validators=[Length(min=4, max=16)])
# Construct an instance of 'LoginForm' called 'form'
>>> form = LoginForm()
```

```
# You can reference a field via dictionary-style or attribute-style
>>> form.username
<wtforms.fields.core.StringField object at 0x7f51dfcbbf50>
>>> form['username']
<wtforms.fields.core.StringField object at 0x7f51dfcbbf50>

# Show field attributes 'label', 'id' and 'name'
>>> form.username.label
Label('username', 'User Name:')
>>> form.username.id
'username'
>>> form.username.name
'username'
>>> form.passwd.validators
[<wtforms.validators.Length object at 0x7fdf0d99ee90>]

# Show current form data
>>> form.data
{'username': None, 'passwd': None}

# Run validator
>>> form.validate()
False
# Show validation errors
>>> form.errors
{'username': ['This field is required.'],
 'passwd': ['Field must be between 4 and 16 characters long.']}
    # errors in the form of {'field1': [err1a, err1b, ...], 'field2': [err2a, err2b, ...]}

# To render a field, coerce it to a HTML string:
>>> str(form.username)
'<input id="username" name="username" type="text" value="">'
# Or, you can also "call" the field
>>> form.username()
'<input id="username" name="username" type="text" value="">'
# Render with additional attributes via keyword arguments
>>> form.username(class_='red', style='width:300px')
'<input class="red" id="username" name="username" style="width:300px" type="text" value="">'
>>> form.username(**{'class':'green', 'style':'width:400px'})
'<input class="green" id="username" name="username" style="width:400px" type="text" value="">'
```

The `Field` has the following constructor:

```
__init__(label='', validators=None, filters=(), description='', id=None, default=None,
   widget=None, _form=None, _name=None, _prefix='', _translations=None)
```

- `validators`: a sequence of validators called by `validate()`. The `form.validate_on_submit()` checks on all these validators.
- `filters`: a sequence of filter run on input data.
- `description`: A description for the field, for tooltip help text.
- `widget`: If provided, overrides the widget used to render the field.

The commonly-used `Field`s are:

- `TextField`: Base `Field` for the others, rendered as `<input type='text'>`.
- `StringField`, `IntegerField`, `FloatField`, `DecimalField`, `DateField`, `DateTimeField`: subclass of `TextField`, which display and coerces data into the respective type.
- `BooleanField`: rendered as `<input type='checkbox'>`.
- `RadioField(..., choices=[(value, label)])`: rendered as radio buttons.
- `SelectField(..., choices=[(value, label)])`:
- `SelectMultipleField(..., choices=[(value, label)])`
- `FileField`
- `SubmitField`
- `HiddenField`: rendered as `<input type='hidden'>`.
- `PasswordField`: rendered as `<input type='password'>`.
- `TextAreaField`

The commonly-used `validators` are as follows. Most validators have a keyword argument `message` for a custom error message.

- `InputRequired()`
- `Optional()`: allow empty input, and stop the validation chain if input is empty.
- `Length(min=-1, max=-1)`
- `EqualTo(`*`fieldname`*`)`
- `NumberRange(min=None, max=None)`
- `AnyOf(`*`sequence-of-validators`*`)`
- `NoneOf(`*`sequence-of-validators`*`)`
- `Regexp(`*`regex`*`, flags=0)`: flags are regex flags such as `re.IGNORECASE`.

- Email()
- URL()

5.2 Using Flask-WTF

Flask integrates WTForms using extension Flask-WTF.

Besides integrating WTForms, Flask-WTF also:

- Provides CSRF protection to ensure secure form
- Supports file upload
- Supports recaptcha
- Supports internationalization (i18n)

By default, Flask-WTF protects all forms against Cross-Site Request Forgery (CSRF) attacks. You need to set up an encryption key to generate an encrypted token in `app.config` dictionary, as follows:

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'a random string'  # Used for CSRF
```

You can generate a random hex string via:

```
import os; os.urandom(24).encode('hex') # Python 2.7

import os, binascii; binascii.hexlify(os.urandom(24)) # Python 3 and 2
```

Flask-WTF Example 1: Creating a Login Form

Under your project directory, create the following source files:

`wtfeg1_form.py`

```
"""
wtfeg1_form: Flask-WTF Example 1 - Login Form
"""
# Import 'FlaskForm' from 'flask_wtf', NOT 'wtforms'
from flask_wtf import FlaskForm
# Fields and validators from 'wtforms'
from wtforms import StringField, PasswordField
from wtforms.validators import InputRequired, Length

# Define the 'LoginForm' class by sub-classing 'Form'
```

```
class LoginForm(FlaskForm):
    # This form contains two fields with input validators
    username = StringField('User Name:', validators=[InputRequired(), Length(max=20)])
    passwd = PasswordField('Password:', validators=[Length(min=4, max=16)])
```

1. You need to import `Form` from `flask_wtf`, instead of `wtforms` (unlike the previous section).
2. I separated the forms from the app controller below, for better software engineering.

`wtfeg1_controller.py`

```
"""
wtfeg1_controller: Flask-WTF Example 1 - app controller
"""
import os, binascii
from flask import Flask, render_template
from wtfeg1_form import LoginForm

# Initialize Flask app
app = Flask(__name__)
app.config['SECRET_KEY'] = binascii.hexlify(os.urandom(24))  # Flask-WTF: Needed for CSRF

@app.route('/')
def main():
    # Construct an instance of LoginForm
    _form = LoginForm()

    # Render an HTML page, with the login form instance created
    return render_template('wtfeg1_login.html', form=_form)

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

`templates/wtfeg1_login.html`

```
<form method="POST" action="/">
  {{ form.hidden_tag() }} {# Renders ALL hidden fields, including the CSRF #}
  {{ form.username.label }} {{ form.username(size=20) }}
  {{ form.passwd.label }} {{ form.passwd(class='passwd', size=16) }}
  <input type='submit' value='Go'>
</form>
```

The rendered HTML page is as follows:

```
<form method="POST" action="/">
  <input id="csrf_token" name="csrf_token" type="hidden" value="xxxxxx">
  <label for="username">User Name:</label> <input id="username" name="username" size="20" type="text" value="">
  <label for="passwd">Password:</label> <input class="passwd" id="passwd" name="passwd" size="16" type="password" value="">
  <input type="submit" value="Go">
</form>
```

1. As mention, Flask-WTF provides CSRF protection for all pages by default. CSRF protection is carried out via a secret random token in a hidden field called `csrf_token`, generated automatically by Flask-WTF - you can find this hidden field in the rendered output. Nonetheless, in your HTML `form`, you need to include `{{ form.hidden_tag() }}` or `{{ form.csrf_token }}` to generate this hidden field.
2. For AJAX CSRF protection, see Flask-WTF documentation.

Flask-WTF Example 2: Processing the Login Form

`wtfeg2_form.py`: same as `wtfeg1_form.py`

`wtfeg2_controller.py`

```
"""
wtfeg2_controller: Flask-WTF Example 2 - Processing the Login Form
"""
import os, binascii
from flask import Flask, render_template, flash, redirect, url_for
from wtfeg2_form import LoginForm

# Initialize Flask app
app = Flask(__name__)
app.config['SECRET_KEY'] = binascii.hexlify(os.urandom(24))  # Flask-WTF: Needed for CSRF

@app.route('/', methods=['get', 'post'])  # First request via GET, subsequent requests via POST
def main():
    form = LoginForm()  # Construct an instance of LoginForm

    if form.validate_on_submit():  # POST request with valid input?
        # Verify username and passwd
        if (form.username.data == 'Peter' and form.passwd.data == 'xxxx'):
            return redirect(url_for('startapp'))
        else:
            # Using Flask's flash to output an error message
            flash('Wrong username or password')

    # For the initial GET request, and subsequent invalid POST request,
    # render an login page, with the login form instance created
    return render_template('wtfeg2_login.html', form=form)

@app.route('/startapp')
def startapp():
    return 'The app starts here!'

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

1. The initial access to `'/'` triggers a GET request (default HTTP protocol for a URL access), with `form.validate_on_submit()` returns `False`. The `form.validate_on_submit()` returns True only if the request is POST, and all input data validated by the validators.
2. Subsequently, user submits the login form via POST request, as stipulated in `<form method='POST'>` in the HTML `form`.
3. If username/password is incorrect, we output a message via `flash()` using Flask's flashing message facility. This message is available to the next request only.

`templates/wtfeg2_base.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Application</title>
  <meta charset="utf-8">
</head>
<body>
{# Display flash messages, if any, for all pages #}
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class='flashes'>
    {% for message in messages %}<li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}

{# The body contents here #}
{% block body %}{% endblock %}
</body>
</html>
```

1. We use layer template (or template inheritance) in this example.
2. The base template displays the flash messages, retrieved via `get_flashed_messages()`, if any, for ALL the pages.
3. It defines a block called `body` for the body contents, to be extended by the sub-templates.

templates/wtfeg2_login.html

```
{% extends "wtfeg2_base.html" %}
{% block body %}
<h1>Login</h1>
<form method="POST"> {# Default action to the same page #}
  {{ form.hidden_tag() }} {# Renders any hidden fields, including the CSRF #}
  {% for field in form if field.widget.input_type != 'hidden' %}
    {# Display filed #}
    <div class="field">
      {{ field.label }} {{ field }}
    </div>
    {# Display filed's validation error messages, if any #}
    {% if field.errors %}
      {% for error in field.errors %}
        <div class="field_error">{{ error }}</div>
      {% endfor %}
    {% endif %}
  {% endfor %}
  <input type="submit" value="Go">
</form>
{% endblock %}
```

1. The inherited template first displays the hidden fields, including `csrf_token`.
2. It then display ALL the field (except hidden fields) together with the field-errors, via a for-loop (unfortunately, you typically need to customize the appearance of each of the fields with BootStrap instead of a common appearance).

Start the Flask webapp. Issue URL `http://localhost:5000` and try:

1. Filling in invalid input. Observe the validation error messages.
2. Filling in valid input, but incorrect username/password. Observe the flash message.
3. Filling in correct username/password. Observe the redirection.

Message Flashing

Reference: Message Flashing @ http://flask.pocoo.org/docs/0.10/patterns/flashing/.

Good UIUX should provide enough feedback to the users. Flask provides a simple way to give feedback with the flashing system, which records a message at the end of a request and make it available for the next request and only next request.

You can include a category when flashing a message, e.g., error or warning (default is called message), and display or filter messages based on this category. For example,

```
flash('This is an error', 'danger')  # BootStrap uses 'danger', 'warning', 'info', 'success', and 'primary'
```

You can use the category in your Jinja2 template for CSS:

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="text-{{ category }}">{{ message }}</li> <!-- BootStrap classes -->
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

You can also filter the flashed message by category, e.g.,

```
{% with messages = get_flashed_messages(category_filter=['danger']) %}
  ......
{% endwith %}
```

6. More on Flask

6.1 The flask's API

A typical "`from flask import`" statement imports the following attributes from the `flask` package:

```
from flask import Flask, request, session, g, redirect, url_for, abort, render_template, flash
```

Check out the Flask's API @ http://flask.pocoo.org/docs/0.12/api/ and try to inspect these objects in a debugging session.

The Flask class

We create an instance of the `Flask` class for our flask webapp.

The request and response objects

The `request` object encapsulates the HTTP request message (header and data). We often need to extract the request data:

- `request.form`: parsed form data (name-value pairs) from POST or PUT requests, with a request header `{'Content-Type': 'application/x-www-form-urlencoded; charset=UTF-8'}`.
- `request.args`: parsed query string (name-value pairs) from GET request.

- request.values: combination of request.form and request.args.
- request.data: raw request data string.

The following properties are used in AJAX/JSON requests, which is popular nowadays in implementing Single Page Architecture (SPA):

- request.is_xhr: True for AJAX request (Check for presence of a request header {'X-Requested-With': 'XMLHttpRequest'}).
- request.is_json: True for JSON request data. (Check for presence of a request header {'Content-Type': 'application/json; charset=utf-8'}).
- request.get_json(force=False, silent=False, cache=True): Parse the incoming JSON data and return the Python object. By default, it returns None if MIME type is not application/json. You can set force=True to ignore the MIME type. If silent=True, this method will fail silently and return None.

In addition:

- request.method: gives the request methods such as 'GET', 'POST', 'PUT', 'PATCH', 'DELETE' (in uppercase).
- request.url:

The response object encapsulates the HTTP response message (header and data). The flask program typically does not manipulate the response object (which is manipulating at the client-side's JavaScript to extract the response's data and status code). Instead, a view function shall return a response with an optional status code, e.g.,

```
return plain_text
return plain_text, status_code
return html_text

from flask import render_template
return render_template(template_filename)

from flask import jsonify
return jsonify(data), status_code  # Convert Python data to a JSON string
```

You can use make_response() to add additional headers, e.g.,

```
from flask import make_response
response = make_response(render_template(index.html))
response.headers['myheader'] = 'myheader-value'  # Add an additional response header
```

The response object contains:

- response.data: response data.
- response.status_code: The numeric status code, e.g., 200, 404.
- response.status: The status string, e.g, "200 OK", "404 Not Found".

The session and g objects

The `session` object is meant for storing information for a user session, used for passing information from one request to the next request, in a multi-thread environment.

The `g` object is for storing information for a user for one request only, used for passing information from one function to another function within the request, in a multi-thread environment. It is also known as request-global or application-global.

Take note that `session` is used for passing information between requests; while `g` is used to pass information with one request (between functions).

The `escape()` Function and the `Markup` class

The `escape(str)` function converts the characters &, <, >, " and ' to their respective HTML escape sequences `&amp;`, `&lt;`, `&gt;`, `&quot;` and `&#39;` which is meant for safely display the text to prevent code injection. It returns a `Markup` object. For example,

```
>>> from flask import escape
>>> escape('Hello <em>World</em>!')
Markup('Hello &lt;em&gt;World&lt;/em&gt;!')
```

The `Markup` class can be used to create and manipulate markup text.

1. If a string (or object) is passed into `Markup`'s constructor, it is assumed to be safe and no escape will be carried out, e.g.,

```
>>> from flask import Markup
>>> h1 = Markup("Hello <em>World</em>!")
>>> h1
Markup('Hello <em>World</em>!')
>>> print(h1)
Hello <em>World</em>!
```

2. To treat the string (or object) as unsafe, use `Markup.escape()`:

```
>>> h2 = Markup.escape('Hello <em>World</em>!')
>>> h2
Markup('Hello &lt;em&gt;World&lt;/em&gt;!')
```

3. When a object with a `__html__()` representation is passed to `Markup`, it uses the representation, assuming that it is safe, e.g.,

```
>>> class Page:
        def __html__(self):
            return 'Hello <em>World</em>!'

>>> Markup(Page())  # Markup an instance of Page class
Markup('Hello <em>World</em>!')
```

Flashing Message

The `flash(message, category='message')` function flashes a message of the category to the *next* request (via the `session`).

The Jinja2 template uses `get_flashed_messages()` to remove and display the flash messages. For example, I use the following Jinja2 macro to display all the flash messages (in conjunction with BootStrap and Font Awesome). I will describe Jinja2 in the next section.

```
{% macro list_flash_messages() %}
  {% with messages = get_flashed_messages(with_categories=True) %}
    {% if messages %}
      <ul class="fa-ul">
      {% for category, message in messages %}
      <li class="big text-{{category if category else 'danger'}}" ><i class="fa-li fa fa-times-circle"></i>{{ message }}</li>
      {% endfor %}
      </ul>
    {% endif %}
  {% endwith %}
{% endmacro %}
```

`jsonify()`

The `jsonify(*args, **kwargs)` function creates a JSON string to be used as the `response` object. For example,

```
"""
hello_jsonify: Test JSON response
"""
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/a')
def a():
    return jsonify({'message': 'unknown user'}), 400
    # response.data is:
    # {
    #   "message": "unknown user"
    # }

@app.route('/b')
def b():
    return jsonify(username='peter', id=123), 200
    # response.data is:
    # {
    #   "id": 123,
    #   "username": "peter"
    # }
```

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

Miscellaneous

- `redirect(url)`: redirect the HTTP request to the given url.
- `url_for(view-function)` functions: returns the url for the given view function, as as to avoid hard-coding url.
- `abort(response-status-code)` function: abort the request and return the given status code (such as "404 Not Found", "400 Bad Request")
- `render_template(Jinja2-template-filename, **kwargs)` function: Render the given Jinja2 template into HTML page. The optional `kwargs` will be passed from the view function into the Jinja2 template.
- `Config` class: for managing the Flask's configuration. You can use methods `from_object()`, `from_envvar()` or `from_pyfile()` to load the configurations. This will be describe in the configuration section.

6.2 Custom Error Pages

You can provide consistent custom error pages via `@app.errorhandler(response_status_code)`. For example,

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404  # Not Found

@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500  # Internal Server Error
```

1. The error handlers return a response page, and a numeric error code.

6.3 The flask's session object

If you need to keep some data from one request to another request for a particular user, but no need to persist them in the database, such as authentication, user information, or shopping cart, use the `session` object.

`flask`'s `session` object is implemented on top of cookies, and encrypts the session values before they are used in cookies. To use `session`, you need to set a secret key in `app.config['SECRET_KEY']`. (This value is also used in Flask-WTF for CSRF protection).

Example: Using flask's session object

```
# -*- coding: UTF-8 -*-
"""
test_session: Testing Flask's session
"""
from flask import Flask, session, redirect, url_for, escape, request

app = Flask(__name__)
app.config['SECRET_KEY'] = 'Your Secret Key'

@app.route('/')
def main():
    if 'username' in session:
        return 'You are already logged in as %s' % escape(session['username'])
            # Escape special HTML characters (such as <, >) in echoing to prevent XSS
    return (redirect(url_for('login')))

@app.route('/login', methods=['GET', 'POST'])
def login():
    # For subsequent POST requests
    if request.method == 'POST':
        username = request.form['username']
        session['username'] = username   # Save in session
        return 'Logined in as %s' % escape(username)

    # For the initial GET request
    return '''
        <form method='POST'>
          Enter your name: <input type='text' name='username'>
          <input type='submit' value='Login'>
        </form>'''

@app.route('/logout')
def logout():
    # Remove 'username' from the session if it exists
    session.pop('username', None)
    return redirect(url_for('main'))

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

1. If you are not using template engine, use `escape()` to escape special HTML characters (such as <, >) to prevent XSS. Templates ending in `.html`, etc, provide auto-escape when echoing variables in their rendered output.
2. You can generate a secret key via "`import os; os.urandom(24)`".

[TODO] Shopping Cart Example

6.4 flask's 'g' object

In the previous section, we use the `session` object to store data for a user (thread) that are passed from one request to the next requests, in a multi-thread environment. On the other hand, the `g` object allows us to store data that is valid for one request only, i.e., request-global. The `g` object is used to pass information from one function into another within a request. You cannot use an ordinary global variable, as it would break in a multi-thread environment.

For example, to check the time taken for a request:

```
@app.before_request
def before_request():
    g.start_time = time.time()  # Store in g, applicable for this request and this user only

@app.teardown_request
def teardown_request(exception=None):
    time_taken = time.time() - g.start_time   # Retrieve from g
    print(time_taken)
```

The decorators `@app.before_request` and `@app.teardown_request` mark the functions to be run before and after each user's request.

6.5 Flask's Application Context and Request Context

References:

1. "What is the purpose of Flask's context stacks?" @ http://stackoverflow.com/questions/20036520/what-is-the-purpose-of-flasks-context-stacks
2. "Understanding Contexts in Flask" @ http://kronosapiens.github.io/blog/2014/08/14/understanding-contexts-in-flask.html

In dictionary, *context* means environment, situation, circumstances, etc. In programming, a *context* is an entity (typically an object) that encapsulate the state of computation, which is like a bucket that you can store and pass information around; push onto a stack and pop it out. It typically contains a set of variables (i.e., name and object bindings). The same name can appear on different contexts and binds to different objects.

Flask supports multiple app instances, with multiple requests from multiple users, run on multiple threads. Hence, it is important to isolate the requests and the app instances (using global variables do not work). This is carried out via the context manager.

Flask supports 2 types of contexts:

1. An application context (app context) that encapsulates the state of an app instance, and contains the necessary name-object bindings for the app, such as `current_app`, `url_for`, `g` objects. For example, in order to execute `url_for('hello')` to lookup for the URL for the view function `hello()`, the system needs to look into current app context's URL map, which could be different for difference Flask instances.

2. A request context that encapsulate the state of a request (from a user), and contains the necessary name-object bindings for the request (such as the `request`, `session` and `g` objects; and `current_user` of the Flask-Login extension).

Flask also maintain 2 separate stacks for the app context and request context. Flask automatically creates the request context and app context (and pushes them onto the stacks), when a request is made to a view function. In other situations, you may need to manually create the contexts. You can push them onto the stacks via the `push()|pop()` methods; or via the with-statement context manager (which automatically pushes on entry and pops on exit).

Running code outside an app context via `app.app_context()`

When a request is made to trigger a view function, Flask automatically set up the request and application context. However, if you are running codes outside an app context, such as initializing database, you need to set up your own app context, e.g,

```
app = Flask(__name__)  # Create a Flask app instance
db = SQLAlchemy()      # Initialize the Flask-SQLAlchemy extension instance
db.init_app(app)       # Register with Flask app

# Setup models
with app.app_context():
    # Create an app context, which contains the necessary name-object bindings
    # The with-statement automatically pushes on entry
    db.create_all()   # run under the app context
    # The with-statement automatically pops on exit
```

Instead of the with-statement, you can explicitly push/pop the context:

```
ctx = app.app_context()
ctx.push()
......
......
ctx.pop()
```

In this case, we are concerned about the app context, instead of request context (as no HTTP request is made).

Running unit-tests via `app.test_request_context()`

To run unit-test (which issues HTTP requests), you need to manually create a new request context via `app.test_request_context()` (which contains the necessary name-object bindings), e.g.,

```
import unittest
from flask import request

class MyTest(unittest.TestCase):
    def test_something(self):
        with app.test_request_context('/?name=Peter'):
            # Create a request context, which is needed to pass the URL parameter to a request object
            # The with-statement automatically pushes on entry
            # You can now view attributes on request context stack by using 'request'.
            assert flask.request.path == '/'
            assert flask.request.args['name'] == 'Peter'
            # The with-statement automatically pops on exit

        # After the with-statement, the request context stack is empty
```

Inspecting the app and request context objects

[TODO]

Context Variables and @app.context_processor Decorator

You can create a set of variables which are available to all the views in their context, by using the `context_processor` decorator. The decorated function shall return a dictionary of key-value pairs. For example,

```
# -*- coding: UTF-8 -*-
"""
test_contextvar: Defining Context Variables for all Views
"""
from flask import Flask, render_template, session
app = Flask(__name__)

@app.context_processor
def template_context():
    '''Return a dictionary of key-value pairs,
       which will be available to all views in the context'''
    if 'username' in session:
        username = session['username']
    else:
        username = 'Peter'

    return {'version':88, 'username':username}

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

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

The `templates/test_contextvar.html` is as follows. Take note that the context variables `username` and `version` are available inside the view.

```
<!DOCTYPE html>
<html>
<head><title>Hello</title></head>
<body>
<h1>Hello, {{ username }}</h1>
<p>Version {{ version }}<p>
</body>
</html>
```

6.6 Logging

Reference: Logging Application Errors @ http://flask.pocoo.org/docs/0.10/errorhandling/.

See Python Logging for basics on Python `logging` module.

Proper Logging is ABSOLUTELY CRITICAL!!!

Flask uses the Python built-in logging system (Read Python's logging for details).

- Flask constructs a `Logger` instance and attach to the current app as `app.logger`.
- You can use `app.logger.addhandler()` to add a handler, such as `SMTPHandler`, `RotatingFileHandler`, or `SysLogHanlder` (for Unix syslog).
- Use `app.logger.debug()`,... `app.logger.critical()` to send log message.

Example: Flask's logging

```
# -*- coding: UTF-8 -*-
"""
test_logging: Testing logging under Flask
"""
from flask import Flask
app = Flask(__name__)

# Create a file handler
import logging
from logging.handlers import RotatingFileHandler

handler = RotatingFileHandler('test.log')
handler.setLevel(logging.INFO)
handler.setFormatter(logging.Formatter(
    "%(asctime)s|%(levelname)s|%(message)s|%(pathname)s:%(lineno)d"))
app.logger.addHandler(handler)
app.logger.setLevel(logging.DEBUG)

app.logger.debug('A debug message')
app.logger.info('An info message')
app.logger.warning('A warning message')
app.logger.error('An error message')
app.logger.critical('A critical message')

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

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

To include other libraries' (such as SQLAlchmey):

```
from logging import getLogger
loggers = [app.logger,
           getLogger('sqlalchemy'),
           getLogger('otherlibrary')]
for logger in loggers:
    logger.addHandler(mail_handler)
    logger.addHandler(file_handler)
```

6.7 Configuration and Flask's 'app.config' object

Reference: Flask Configuration @ http://flask.pocoo.org/docs/0.10/config/.

The app.config object

The Flask app maintains its configuration in a app.config dictionary. You can extend with your own configuration settings, e.g.,

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

app.config['DEBUG'] = True  # Key in uppercase
print(app.config['DEBUG'])
print(app.debug)            # proxy to app.config['DEBUG']
```

The keys in app.config must be in UPPERCASE, e.g., app.config['DEBUG'], app.config['SECRET_KEY'].

SOME of the `app.config` values are also forwarded to Flask's object `app` (as proxies) with their keys change from UPPERCASE to lowercase. For example, `app.config['DEBUG']` (in uppercase) is the same as `app.debug` (in lowercase), `app.config['SECRET_KEY']` is the same as `app.secret_key`.

Configuration Files

There are many options to maintain configuration files:

- Using Python modules/classes with `from_object()`, `from_pyfile()` and `from_envvar()`: Recommended. See below.
- Using `INI` file with `ConfigParser`: see Python ConfigParser.
- JSON
- YAML

Using Python Modules/Classes with `from_object()`, `from_pyfile()` and `from_envvar()`

You can use `app.config.from_object(modulename|classname)` to load attributes from a python module or class into `app.config`. For example,

```
"""
test_config: Configuration file
"""
DEBUG = True
SECRET_KEY = 'your-secret-key'
USERNAME = 'admin'
invalid_key = 'hello'   # lowercase key not loaded
```

```
from flask import Flask

app = Flask(__name__)
app.config.from_object('test_config')
    # Load ALL uppercase variables
    #   from Python module 'test_config.py' into 'app.config'

print(app.config['DEBUG'])
print(app.debug)            # Proxy to 'DEBUG'
print(app.config['SECRET_KEY'])
print(app.secret_key)       # Proxy to 'SECRET_KEY'
print(app.config['USERNAME'])
# print(app.username)       # No proxy for 'USERNAME'
# print(app.config['invalid_key'])  # lowercase key not loaded
```

You can further organize your configuration settings for different operating environments via subclasses. For example,

```
"""
test_config1: Configuration file
"""
class ConfigBase():
    SECRET_KEY = 'your-secret-key'
    DEBUG = False

class ConfigDevelopment(ConfigBase):
    """For development. Inherit from ConfigBase and override some values"""
    DEBUG = True

class ConfigProduction(ConfigBase):
    """For Production. Inherit from ConfigBase and override some values"""
    SECRET_KEY = 'production-secret-key'
```

You can choose to load a particular configuration class:

```
app.config.from_object('test_config1.ConfigDevelopment') # modulename.classname
```

To store sensitive data in a dedicated folder which is not under source control, you can use the *instance folders* with `from_pyfile(filename|modulename)`. For example,

```
app = Flask(__name__)
app.config.from_object('test_config1.ConfigDevelopment')
    # Default configuration settings
app.config.from_pyfile('local/config_local.py', silent=True)
    # Override the defaults or additional settings
    # silent=True: Ignore if no such file exists
```

You can also override the default settings, through an environment variable (which holds a configuration filename) with `from_env(envvar_name)`. For example,

```
# Load from an environment variable which holds the configuration filename.
# Set the environment variable 'EXTRA_SETTINGS' as follows before running the app:
# export EXTRA_SETTINGS=test_config2.py
app.config.from_envvar('EXTRA_SETTINGS', silent=True)
      # Override existing values
      # silent=True: Ignore if no such environment variable exists
```

```
"""
test_config2: More configuration loaded via environment setting
"""
USERNAME = 'peter'
```

7. 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.

7.1 Installing SQLAlchemy (under virtual environment)

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

(venv)$ pip install sqlalchemy

(venv)$ pip show sqlalchemy
Name: SQLAlchemy
Version: 1.1.5
Location: .../venv/lib/python3.5/site-packages
Requires:
```

7.2 Using SQLAlchemy with MySQL/PostgreSQL

Installing MySQL Driver for Python

See "Python-MySQL Database Programming".

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

# Python 3
(venv)$ pip install mysqlclient
Successfully installed mysqlclient-1.3.9
(venv)$ pip show mysqlclient
Name: mysqlclient
Version: 1.3.9
Summary: Python interface to MySQL
```

```
Location: .../venv/lib/python3.5/site-packages
Requires:

# Python 2
(venv)$ pip install MySQL-python
......
(venv)$ pip show MySQL-python
......
```

Installing PostgreSQL Driver for Python

See "Python-PostgreSQL Database Programming".

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

(venv)$ pip install psycopg2
Successfully installed psycopg2-2.6.2
(venv)$ pip show psycopg2
Name: psycopg2
Version: 2.6.2
Summary: psycopg2 - Python-PostgreSQL Database Adapter
Location: .../venv/lib/python3.5/site-packages
Requires:
```

Setting up MySQL

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

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

Setting up PostgreSQL

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

```
# Create a new PostgreSQL user called testuser, allow user to login, but NOT creating databases
$ sudo -u postgres createuser --login --pwprompt testuser
Enter password for new role: ......
```

```
# Create a new database called testdb, owned by testuser.
$ sudo -u postgres createdb --owner=testuser testdb
```

Tailor the PostgreSQL configuration file `/etc/postgresql/9.5/main/pg_hba.conf` to allow user `testuser` to login to PostgreSQL server, by adding the following entry:

```
# TYPE  DATABASE    USER        ADDRESS          METHOD
local   testdb      testuser                     md5
```

Restart PostgreSQL server:

```
$ sudo service postgresql restart
```

Example 1: Connecting to MySQL/PostgreSQL and Executing SQL statements

```
# -*- coding: UTF-8 -*-
"""
sqlalchemy_eg1_mysql: SQLAlchemy Example 1 - Testing with MySQL
"""
from sqlalchemy import create_engine
engine = create_engine('mysql://testuser:xxxx@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()
```

```
# -*- coding: UTF-8 -*-
"""
sqlalchemy_eg1_postgresql: SQLAlchemy Example 1 - Testing with PostgreSQL
"""
from sqlalchemy import create_engine
engine = create_engine('postgresql://testuser:xxxx@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.

Study the console message (enabled via `db.echo = True`) and the output. Take note that a COMMIT is issued after the CREATE TABLE and INSERT statements (i.e., in auto-commit mode).

7.3 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 2: Using ORM to CREATE/DROP TABLE, INSERT, SELECT, and DELETE

```
# -*- coding: UTF-8 -*-
"""
sqlalchemy_eg2: 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://testuser:xxxx@localhost:3306/testdb')
#engine = create_engine('postgresql://testuser:xxxx@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)
```

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 3: ORM with One-to-Many Relation

```
# -*- coding: UTF-8 -*-
"""
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://testuser:xxxx@localhost:3306/testdb')
#engine = create_engine('postgresql://testuser:xxxx@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')
```

How It Works [TODO]

1. Defining relation via `relationship`.
2. Query with JOIN.

Many-to-Many Relationship

See Flask-SQLAlchemy.

Transaction and DB session

See Flask-SQLAlchemy.

7.4 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: 2.1
Location: .../venv/lib/python3.5/site-packages
Requires: SQLAlchemy, Flask
```

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.

`fsqlalchemy_eg1_models.py`

```
# -*- coding: UTF-8 -*-
"""
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
```

1. Flask-SQLAlchemy is simpler than the pure SQLAlchemy! You initialize via `db = SQLAlchemy()`, and subsequently access all properties via `db`.
2. The default tablename is derived from the classname converted to lowercase; or `CamelCase` converted to `camel_case`. You can also set via `__tablename__` property.
3. Flask-SQLAlchemy's base model class defines a constructor that takes `**kwargs` and stores all the keyword arguments given. 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
   ```

4. 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"`.

`fsqlalchemy_eg1_controller.py`

```
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg1_controller: Flask-SQLAlchemy EG 1 - Main controller.
"""
from flask import Flask, render_template
from fsqlalchemy_eg1_models import db, Cafe, load_db  # Flask-SQLAlchemy

# Flask: Initialize
app = Flask(__name__)

# Flask-SQLAlchemy: Initialize
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
#app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://testuser:xxxx@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('fsqlalchemy_eg1_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()
```

1. You need to issue `load_db(db)` under the `test_request_context()`, which tells Flask to behaves as if it is handling a request.

`templates/fsqlalchemy_eg1_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>
```

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

```
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg2_models: 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='XXX 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()
```

1. In this example, the `Supplier` is the parent table; the `Cafe` is the child table. There is a one-to-many relationship between parent and the child tables.
2. In `Supplier` (the parent table), a `relationship` is defined, with a `backref` for the child table to reference the parent table.
3. In `Cafe` (the child table), you can access the parent via the `backref` defined in the `relationship` in the parent table. For example, in '`Cafe`', we use `backref` of '`supplier`' to reference the `Supplier` object (in `fsqlalchemy_eg2_list.html`).
4. Take note that the SQLAlchemy manages the relationship internally! Study the codes in adding objects via relationship, in the above example.
5. With the use of `backref` to establish bi-directional relationship, you can also define the `relationship` in the child class.
6. The `lazy` parameter specifies the loading strategy, i.e., controls when the child record is loaded - lazily (loaded during the first access) or eagerly (loaded together with the parent record):
    - `select` (default): lazily loaded using a SQL SELECT statement.

    - `immediate` (default): eagerly loaded using a SQL SELECT statement.
    - `joined`: eagerly loaded with a SQL JOIN statement.
    - `subquery`: eagerly loaded with a SQL subquery.
    - `dynamic`: lazily loaded by returning a `Query` object, which you can further refine using filters, before executing the Query.
    - `noload`: no loading for supporting "write-only" attribute.

    You can also set the `lazy` for the `backref` using the `backref()` function

    You can enable `SQLALCHEMY_ECHO` to study the various loading strategies.

7. In `ForeignKey()`, you can include keywords `onupdate` and `ondelete` with values of `CASCADE`, `SET NULL`
8. You can use `PrimaryKeyConstraint` to set the primary key to a combination of columns.
9. To define a foreign key with multiple columns, use `ForeignKeyConstraint`.
10. `UNIQUE`: You can set a column to unique via `unique=True`; or a combination of columns via `UniqueConstraint()`.
11. `INDEX`: You can build index on a column via `index=True`; or a combination of columns via `main()`.

`fsqlalchemy_eg2_controller.py`

```
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg2_controller: Flask-SQLAlchemy EG 2 - Using one-to-many relation.
"""
from flask import Flask, render_template, redirect, flash, abort
from fsqlalchemy_eg2_models import db, Supplier, Cafe, load_db  # Flask-SQLAlchemy

# Flask: Initialize
app = Flask(__name__)

# Flask-SQLAlchemy: Initialize
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
#app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://testuser:xxxx@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('fsqlalchemy_eg2_list.html', item_list=_items)

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

`templates/fsqlalchemy_eg2_list.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,)
```

Defining relationship

For one-to-many:

```
class Parent(db.Model): # default tablename 'parent'.
   id = db.Column(db.Integer, primary_key=True)
   children = db.relationship('Child', backref='parent', lazy='dynamic')

class Child(db.Model):  # default tablename 'child'.
   id = db.Column(db.Integer, primary_key=True)
   parent_id = db.Column(db.Integer, db.ForeignKey('parent.id'))
   # property 'parent' defined via backref.
```

For many-to-one:

```
class Parent(db.Model):  # default tablename 'parent'.
   id = db.Column(db.Integer, primary_key=True)
   child_id = db.Column(db.Integer, db.ForeignKey('child.id'))
   child = db.relationship('Child', backref=db.backref('parents', lazy='dynamic'))

class Child(db.Model):  # default tablename 'child'.
   id = db.Column(db.Integer, primary_key=True)
   # property 'parents' defined via backref.
```

For one-to-one:

```
class Parent(db.Model): # default tablename 'parent'.
   id = db.Column(db.Integer, primary_key=True)
   child = db.relationship("Child", uselist=False, backref=db.backref("parent", uselist=False))
         # uselist=False for one-to-one.

class Child(db.Model):  # default tablename 'child'.
   id = db.Column(db.Integer, primary_key=True)
   parent_id = db.Column(db.Integer, db.ForeignKey('parent.id'))
   # Property 'parent' defined via backref.
```

For many-to-many:

```
joint_table = db.Table('joint_table',
    db.Column('left_id', db.Integer, db.ForeignKey('left.id')),
    db.Column('right_id', db.Integer, db.ForeignKey('right.id')))
```

```
class Left(db.Model):
   __tablename__ = 'left'
   id = db.Column(db.Integer, primary_key=True)
   rights = db.relationship('Right', secondary='joint_table', lazy='dynamic',
         backref=db.backref('lefts', lazy='dynamic'))
         # Relate to 'Right' thru an intermediate secondary table

class Right(db.Model):
   __tablename__ = 'right'
   id = db.Column(db.Integer, primary_key=True)
   # Property 'lefts' defined via backref.
```

For many-to-many with additional columns in an Association Object

```
class Association(db.Model):
   __tablename__ = 'association'
   left_id = db.Column(db.Integer, db.ForeignKey('left.id'), primary_key=True)
   right_id = db.Column(db.Integer, db.ForeignKey('right.id'), primary_key=True)
   extra_column = db.Column(String(50))

class Left(db.Model):
   __tablename__ = 'left'
   id = db.Column(db.Integer, primary_key=True)
   rights = db.relationship("Right", secondary=Association, backref="lefts")

class Right(db.Model):
   __tablename__ = 'right'
   id = db.Column(db.Integer, primary_key=True)
   # property 'lefts' defined via backref.
```

Example 3: Using Many-to-Many relation with an Association Table

fsqlalchemy_eg3_models.py

```
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg3_models: Flask-SQLAlchemy EG 3 - Define Models and Database Interface
"""
from flask_sqlalchemy import SQLAlchemy  # Flask-SQLAlchemy

# Flask-SQLAlchemy: Initialize
db = SQLAlchemy()
```

```
# Define the joint table between 'cafe' and 'supplier'.
cafe_supplier = db.Table('cafe_supplier',
    db.Column('cafe_id', db.Integer, db.ForeignKey('cafe.id')),
    db.Column('supplier_id', db.Integer, db.ForeignKey('supplier.id'))
)

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)
    suppliers = db.relationship('Supplier', secondary=cafe_supplier,
        backref=db.backref('items', lazy='dynamic'), lazy='dynamic')

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

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)
    # property 'items' is defined in 'Cafe' via backref.

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

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

    # Add records thru relationship
    _supplier1 = Supplier(name='AAA Corp', id=501)
    _supplier2 = Supplier(name='BBB Corp', id=502)
    _item1 = Cafe(name='Espresso', price=3.19, suppliers=[_supplier1, _supplier2])
    _item2 = Cafe(name='Cappuccino', price=3.29)
    _supplier2.items.append(_item2)
    _supplier3 = Supplier(name='CCC Corp', id=503, items=[_item2])
    db.session.add(_item1)
    db.session.add(_item2)
    db.session.commit()
```

fsqlalchemy_eg3_controllers.py

```python
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg3_controllers: Flask-SQLAlchemy EG 3 - Using one-to-many relation.
"""
from flask import Flask, render_template, redirect, flash, abort
from fsqlalchemy_eg3_models import db, Supplier, Cafe, load_db  # Flask-SQLAlchemy

# Flask: Initialize
app = Flask(__name__)

# Flask-SQLAlchemy: Initialize
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
#app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://testuser:xxxx@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('fsqlalchemy_eg3_list.html', item_list=_items)

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

templates/fsqlalchemy_eg3_list.html

```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 }}
    <ul>
      {% for supplier in item.suppliers %}<li>{{ supplier.name }}</li>
      {% endfor %}
    </ul>
  </li>
  {% endfor %}
</ul>
</body>
</html>
```

[TODO] Explanations

Example 4: Using Many-to-many with additional columns in Association Class

fsqlalchemy_eg4_models.py

```
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg4_models: Flask-SQLAlchemy EG 4 - Many-to-many with extra columns
"""
from flask_sqlalchemy import SQLAlchemy  # Flask-SQLAlchemy

# Flask-SQLAlchemy: Initialize
db = SQLAlchemy()

class CafeSupplier(db.Model):
    """Association Table with extra column"""
    item_id = db.Column(db.Integer, db.ForeignKey('cafe.id'), primary_key=True)
    supplier_id = db.Column(db.Integer, db.ForeignKey('supplier.id'), primary_key=True)
    cost = db.Column(db.Numeric(precision=5, scale=2), default=999.99)

    item = db.relationship('Cafe', backref="assoc_suppliers")
    supplier = db.relationship('Supplier', backref="assoc_items")

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)
    #suppliers = db.relationship('Supplier', secondary='cafe_supplier',
    #      backref=db.backref("items", lazy='dynamic'), lazy='dynamic')

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

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)
    # property 'items' is defined in 'Cafe' via backref.

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

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

    # Add records thru relationship
    _supplier1 = Supplier(name='AAA Corp', id=501)
    _supplier2 = Supplier(name='BBB Corp', id=502)
    _item1 = Cafe(name='Espresso', price=3.19)
    _item2 = Cafe(name='Cappuccino', price=3.29)
    _assoc1 = CafeSupplier(item=_item1, supplier=_supplier1, cost=1.99)
    _assoc2 = CafeSupplier(item=_item1, supplier=_supplier2, cost=1.88)
    _assoc3 = CafeSupplier(item=_item2, supplier=_supplier1, cost=1.77)

#   _supplier2.items.append(_item2)
#   _supplier3 = Supplier(name='CCC Corp', id=503, items=[_item2])
    db.session.add(_assoc1)
    db.session.add(_assoc2)
    db.session.add(_assoc3)
    db.session.commit()
```

fsqlalchemy_eg4_controller.py

```python
# -*- coding: UTF-8 -*-
"""
fsqlalchemy_eg4_controller: Flask-SQLAlchemy EG 4 - Using one-to-many relation with extra columns
"""
from flask import Flask, render_template, redirect, flash, abort
from fsqlalchemy_eg4_models import db, Supplier, Cafe, load_db  # Flask-SQLAlchemy

# Flask: Initialize
app = Flask(__name__)

# Flask-SQLAlchemy: Initialize
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
#app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://testuser:xxxx@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('fsqlalchemy_eg4_list.html', item_list=_items)

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

templates/fsqlalchemy_eg4_list.html

```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 }}
    <ul>
      {% for assoc_supplier in item.assoc_suppliers %}
      <li>{{ assoc_supplier.cost }}, {{ assoc_supplier.supplier.name }}</li>
      {% endfor %}
    </ul>
  </li>
  {% endfor %}
</ul>
</body>
</html>
```

[TODO] Explanations

Cascades

See http://docs.sqlalchemy.org/en/latest/orm/cascades.html.

You can specify the cascade option when defining a relationship, e.g.,

```
suppliers = relationship("Supplier", cascade="all, delete-orphan")
```

The various options are:

- `save-update`: when an object is place into a `Session` via `Session.add()`, all objects in `relationship()` shall also be added to the same `Session`.
- `merge`: The `Session.merger()` operation shall be propagate to its related objects.
- `delete`: When a parent object is marked for deletion, its related child objects should also be marked for deletion. Both object will be deleted during `Session.commit()`.
- `delete-orphan`: Marked the child object for deletion, if it is de-associated from its parent, i.e., its foreign key not longer valid.
- `refresh-expire`: The `Session.expire()` operation shall be propagated to the related objects.
- `expunge`: When the parent object is removed from the `Session` using `Session.expunge()`, the operation shall be propagated to its related objects.
- `all`: `save-update`, `merge`, `refresh-expire`, `expunge`, `delete`; exclude `delete-orphan`.
- The defaults are `save-update` and `merge`.

It can also be defined inside the `backref()` function for applying to the `backref` attribute.

Flask-SQLAlchemy's Query Methods

To execute a query: You can use the SQLAlchemy's query methods (in the base class). Flask-SQLAlchemy adds more methods (in its subclass):

- `all()`: return all objects as a list.
- `first()`: return the first object, or None.
- `first_or_404()`: return the first object, or abort with 404.
- `get(`*`primary-key-id`*`)`: return a single object based on the primary-key identifier, or None. If the primary key consists of more than one column, pass in a tuple.
- `get_or_404(`*`primary-key-id`*`)`: return a single object based on the primary-key identifier, or abort with 404.
- `paginate(page=None, per_page=None, error_out=True)`: return a `Pagination` object of `per_page` items for page number of `page`.
  If `error_out` is True, abort with 404 if a page outside the valid range is requested; otherwise, an empty list of items is returned.

To filter a query object:

- `order_by(`*`criterion`*`)`:
- `limit(`*`limit`*`)`:
- `offset(`*`offset`*`)`:
- `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'`.

Pagination

You can use the `pagination()` method to execute a query, which returns a `Pagination` object.

Example

```
# The URL has a GET parameter page=x
page_num = request.args.get('page', 1, type=int)
      # Retrieve the GET param 'page'. Set to 1 if not present. Coerce into 'int'
pagination = Model.query.order_by(...).paginate(page_num, per_page=20, error_out=False)
items = pagination.items
return render_template('xxx.html', form=form, pagination=pagination)
```

The `Pagination` object has these properties/methods:

- `items`: items for the current page.
- `per_page`: number of items per page.
- `total`: total number for this query.

- `pages`: number of pages.
- `page`, `next_num`, `prev_num`: the current/next/previous page number.
- `has_next`, `has_prev`: True if a next/previous page exists.
- `next`, `has_prev`: True if a next/previous page exists.
- `next()`, `prev()`: return a `Pagination` object for the next/previous page.
- `iter_pages(left_edgs=2, left_current=2, right_current=5, right_edge=2)`: return a sequence of page numbers (for providing links to these pages). Suppose that the current page number is 50 of 80, this iterator returns this page sequence: 1, 2, None, 48, 49, 50, 51, 52, 53, 54, 55, None, 79, 80.

8. RESTful Web Services API

In recent years, there is a tendency to move some business logic to the client side, in a architecture known as Rich Internet Application (RIA). In RIA, the server provides the clients with data retrieval and storage services, becoming a 'web service' or 'remote API'.

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 the RIA clients can 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 favorite nowadays.

REST is a *programming pattern* which describes how data should be transfer 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.

Collections are identified by a trailing slash to give a directory representation. You can also include a version number in the URI, e.g., `/api/1.2/user/`, so that a client can choose a suitable version, whereas `/api/user/` uses the latest version.

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.

The HTTP requests could be synchronous or asynchronous (AJAX). Again, AJAX is becoming popular for Single-Page Architecture (SPA).

8.1 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: 2.12.2
Location: .../venv/lib/python3.5/site-packages
Requires:
```

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

[TODO] Examples

8.2 Flask RESTful API - Roll Your Own

Flask can support 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://testuser:xxxx@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.data)  # 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.data)
```

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

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`

Example 2: AJAX/JSON

```
"""
resteg2_ajax.py: AJAX request with JSON response
"""
import simplejson as json  # Needed to support 'Decimal' field
from flask import Flask, jsonify, request, render_template, abort
from flask_sqlalchemy import SQLAlchemy
from marshmallow import Schema

app = Flask(__name__)
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Needed for CSRF
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@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()

# 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 request.is_xhr:  # AJAX?
            # Return JSON
            if item is None:
                return jsonify({"err_msg": ["We could not find item '{}'".format(id)]}), 404
            else:
                result = item_schema.dump(item)
                return jsonify(result.data)

        else:
            # Return a web page
            if item is None:
                abort(404)
            else:
                return render_template('resteg2_ajax_query.html')

    else:  # if id is None
        items = Cafe.query.limit(3)  # don't return the whole set

        if request.is_xhr:
            # Return JSON
            result = items_schema.dump(items)
            return jsonify(result.data)
        else:
            # Return a web page
            return render_template('resteg2_ajax_query.html')

if __name__ == '__main__':
    # Debug only if running through command line
    app.config['SQLALCHEMY_ECHO'] = True
    app.run(debug=True)
```

templates/resteg2_ajax_query.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Cafe Query</title>
</head>
<body>
  <h1>Cafe Query</h1>
  <ul id="items"></ul>

<script src="https://code.jquery.com/jquery-1.11.2.min.js"></script>
<script>
// Send AJAX request after document is fully loaded
$(document).ready(function() {
   $.ajax({
      // default url of current page and method of GET
   })
      .done( function(response) {
         $(response).each(function(idx, elm) {
            $("#items").append("<li>" + elm['category'] + ", "
                  + elm['name'] + ", $" + elm['price'] + "</li>");
         })
      });
});
</script>
</body>
</html>
```

There are two requests for each URL. The initial request is a regular HTTP GET request, which renders the template, including the AJAX codes, but without any items. When the page is fully loaded, an AJAX request is sent again to request for the items, and placed inside the document.

Turn on the web browser's developer console to trace the request/response messages.

Also, try using firefox's plug-in 'HttpRequester' to trigger a AJAX GET request.

8.3 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: 0.3.5
Summary: Simple framework for creating REST APIs
Requires: Flask, aniso8601, pytz, six
```

Flask-Restful Example 1: Using Flask-Restful Extension

```
"""
frestful_eg1: 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)
```

1. We define two URLs: `/api/item/` for get-all and create-new via GET and POST methods; and `/api/item/<item_id>` for get, update, delete via GET, PUT, and DELETE methods.
2. We extend the `Resource` class to support all these methods.
3. In this example, we did not use an actual data model.
4. 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, which is much easier to use with a user-friendly graphical interface.
   For example, use Firefox's HttpRequester to send the following requests:
   - GET request to `http://localhost:5000/api/item/` to list all items.
   - GET request to `http://localhost:5000/api/item/1` to list one item.
   - POST request to `http://localhost:5000/api/item/` to create a new item.
   - PUT request to `http://localhost:5000/api/item/1` to update one item.
   - DELETE request to `http://localhost:5000/api/item/1` to delete one item.

Sending AJAX-POST/PUT/PATCH/DELETE HTTP Requests

When you enter a URL on a web browser, an HTTP GET request is sent. You can send a POST request via an HTML Form. There are a few ways to test PUT/PATCH/DELETE/Ajax-POST requests:

1. Via web browser's plug-in such as Firefox's HttpRequester.
2. Via the `curl` command, e.g.,

```
// Show manual page
$ man curl
... manual page ...
// Syntax is:
// $ curl options url

// Send GET request
$ curl --request GET http://localhost:5000/api/item/1
"reading item 1"

// Send DELETE request. To include the response header
$ curl --include --request DELETE http://localhost:5000/api/item/1
HTTP/1.0 204 NO CONTENT
Content-Type: application/json
Content-Length: 0
Server: Werkzeug/0.11.15 Python/3.5.2
Date: Thu, 16 Mar 2017 02:44:11 GMT

// Send PUT request, with json data and additional header
$ curl --include --request PUT --data '{"price":"9.99"}'
       --Header "Content-Type: application/json" http://localhost:5000/api/item/1
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 16
Server: Werkzeug/0.11.15 Python/3.5.2
Date: Thu, 16 Mar 2017 03:00:43 GMT

"update item 1"
```

3. Via client-side script in JavaScript/jQuery/AngularJS
4. Via Flask client, e.g., [TODO]

Flask-Restful Example 2: with Flask-SQLAlchemy

[TODO]

Flask-Restful Example 3: Argument Parsing

[TODO]

8.4 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
Location: /usr/local/lib/python2.7/dist-packages
Requires: flask, sqlalchemy, python-dateutil, mimerender
```

Example: Using Flask-Restless Extension

```
"""
frestless_eg1.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://testuser:xxxx@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.

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: 901
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/0.11.2 Python/2.7.6
{
  "num_results": 5,
  "objects": [
    {
      "category": "coffee",
      "id": 1,
      "name": "Espresso",
      "price": 3.19
    },
    .....
  ],
  "page": 1,
  "total_pages": 1
}

# 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: 79
Vary: Accept
Content-Type: application/json
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Thu, 03 Dec 2015 18:55:24 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: 78
Location: http://127.0.0.1:5000/api/cafe/6
Vary: Accept
Content-Type: application/json
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Thu, 03 Dec 2015 18:53:48 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/9
HTTP/1.0 204 NO CONTENT
Content-Type: application/json
Content-Length: 0
Vary: Accept
Content-Type: application/json
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Thu, 03 Dec 2015 18:57:32 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: 76
Vary: Accept
Content-Type: application/json
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Thu, 03 Dec 2015 19:09:36 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).

9. Unit Testing and Acceptance Testing for Flask Apps

9.1 Unit Testing

Read "Python Unit Testing" for the basics.

To test a Flask app under the web, we need to simulate the requests and retrieve the responses. Flask provides a test-client tools (created via `test_client()`) for sending HTTP requests and retrieving the responses.

Example

`uteg1_controller.py`

```
"""
uteg1_controller: controller for main app
"""
from flask import Flask, render_template

def setup_app(app):
    """Register the routes and view functions for the given app"""
    @app.route('/')
    @app.route('/<name>')
    def hello(name=None):
        return render_template('uteg1.html', name=name)

def app_factory(name=__name__, debug=False):
    """Generate an instance of the app"""
    app = Flask(name)
    app.debug = debug
    setup_app(app)
    return app

if __name__ == '__main__':
    # Run the app
    app = app_factory(debug=True)
    app.run()
```

`templates/uteg1.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello</title>
</head>
<body>
  <h1>Hello, {% if name %}{{ name }}{% else %}world{% endif %}</h1>
</body>
</html>
```

Try running the application, with URL `http://localhost:5000` and `http://localhost:5000/peter`.

The unit test module for testing the `hello()` view function is as follows:

`uteg1_hello.py`

```
"""
uteg1_hello: Test the hello() view function
"""
import unittest
from flask import url_for, request
from uteg1_controller import app_factory

class TestHello(unittest.TestCase):
    """Test Case for hello() view function"""

    def setUp(self):
        """Called before each test method"""
        # Generate a Flask app instance and a test client for every test method
        self.app = app_factory(debug=True)
        self.client = self.app.test_client()

    def tearDown(self):
        """Called after each test method"""
        pass

    def test_hello_no_name(self):
        """Test hello() view function"""
        with self.app.test_request_context():  # Setup a request context
            _path = url_for('hello')  # url_for() needs an application context
            self.assertEquals(request.path, '/')  # request needs a request context
                # For illustration, as path could change
            response = self.client.get(_path)  # Send GET request and retrieve the response
            self.assertIn(b'Hello, world', response.data) # response.data in UTF-8

    def test_hello_with_name(self):
        """Test hello(name) view function"""
        with self.app.test_request_context():
            _name = 'Peter'
            _path = url_for('hello', name=_name)
            response = self.client.get(_path)
            self.assertIn(b'Hello, ' + _name.encode('UTF-8'), response.data)

    def test_hello_with_name_escape(self):
        """Test hello(name) with special characters in name"""
        with self.app.test_request_context():
            _name = '<peter>'  # Contains special HTML characters
            _path = url_for('hello', name=_name)
            response = self.client.get(_path)  # Jinja2 automatically escapes special characters
            _escaped_name = _name.replace('<', '&lt;')
            _escaped_name = _escaped_name.replace('>', '&gt;')
            self.assertIn(b'Hello, ' + _escaped_name.encode('UTF-8'), response.data)

if __name__ == '__main__':
    unittest.main()  # Run the test cases in this module
```

How It Works

1. In `setup()`, which is run before each test method, we define two instance variables `self.app` and `self.client`, to be used by the test method.
2. There are 3 tests in the above test scripts. Run the test script and check the test results:

   ```
   $ python test_view_hello.py
   ...
   ----------------------------------------------------------------------
   Ran 3 tests in 0.020s

   OK
   ```

3. In this example, there is no necessity to setup a request context (you can remove the with-statement). But if you need to access properties such as `current_user` (of the Flask-Login), then the request context is necessary.

Application Context and Request Context

During normal operation, when a request is made to a view function, Flask automatically setups a request context (containing objects such as `request`, `session`, `current_user` (for Flask-Login), etc.) and application context (containing `current_app`, `url_for()`, etc.); and pushes them onto the context stacks. However, you need to manage these contexts yourself in testing.

If you simply issue a "`self.client.get|post()`" request, there is no need to setup a request context - all thread-local objects are torn down when the get|post request completes.

However, to access the `url_for` (in app context) and `request` (in request context), you need to setup the contexts. The easier mean is to wrap the statements under the "`with self.app.test_request_context():`". The with-statement creates the contexts, pushes them onto the stack upon entry, and pops out upon exit automatically.

9.2 Unit Testing with Flask-Testing Extension

Reference: Flask-Testing @ https://pythonhosted.org/Flask-Testing/.

The Flask-Testing extension provides unit testing utilities for Flask, in particular, handling request to Flask app.

Installing Flask-Testing Extension

```
# Activate your virtual environment
(venv)$ pip install Flask-Testing
Successfully installed Flask-Testing-0.6.1

(venv)$ pip show Flask-Testing
Metadata-Version: 2.0
Name: Flask-Testing
Version: 0.6.1
Summary: Unit testing for Flask
Requires: Flask
```

Example: Using Flask-Testing Extension

We shall rewrite our test script using Flask-Testing, for the above example.

`uteg1_controller.py`, `templates/uteg1.html`: same as the above example.

`futeg1_hello.py`

```
"""
futeg1_hello: Using Flask-Testing Extension to test hello() view function
"""
import unittest
from flask import url_for, request
from flask_testing import TestCase  # Flask-Testing extension
from uteg1_controller import app_factory

class TestHello(TestCase):   # from Flask-Testing Extension
    """Test Case for hello() view function"""
    def create_app(self):
        """
        To return a Flask app instance.
        Run before setUp() for each test method.
        Automatically setup self.app, self.client, and manage the contexts.
        """
        print('run create_app()')
        # Generate a Flask app instance and a test client for every test method
        app = app_factory(debug=True)
        app.config['TESTING'] = True
        return app

    def setUp(self):
        """Called before each test method"""
        print('run setUp()')

    def tearDown(self):
        """Called after each test method"""
        print('run tearDown()')

    def test_hello_no_name(self):
        """Test hello() view function"""
        _path = url_for('hello')              # url_for() needs app context
        self.assertEquals(request.path, '/')  # request need request context
            # For illustration, as path could change
        response = self.client.get(_path)  # Send GET request and retrieve the response
        self.assertIn(b'Hello, world', response.data) # response.data in UTF-8

    def test_hello_with_name(self):
        """Test hello(name) view function"""
        _name = 'Peter'
        _path = url_for('hello', name=_name)
        response = self.client.get(_path)
        self.assertIn(b'Hello, ' + _name.encode('UTF-8'), response.data)

    def test_hello_with_name_escape(self):
        """Test hello(name) with special characters in name"""
        _name = '<peter>'  # Contains special HTML characters
        _path = url_for('hello', name=_name)
        response = self.client.get(_path)  # Jinja2 automatically escapes special characters
        _escaped_name = _name.replace('<', '&lt;')
        _escaped_name = _escaped_name.replace('>', '&gt;')
        self.assertIn(b'Hello, ' + _escaped_name.encode('UTF-8'), response.data)
```

```
if __name__ == '__main__':
    unittest.main()  # Run the test cases in this module
```

How It Works

1. We create the test-case class by sub-classing `flask_testing.TestCase` class, instead of `unittest.TestCase`.
2. Flask-Testing requires a method called `create_app()`, which shall return a Flask app instance. The documentation is not clear here, but the source code shows that a method called `_pre_setup()`, which is called before `setUp()` contains the followings to setup the `self.app`, `self.client`, and the request contexts.

   ```
   def _pre_setup(self):
       # Call create_app() and assign to self.app
       self.app = self.create_app()

       # Create a test client
       self.client = self.app.test_client()

       # Create a request context and push onto the stack
       self._ctx = self.app.test_request_context()
       self._ctx.push()
       ......
   ```

3. Take note that we use the same example as the `unittest`, but the codes are much simpler.
4. Flask-Testing adds the following assert instance methods (in addition to those provided by `unittest` module):
   - `assertXxx(`*`response`*`, `*`message`*`=None)`: where `response` is the Flask's `response` object; `Xxx` is the `response.status_code`. The supported status-codes are 200, 400, 401, 403, 404, 405 and 500.
   - `assertStatus(`*`response`*`, `*`status_code`*`, `*`message`*`=None)`:
   - `assertMessageFlashed(`*`message`*`, `*`category`*`='message', `*`message`*`=None)`:
   - `assertContext(`*`name`*`, `*`value`*`, `*`message`*`=None)`: Check if *name* exists in the context and equals to the given *value*.
   - `assertRedirects(`*`response`*`, `*`location`*`, `*`message`*`=None)`: Check if the *response* is a redirect to the given *location*.
   - `assertTemplateUsed(`*`template_name`*`, `*`message`*`=None)`: Check if the given template is used in the request.

9.3 Acceptance Test with Selenium

[TODO]

10. Some Useful Flask Extensions

Reference: Flask Extension @ http://flask.pocoo.org/extensions/.

So far, we have discussed:

- Flask-WTF for web forms
- Flask-SQLAlchemy for database queries
- Flask-Restless for generating RESTful API

There are many more extensions in the Flask Ecosystem. We shall cover these in this section:

- Flask-Login for User Session Management
- Flask-Principal for permissions
- Flask-Admin for building admin interfaces

There are many more:

- Flask-Security for authentication
- Flask-Assets for asset management
- Flask-DebugToolbar for debugging and profiling
- Flask-Markdown for forum posts
- Flask-Script for basic commands

10.1 Flask-Login Extension for User Session Management

Reference: Flask-Login @ https://flask-login.readthedocs.org/en/latest/.

Flask-Login Extension is an authentication and session manager. It handles common tasks of logging in, logging out, and keep track of your users' sessions over extended periods of time.

To install Flask-Login

```
# Activate your virtual environment
(venv)$ pip install Flask-Login
Successfully installed Flask-Login-0.4.0

(venv)$ pip show Flask-Login
Name: Flask-Login
Version: 0.4.0
Location: .../venv/lib/python3.5/site-packages
Requires: Flask
```

Hashing Password

In the examples, we also use `passlib` package to hash the password. To install:

To install `passlib` package:

```
# Activate your virtual environment
(venv)$ pip install passlib
Successfully installed passlib-1.7.1

(venv)$ pip show passlib
Name: passlib
Version: 1.7.1
Summary: comprehensive password hashing framework supporting over 30 schemes
Location: .../venv/lib/python3.5/site-packages
Requires:
```

We also need to install `bcrypt` package:

```
# Need these libraries:
$ sudo apt-get install build-essential libssl-dev libffi-dev python-dev

# Also need these:
# Activate your virtual environment
(venv)$ pip install cffi
Successfully installed cffi-1.9.1 pycparser-2.17

# Now, you can install bcrypt
(venv)$ pip install bcrypt
Successfully installed bcrypt-3.1.2

(venv)$ pip show bcrypt
Name: bcrypt
Version: 3.1.2
Summary: Modern password hashing for your software and your servers
Location: .../venv/lib/python3.5/site-packages
Requires: cffi, six
```

To use `bcrypt` for hashing password:

```
from passlib.hash import bcrypt

# To hash
passwd_hash= bcrypt.encrypt(password)
```

```
# To verify
if bcrypt.verify(password, passwd_hash):
   ......
```

Password Hashing with Flask-Bcrypt Extension

Reference: Flask-Bcrypt @ http://flask-bcrypt.readthedocs.org/en/latest/.

Flask-Bcrypt is a Flask extension that provides bcrypt hashing utilities. Unlike hashing algorithm such as MD5 and SHA1, which are optimized for speed, bcrypt is intentionally "de-optimized" to be slow, so as to protect against cracking with powerful hardware.

To install Flask-Bcrypt Extension:

```
# Activate your virtual environment
(venv)$ pip install flask-bcrypt
Successfully installed flask-bcrypt-0.7.1

(venv)$ pip show flask-bcrypt
Name: Flask-Bcrypt
Version: 0.7.1
Summary: Brcrypt hashing for Flask.
Location: .../venv/lib/python3.5/site-packages
Requires: Flask, bcrypt
```

Take note that Flask-Bcrypt requires bcrypt.

To use Flask-Bcrypt Extension:

```
from flask import Flask
from flask_bcrypt import Bcrypt

# Construct an instance and bind to Flask app
app = Flask(__name__)
bcrypt = Bcrypt(app)
      # You can also use:
      #    bcrypt = Bcrypt()
      #    bcrypt.init_app(app)

# To hash
pw_hash = bcrypt.generate_password_hash('xxxx')

# To verify against the hash
bcrypt.check_password_hash(pw_hash, 'xxxx')
```

Example: Using Flask-Login Extension for Login, Logout and User Session Management

flogin_eg1_models.py

```
# -*- coding: UTF-8 -*-
"""
flogin_eg1_models: Flask-Login Example - Define Models and Database Interface
"""
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt  # For hashing password

# Flask-SQLAlchemy: Initialize
db = SQLAlchemy()

# Flask-Bcrypt: Initialize
bcrypt = Bcrypt()

# Flask-SQLAlchemy: Define a 'User' model mapped to table 'user' (default to lowercase).
# Flask-Login: needs 4 properties (You could also use default implementations in UserMixin)
class User(db.Model):
    username = db.Column(db.String(30), primary_key=True)
    pwhash = db.Column(db.String(300), nullable=False)  # Store password hash
    active   = db.Column(db.Boolean, nullable=False, default=False)

    def __init__(self, username, password, active=False):
        """Constructor"""
        self.pwhash = bcrypt.generate_password_hash(password)  # hash submitted password
        self.username = username
        self.active = active

    @property  # Convert method to property (instance variable)
    def is_authenticated(self):
        """Flask-Login: return True if the user's credential is authenticated."""
        return True

    @property
    def is_active(self):
        """Flask-Login: return True if the user is active."""
        return self.active

    @property
    def is_anonymous(self):
        """Flask-Login: return True for anonymous user."""
        return False
    def get_id(self):  # This is method. No @property
        """Flask-Login: return a unique ID in unicode."""
        return self.username

    def verify_password(self, password_in):
        """Verify the given password with the stored password hash"""
        return bcrypt.check_password_hash(self.pwhash, password_in)

def load_db(db):
    """Create database tables and records"""
    db.drop_all()
    db.create_all()
    db.session.add_all([User('user1', 'xxxx', True),
                        User('user2', 'yyyy')])  # inactive
    db.session.commit()
```

1. We included the password hashing and verification in the `User` model. This shows the power and flexibility of ORM, which cannot be done in relational database.
2. Flask-Login requires a `User` model with the following properties/methods:
   - an `is_authenticated` property that returns `True` if the user has provided valid credentials.
   - an `is_active` property that returns `True` if the user's account is active.
   - a `is_anonymous` property that returns `True` if the current user is an anonymous user.
   - a `get_id()` method that returns the unique ID for that object.

   You could implement these yourself, or use the default implementation in `UserMixin` class (see source code @ https://flask-login.readthedocs.org/en/latest/_modules/flask_login.html#UserMixin).

   Take note that we decorate `is_authenticated`, `is_active` and `is_anonymous` with `@property`, to turn the method into property. Without this decorator, they won't work (it took me a few hours to find out)!

`flogin_eg1_forms.py`

```
# -*- coding: UTF-8 -*-
"""
flogin_eg1_forms: Flask-Login Example - Login Form
"""
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField
from wtforms.validators import InputRequired, Length

# Define the LoginRorm class by sub-classing FlaskForm
class LoginForm(FlaskForm):
    # This form contains two fields with validators
    username = StringField('User Name:', validators=[InputRequired(), Length(max=20)])
    passwd = PasswordField('Password:', validators=[Length(min=4, max=16)])
```

flogin_eg1_controller.py

```
# -*- coding: UTF-8 -*-
"""
flogin_eg1_controller: Flask-Login Example - Using Flask-Login for User Session Management
"""
from flask import Flask, render_template, redirect, flash, abort, url_for
from flask_login import LoginManager, login_user, logout_user, current_user, login_required
from flogin_eg1_models import db, User, load_db, bcrypt  # Our models
from flogin_eg1_forms import LoginForm           # Our web forms

# Flask: --- Setup ----------------------------------------------
app = Flask(__name__)

# Flask-WTF: ---  Setup ------------------------------------------
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Flask-WTF: needed for CSRF

# Flask-Bcrypt: ---  Setup ----------------------------------
bcrypt.init_app(app)

# Flask-SQLAlchemy: ---  Setup ----------------------------------
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/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)

# Flask-Login: ---  Setup ---------------------------------------
login_manager = LoginManager()
login_manager.init_app(app)

@login_manager.user_loader
def user_loader(username):
    """Flask-Login: Given a username, return the associated User object, or None"""
    return User.query.get(username)

# Flask: --- Routes ---------------------------------------------
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()  # Construct a LoginForm

    if form.validate_on_submit():  # POST request with valid data?
        # Retrieve POST parameters (alternately via request.form)
        username = form.username.data
        passwd = form.passwd.data
        # Query 'user' table with 'username'. Get first row only
        user = User.query.filter_by(username=username).first()

        # Check if username presents?
        if user is None:
            flash('Username or Password is incorrect')  # to log incorrect username
            return redirect(url_for('login'))

        # Check if user is active? (We are not using Flask-Login inactive check!)
        if not user.active:
            flash('Your account is inactive')  # to log inactive user
            return redirect(url_for('login'))

        # Verify password
        if not user.verify_password(passwd):
            flash('Username or Password is incorrect')  # to log incorrect password
            return redirect(url_for('login'))

        # Flask-Login: establish session for this user
        login_user(user)
        return redirect(url_for('startapp'))

    # Initial GET request, and POST request with invalid data
    return render_template('flogin_eg1_login.html', form=form)

@app.route('/startapp')
@login_required  # Flask-Login: Check if user session exists
def startapp():
    return render_template('flogin_eg1_startapp.html')

@app.route('/logout')
@login_required
def logout():
    # Flask-Login: clear this user session
    logout_user()
    return redirect(url_for('login'))

if __name__ == '__main__':
    # Debug only if running through command line
    app.config['SQLALCHEMY_ECHO'] = True
    app.debug = True
    app.run()
```

1. Flask-Login requires you to define a `user_loader()` method which, given a username, returns the associated `User` object.
2. You can use `user_login(anUserInstance)` to login and establish a user session; and `user_logout()` to logout the current user and clear the session.
3. Use `@login_required` decorator to mark those routes, that require login.
4. The property `current_user`, a proxy for the current user, is available to all views (see the templates below).
5. Flask-Login issues "401 Unauthorized" for inactive user, same as incorrect credential. I decided to do my own check for inactive user, so as to issue (and log) different messages.

`templates/flogin_eg1_base.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Application</title>
  <meta charset="utf-8">
</head>
<body>
{# Display flash messages, if any, for all pages #}
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class='flashes'>
    {% for message in messages %}<li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}

{# The body contents here #}
{% block body %}{% endblock %}
</body>
</html>
```

1. The `flogin_eg1_base.html` template displays all the flash message, and define a `body` block.

`templates/flogin_eg1_login.html`

```
{% extends "flogin_eg1_base.html" %}
{% block body %}
<h1>Login</h1>
```

```
<form method="POST">
  {{ form.hidden_tag() }}  {# Renders any hidden fields, including the CSRF #}
  {% for field in form if field.widget.input_type != 'hidden' %}
    <div class="field">
      {{ field.label }} {{ field }}
    </div>
    {% if field.errors %}
      {% for error in field.errors %}
        <div class="field_error">{{ error }}</div>
      {% endfor %}
    {% endif %}
  {% endfor %}
  <input type="submit" value="Go">
</form>
{% endblock %}
```

1. The `flogin_eg1_login.html` template extends the `flogin_eg1_base.html`, and displays the `LoginForm`.

`templates/flogin_eg1_startapp.html`

```
{% extends "flogin_eg1_base.html" %}
{% block body %}

{% if current_user.is_authenticated %}
  <h1>Hello, {{ current_user.username }}!</h1>
{% endif %}

<a href="{{ url_for('logout') }}">LOGOUT</a>
{% endblock %}
```

1. The `flogin_eg1_startapp.html` template extends the `flogin_eg1_base.html`, and uses the `current_user` property to display the `username`.

Try:

1. Login with invalid data to trigger form data validation. Observe the error messages.
2. Login with incorrect username or password. Check the flash message.
3. Login with correct username/password. (Redirect to `flogin_eg1_startapp.html`)
4. Login with inactive user. Check the flash message.
5. Access /startapp and /logout pages with logging in (Error: 401 Unauthorized).

More on Flask-Login

When a user attempts to access a `login_required` view within logging in, Flask-Login will flash a message and redirect him to `LoginManager.login_view` (with a URL appending with `?next=page-trying-to-access`) with status code of "302 Found"; or abort with "401 Unauthorized" if `LoginManager.login_view` is not set. You can also customize the flash message in `LoginManager.login_message` and `Login_manager.login_message_category`.

10.2 Python-LDAP

The Lightweight Directory Access Protocol (LDAP) is a distributed directory service protocol that runs on top of TCP/IP. LDAP is based on a subset of X.500 standard. A common usage of LDAP is to provide a single-sign-on where a user can use a common password to access many services.

LDAP provides a mechanism for connecting to, searching and modifying Internet directories. A client may request for the following operations:

1. StartTLS: Use TLS (or SSL) for secure connection.
2. Bind/Unbind: to the server.
3. Search: search and retrieve directory entries.
4. Add, delete, modify, compare an entry.

A directory entry contains a set of attributes, e.g.,

```
dn: cn=Peter Smith,ou=IT Dept,dc=example,dc=com
cn: Peter Smith
givenName: Peter
sn: Smith
......
```

1. `dn`: the *Distinguish Name* of the entry. "`cn=Peter Smith`" is the RDN (Relative Distinguish Name); "`dc=example,dc=com`" is the DN (Domain Name), where `dc` denotes Domain Component.
2. `cn`: *Common Name*
3. `sn`: *Surname*

Python-LDAP provides utilities to access LDAP directory servers using OpenLDAP.

Installing Python-LDAP

```
# These Python packages are needed
$ sudo apt-get install python-dev libldap2-dev libsasl2-dev

# Activate your virtual environment
(venv)$ pip install python3-ldap  # For Python 3
Successfully installed pyasn1-0.2.2 python3-ldap-0.9.8.4
(venv)$ pip install python-ldap     # For Python 2
```

```
(venv)$ pip show python3-ldap
Name: python3-ldap
Version: 0.9.8.4
Summary: A strictly RFC 4511 conforming LDAP V3 pure Python client.
         Same codebase for Python 2, Python3, PyPy and PyPy 3
Location: .../venv/lib/python3.5/site-packages
Requires: pyasn1
```

Example: Authenticate a User

```
import ldap

# Connect to the LDAP server
conn = ldap.initialize('ldap://server-name')

# Set LDAP options, if needed
conn.set_option(ldap.OPT_X_TLS_DEMAND, True)

# Start a secure connection
conn.start_tls_s()  # s for synchronous operation

# Authenticate a user by binding to the directory
# dn: user's distinguish name
# pw: password
# Raise LDAPError if bind fails
try:
    conn.bind_s(dn, pw)   # You need to figure out the DN
    print('Authentication Succeeds')
except ldap.LDAPError as e:
    print(e)
    print('Authentication Fails')
```

10.3 Flask-Principal Extension

Reference: Flask-Principal @ http://pythonhosted.org/Flask-Principal/.

Installing Flask-Principal Extension

```
# Activate your virtual environment
(venv)$ pip install flask-principal
Successfully installed blinker-1.4 flask-principal-0.4.0
```

```
(venv)$ pip show flask-principal
Name: Flask-Principal
Version: 0.4.0
Summary: Identity management for flask
Location: .../venv/lib/python3.5/site-packages
Requires: Flask, blinker
```

Using Flask-Principal Extension

Flask-Principal is a permission extension, which manages who can access what and to what extent. It is often used together with Flask-Login, which handles login, logout and user session.

Flask-Principal handles permissions through these objects:

1. `Permission` and `Need`: A `Permission` object contains a list of `Need`s that must be satisfied in order to do something (e.g., accessing a link). A `Need` object is simply a `namedtuple` (in `collections` module), e.g., `RoleNeed('admin')`, `UserNeed('root')`. You could define your own needs, or use pre-defined needs such as `RoleNeed(`*`role_name`*`)`, `UserNeed(`*`username`*`)` or `ItemNeed`.
   To create a `Permission`, invoke `Permission(*Needs)` constructor, e.g.,

   ```
   # Create a 'Permission' with a single 'Need', in this case a 'RoleNeed'.
   admin_permission = Permission(RoleNeed('admin'))

   # Create a 'Permission' with a set of 'Need's.
   root_admin_permission = Permission(RoleNeed('admin'), UserNeed('root'))
   ```

2. `Identity`: An `Identity` instance identifies a user. This instance shall be created immediately after logging in to represent the current user, via constructor `Identity(`*`id`*`)` where *`id`* is a unique ID.
   An `Identity` instance possesses a list of `Need`s that it can satisfy (kept in attribute `provides`). To add a `Need` to an `Identity`:

   ```
   identity.provides.add(RoleNeed('admin'))
   identity.provides.add(UserNeed('root'))
   ```

   There is a pre-defined `AnonymousIdentity`, which provides no `Need`s, to be used after a user has logged out.

3. `IdentityContext(`*`permission`*`)`: The `IdentityContext` object is used to test an `Identity` against a `Permission`. Recall that an `Identity` has a set of `Need`s it can provides; while a protected resource has a `Permission` which is also a set of `Need`s. These two sets are compared to determine whether access could be granted.
   You can test an `IdentityContext` (against a `Permission`) as a decorator, or under a context manager in a `with`-statement, e.g.,

   ```
   admin_permission = Permission(RoleNeed('admin'))

   # Protect a view via a 'decorator'
   @app.route('/admin')
   @admin_permission.require()  # Does the current Identify provide RoleNeed('admin')?
   ```

```
def do_admin():
    return 'Only if you have role of admin'

# Protect under a context manager in with-statement
@app.route('/admin1')
def do_admin1():
    with admin_permission.require():  # Does the current Identify provide RoleNeed('admin')?
        return 'Only if you have role of admin'
```

Example 1: Using Flask-Principal for Permissions with Flask-Login for Authentication

fprin_eg1_models.py

```
# -*- coding: UTF-8 -*-
"""
fprin_eg1_models: Flask-Principal Example - Define Models and Database Interface
"""
from flask_login import UserMixin
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt

db = SQLAlchemy()
bcrypt = Bcrypt()

class User(db.Model, UserMixin):
    """
    Flask-SQLAlchemy: Define a 'User' model mapped to table 'user'.
    Flask-Login: Needs 4 properties/method. Use default implementation in UserMixin.
    """

    username = db.Column(db.String(30), primary_key=True)
    pwhash   = db.Column(db.String(300), nullable=False)
    active   = db.Column(db.Boolean, nullable=False, default=False)
    roles    = db.relationship('Role', backref='user', lazy='dynamic')
        # A user can have zero or more roles

    def __init__(self, username, password, active=False):
        """Constructor"""
        self.pwhash = bcrypt.generate_password_hash(password)  # hash password
        self.username = username
        self.active = active

    @property  # Override UserMixin
    def is_active(self):
        """Flask-Login: return True if the user is active."""
        return self.active

    def get_id(self):  # Override UserMixin
        """Flask-Login: return a unique ID in unicode."""
        return self.username

    def verify_password(self, password_in):
        """Verify given password with stored hash password"""
        return bcrypt.check_password_hash(self.pwhash, password_in)

class Role(db.Model):
    """
    Define the 'Role' model mapped to table 'role'
    """
    rolename = db.Column(db.String(60), primary_key=True)
    username = db.Column(db.String(30), db.ForeignKey('user.username'), primary_key=True)

def load_db(db):
    """Initialize the database tables and records"""
    db.drop_all()
    db.create_all()
    db.session.add_all([User('user1', 'xxxx', True),
                        User('user2', 'yyyy', True)])
    db.session.commit()
    db.session.add_all(
         [Role(rolename='admin', username='user1'),
          Role(rolename='user', username='user1'),  # multiple roles
          Role(rolename='user', username='user2')])
    db.session.commit()
```

1. We added a `Role` model with roles of `'admin'` and `'user'`. We intend to use the pre-defined `RoleNeed(`*`rolename`*`)`.
2. In this example, a user could have one or more roles, which can be retrieved via the relationship `roles`.

`flogin_eg1_forms.py`: Use the same form of Flask-Login example.

`fprin_eg1_controller.py`

```
# -*- coding: UTF-8 -*-
"""
fprin_eg1_controller: Flask-Principal Example - Testing Flask-Principal for managing permissions with Flask-Login
"""
from flask import Flask, render_template, redirect, flash, abort, url_for, session
from flask_login import LoginManager, login_user, logout_user, current_user, login_required
from flask_principal import Principal, Permission, Identity, AnonymousIdentity
from flask_principal import RoleNeed, UserNeed, identity_loaded, identity_changed
from fprin_eg1_models import db, User, Role, load_db, bcrypt   # Our models
from flogin_eg1_forms import LoginForm                  # Our web forms

# Flask: --- Setup ---------------------------------------------
app = Flask(__name__)

# Flask-WTF: ---  Setup ----------------------------------------
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Flask-WTF: needed for CSRF

# Flask-Bcrypt: ---  Setup -----------------------------------
bcrypt.init_app(app)

# Flask-SQLAlchemy: ---  Setup -----------------------------------
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
db.init_app(app)

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

# Flask-Login: ---  Setup ---------------------------------------
login_manager = LoginManager()
login_manager.init_app(app)

@login_manager.user_loader
def user_loader(username):
    """For Flask-Login. Given a username, return the associated User object, or None"""
    return User.query.get(username)

# Flask-Principal: ---  Setup ------------------------------------
principal = Principal()
principal.init_app(app)

# Flask-Principal: Create a permission with a single RoleNeed
admin_permission = Permission(RoleNeed('admin'))

# Flask-Principal: Add the Needs that this user can satisfy
@identity_loaded.connect_via(app)
def on_identity_loaded(sender, identity):
    # Set the identity user object
    identity.user = current_user
    # Add the UserNeed to the identity (We are not using UserNeed here)
    if hasattr(current_user, 'username'):
        identity.provides.add(UserNeed(current_user.username))

    # Assuming the User model has a list of roles, update the
    # identity with the roles that the user provides
    if hasattr(current_user, 'roles'):
        for role in current_user.roles:
            identity.provides.add(RoleNeed(role.rolename))

# Flask: --- Routes ----------------------------------------------
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():  # POST request with valid data?
        # Retrieve POST parameters (alternately via request.form)
        _username = form.username.data
        _passwd = form.passwd.data
        # Query 'user' table with 'username'. Get first row only
        user = User.query.filter_by(username=_username).first()

        # Check if username presents?
        if user is None:
            flash('Username or Password is incorrect')  # to log incorrect username
            return redirect(url_for('login'))

        # Check if user is active? (We are not using Flask-Login inactive check!)
        if not user.active:
            flash('Your account is inactive')  # to log inactive user
            return redirect(url_for('login'))

        # Verify password
        if not user.verify_password(_passwd):
            flash('Username or Password is incorrect')  # to log incorrect password
            return redirect(url_for('login'))

        # Flask-Login: establish session for this user
        login_user(user)

        # Flask-Principal: Create an Identity object and
        #   signal that the identity has changed,
        #    which triggers on_identify_changed() and on_identify_loaded()
        # The Identity() takes a unique ID
        identity_changed.send(app, identity=Identity(user.username))

        return redirect(url_for('startapp'))

    # Initial GET request, and POST request with invalid data
    return render_template('flogin_eg1_login.html', form=form)

@app.route('/startapp')
@login_required              # Flask-Login: Check if user session exists
@admin_permission.require()  # Flask-Principal: Check if user having RoleNeed('admin')
def startapp():
    return render_template('flogin_eg1_startapp.html')

@app.route('/logout')
@login_required
def logout():
    # Flask-Login: Clear this user session
    logout_user()

    # Flask-Principal: Remove session keys
    for key in ('identity.name', 'identity.auth_type'):
        session.pop(key, None)
    # Flask-Principal: the user is now anonymous
    identity_changed.send(app, identity=AnonymousIdentity())

    return redirect(url_for('login'))

if __name__ == '__main__':
    # Debug only if running through command line
    app.config['SQLALCHEMY_ECHO'] = True
    app.debug = True
    app.run()
```

1. Immediately after the user has logged in, an `Identity` object is created, and `identity_changed` signal sent. This signal triggers `on_identity_changed()` (not implemented), and in turn `on_identity_loaded()`.
2. In `on_identity_loaded()`, we update the `RoleNeed` and `UserNeed` for which this user can provide, based on information about this user (`roles` and `username`).
3. The view `main` is protected with decorator `@admin_permission.require()`. In other words, only users who can provide `RoleNeed('admin')` are granted access.
4. After the user has logged out, we remove the current identity, set the identity to `AnonymousIdentity`, and signal `identity_changed` which updates the `RoleNeed` and `UserNeed` (to nothing).

`templates`: Use the same templates of Flask-Login example.

Try:

1. Login with `user1`, who has `admin` role. The app redirects to `startapp.html`, which requires `admin` role.
2. Login with `user2`, who does not have `admin` role. The Flask-Principal raises `PermissionDenied`.

In the above example, if the permission test fails, a `PermissionDenied` is raised. You can tell Flask-Principal that you want to raise a specific HTTP error code instead:

```
@app.route("/startapp")
@login_required
@admin_permission.require(http_exception=403)
def startapp():
    ......
```

The 403 is the response status code for "No Permissions" or "Forbidden" (401 for "Unauthorized"). Now, `flask.abort(403)` will be called. You can also register an error handler for 403 as follows:

```
@app.errorhandler(403)
def unauthorized(e):
    session['redirected_from'] = request.url
    flash('You have no permissions to access this page')
        # Google "Flask message flashing fails across redirects" to fix, if any
    return redirect(url_for('login'))
```

Example 2: Granular Resource Protection

`fprin_eg2_models.py`: Add a model called `Post`, for the post written by users.

```
......
class Post(db.Model):
    post_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(30), db.ForeignKey('user.username'))
    data = db.Column(db.Text)

def load_db(db):
    ......
    db.session.add_all([
        Post(post_id=1, username='user1', data='This is post 1'),
        Post(post_id=2, username='user1', data='This is post 2'),
        Post(post_id=3, username='user2', data='This is post 3')])
    db.session.commit()
```

`flogin_eg1_forms.py`: same as previous example.

`fprin_eg2_controller.py`

```
# -*- coding: UTF-8 -*-
"""
fprin_eg2_controller: Flask-Principal Example - Granular Resource Protection
"""
from collections import namedtuple
from flask import Flask, render_template, redirect, flash, abort, url_for, session, request
from flask_login import LoginManager, login_user, logout_user, current_user, login_required
from flask_principal import Principal, Permission, Identity, AnonymousIdentity
from flask_principal import RoleNeed, UserNeed, identity_loaded, identity_changed
from fprin_eg2_models import db, User, Role, load_db, bcrypt   # Our models
from flogin_eg1_forms import LoginForm                  # Our web forms

# Flask: --- Setup ----------------------------------------------
app = Flask(__name__)

# Flask-WTF: ---  Setup -----------------------------------------
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Flask-WTF: needed for CSRF

# Flask-Bcrypt: ---  Setup ----------------------------------
bcrypt.init_app(app)

# Flask-SQLAlchemy: ---  Setup ----------------------------------
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
db.init_app(app)

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

# Flask-Login: ---  Setup ---------------------------------------
login_manager = LoginManager()
login_manager.init_app(app)

@login_manager.user_loader
def user_loader(username):
    """For Flask-Login. Given a username, return the associated User object, or None"""
    return User.query.get(username)

# Flask-Principal: ---  Setup ------------------------------------
principal = Principal()
principal.init_app(app)

# Flask-Principal: Create a permission with a single RoleNeed
admin_permission = Permission(RoleNeed('admin'))
```

```
PostNeed = namedtuple('PostNeed', ['action', 'post_id'])
    # e.g., PostNeed['get', '123'], PostNeed['update', '123'], PostNeed['delete', '123'],
    #       PostNeed['create'], PostNeed['list']

class PostPermission(Permission):
    """Extend Permission to take a post_id and action as arguments"""
    def __init__(self, action, post_id=None):
        need = PostNeed(action, post_id)
        super(PostPermission, self).__init__(need)

# Flask-Principal: Add the Needs that this user can satisfy
@identity_loaded.connect_via(app)
def on_identity_loaded(sender, identity):
    # Set the identity user object
    identity.user = current_user

    # Add the UserNeed to the identity (We are not using UserNeed here)
    if hasattr(current_user, 'username'):
        identity.provides.add(UserNeed(current_user.username))

    # The User model has a list of roles, update the identity with
    # the roles that the user provides
    if hasattr(current_user, 'roles'):
        for role in current_user.roles:
            identity.provides.add(RoleNeed(role.rolename))

    # The User model has a list of posts the user has authored,
    # add the needs to the identity
    if hasattr(current_user, 'posts'):
        identity.provides.add(PostNeed('list', None))
        identity.provides.add(PostNeed('create', None))
        for post in current_user.posts:
            identity.provides.add(PostNeed('get', post.post_id))
            identity.provides.add(PostNeed('update', post.post_id))
            identity.provides.add(PostNeed('delete', post.post_id))

# Flask: --- Routes ----------------------------------------------
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():  # POST request with valid data?
        # Retrieve POST parameters (alternately via request.form)
        _username = form.username.data
        _passwd = form.passwd.data
        # Query 'user' table with 'username'. Get first row only
        user = User.query.filter_by(username=_username).first()

        # Check if username presents?
        if user is None:
            flash('Username or Password is incorrect')  # to log incorrect username
            return redirect(url_for('login'))

        # Check if user is active? (We are not using Flask-Login inactive check!)
        if not user.active:
            flash('Your account is inactive')  # to log inactive user
            return redirect(url_for('login'))

        # Verify password
        if not user.verify_password(_passwd):
            flash('Username or Password is incorrect')  # to log incorrect password
            return redirect(url_for('login'))

        # Flask-Login: establish session for this user
        login_user(user)

        # Flask-Principal: Create an Identity object and
        #   signal that the identity has changed,
        #    which triggers on_identify_changed() and on_identify_loaded()
        # The Identity() takes a unique ID
        identity_changed.send(app, identity=Identity(user.username))

        return redirect(url_for('startapp'))

    # Initial GET request, and POST request with invalid data
    return render_template('flogin_eg1_login.html', form=form)

@app.route('/startapp')
@login_required              # Flask-Login: Check if user session exists
@admin_permission.require()  # Flask-Principal: Check if user having RoleNeed('admin')
def startapp():
    return render_template('flogin_eg1_startapp.html')

@app.route('/logout')
@login_required
def logout():
    # Flask-Login: Clear this user session
    logout_user()

    # Flask-Principal: Remove session keys
    for key in ('identity.name', 'identity.auth_type'):
        session.pop(key, None)

    # Flask-Principal: the user is now anonymous
    identity_changed.send(app, identity=AnonymousIdentity())

    return redirect(url_for('login'))

@app.route('/post/', methods=['GET', 'POST'])
def posts():
    if request.method == 'GET':
        permission = PostPermission('list')
        if not permission.can():
            abort(403)  # Forbidden
        return 'You can list all the posts!'

    if request.method == 'POST':
        permission = PostPermission('create')
        if not permission.can():
            abort(403)  # Forbidden
        return 'You can create new post!'

    abort(405)  # method not supported

@app.route('/post/<post_id>', methods=['GET', 'PUT', 'DELETE'])
def post(post_id):
    if request.method == 'GET':
        permission = PostPermission('get', post_id)
        if not permission.can():
            abort(403)  # Forbidden
        return 'You can read this post!'

    if request.method == 'POST':
        permission = PostPermission('update', post_id)
        if not permission.can():
            abort(403)  # Forbidden
        return "You can edit this post!"

    if request.method == 'DELETE':
        permission = PostPermission('delete', post_id)
        if not permission.can():
            abort(403)  # Forbidden
        return "You can delete this post!"

    abort(405)  # method not supported

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

`/templates`: Same as previous example.

1. We define our custom `Need` called `PostNeed`, which takes two parameters, `action` and an optional `post_id`. For example, `PostNeed['get', '123']`, `PostNeed['update', '123']`, `PostNeed['delete', '123']`, `PostNeed['create']`, `PostNeed['list']`, followed the conventions used in RESTful API.
2. We also define our custom `Permission` called `PostPermission`, which contains a `PostNeed`, which also takes two arguments, `action` and an optional `post_id`.
3. In `on_identity_loaded`, we added the `PostNeed` that the user can provide.
4. We use two routes: `/post/` for get-all and create-new, `/post/<post-id>` for get, update, and delete for a particular post, followed the conventions in RESTful API.
5. Inside the view function, we create the required permission and verify if the current user can provide via `can()` method, before processing the request.

10.4 Flask-Admin Extension

Reference: Flask-Admin @ https://flask-admin.readthedocs.org/en/latest/.

Flask-Admin helps you build an admin interface on top of an existing data model, and lets you manage your web service's data through a user-friendly interface.

Install Flask-Admin Extension

```
# Activate your virtual environment
(venv)$ pip install flask-admin
Successfully installed flask-admin-1.4.2

(venv)$ pip show flask-admin
Name: Flask-Admin
Version: 1.4.2
Location: .../venv/lib/python3.5/site-packages
Requires: Flask, wtforms
```

Example: Using Flask-Admin

`fprin_eg1_models.py`: same as Flask-Principal example.

`flogin_eg1_forms.py`: same as Flask-Principal example.

`fadmin_eg1_controller.py`

```
# -*- coding: UTF-8 -*-
"""
fadmin_eg1_controller: Flask-Admin - Testing Flask-Admin
"""
from flask import Flask, render_template, redirect, flash, abort, request, url_for, session
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView
from flask_login import LoginManager, login_user, logout_user, current_user, login_required
from flask_principal import Principal, Permission, Identity, AnonymousIdentity
from flask_principal import RoleNeed, UserNeed, identity_loaded, identity_changed
from fprin_eg1_models import db, User, Role, load_db, bcrypt   # Our models
from flogin_eg1_forms import LoginForm                  # Our web forms

# Flask: --- Setup ----------------------------------------------
app = Flask(__name__)

# Flask-WTF: ---  Setup -----------------------------------------
app.config['SECRET_KEY'] = 'YOUR-SECRET'  # Flask-WTF: needed for CSRF

# Flask-Bcrypt: ---  Setup ----------------------------------
bcrypt.init_app(app)

# Flask-SQLAlchemy: ---  Setup ----------------------------------
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://testuser:xxxx@localhost:3306/testdb'
db.init_app(app)

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

# Flask-Login: ---  Setup ---------------------------------------
login_manager = LoginManager()
login_manager.init_app(app)

@login_manager.user_loader
def user_loader(username):
    """For Flask-Login. Given a username, return the associated User object, or None"""
    return User.query.get(username)

# Flask-Principal: ---  Setup ------------------------------------
principal = Principal()
```

```
principal.init_app(app)

# Flask-Principal: Create a permission with a single RoleNeed
admin_permission = Permission(RoleNeed('admin'))

# Flask-Principal: Add the Needs that this user can satisfy
@identity_loaded.connect_via(app)
def on_identity_loaded(sender, identity):
    # Set the identity user object
    identity.user = current_user

    # Add the UserNeed to the identity
    if hasattr(current_user, 'username'):
        identity.provides.add(UserNeed(current_user.username))

    # Assuming the User model has a list of roles, update the
    # identity with the roles that the user provides
    if hasattr(current_user, 'roles'):
        for role in current_user.roles:
            identity.provides.add(RoleNeed(role.rolename))

# Flask-Admin: ---  Setup ---------------------------------------
admin = Admin(name='Admin')
admin.add_view(ModelView(User, db.session, category='Database'))
admin.add_view(ModelView(Role, db.session, category='Database'))
admin.init_app(app)  # Bind Flask-Admin to this Flask app

# Flask: --- Routes ---------------------------------------------
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():  # POST request with valid data?
        # Retrieve POST parameters (alternately via request.form)
        username = form.username.data
        passwd = form.passwd.data
        # Query 'user' table with 'username'. Get first row only
        user = User.query.filter_by(username=username).first()

        # Check if username presents?
        if user is None:
            flash('Username or Password is incorrect')  # to log incorrect username
            return redirect(url_for('login'))
        # Check if user is active? (We are not using Flask-Login inactive check!)
        if not user.active:
            flash('Your account is inactive')  # to log inactive user
            return redirect(url_for('login'))

        # Verify password
        if not user.verify_password(passwd):
            flash('Username or Password is incorrect')  # to log incorrect password
            return redirect(url_for('login'))

        # Flask-Login: establish session for this user
        login_user(user)

        # Flask-Principal: signal that the identity has changed.
        #   Trigger on_identify_changed() and on_identify_loaded()
        # The Identity() takes a unique ID
        identity_changed.send(app, identity=Identity(user.username))

        return redirect(url_for('startapp'))

    # Initial GET request, and POST request with invalid data
    return render_template('flogin_eg1_login.html', form=form)

@app.route('/startapp')
@login_required              # Flask-Login: Check if user session exists
@admin_permission.require(http_exception=403)  # Flask-Principal: Check if user having RoleNeed('admin')
def startapp():
    return render_template('flogin_eg1_startapp.html')

@app.route('/logout')
@login_required
def logout():
    # Flask-Login: Clear this user session
    logout_user()

    # Flask-Principal: Remove session keys
    for key in ('identity.name', 'identity.auth_type'):
        session.pop(key, None)

    # Flask-Principal: the user is now anonymous
    identity_changed.send(app, identity=AnonymousIdentity())

    return redirect(url_for('login'))
```

```
if __name__ == '__main__':
    # Debug only if running through command line
    app.config['SQLALCHEMY_ECHO'] = True
    app.debug = True
    app.run()
```

`templates`: same as Flask-Principal example.

NOTES: Your need to drop the `'post'` table, before running this code.

Try:

1. Login as `user1` (who has `admin` role), and switch URL to `http://127.0.0.1/admin`. Test out the admin pages, which can CRUD our data models.

How It Works

1. We only added these 4 lines to enable the Flask-Admin (together with the necessary imports)!!!

   ```
   admin = Admin(name='Admin')
   admin.add_view(ModelView(User, db.session, category='Database'))
   admin.add_view(ModelView(Role, db.session, category='Database'))
   admin.init_app(app)
   ```

2. Line 1 constructs an `Admin` object, with a `name` shown as the header for the admin pages. Line 4 binds the Flask-Admin to the Flask app.
3. [TODO] Line 2 and 3

Apply Permissions to Admin Pages

In the previous example, the Admin pages are not protection via permissions. Try login with `user2`, who has no `admin` role, and switch the URL to `http://127.0.0.1:5000/admin`.

To protect the admin pages, add the followings into the controller:

```
from flask import g
from flask_admin import expose, AdminIndexView
......
......
# Flask-Admin: Initialize
class AuthMixinView(object):
    def is_accessible(self):
        has_auth = current_user.is_authenticated
        has_perm = admin_permission.allows(g.identity)
        return has_auth and has_perm

class AuthModelView(AuthMixinView, ModelView):
    @expose()
    @login_required
```

```
    def index_view(self):
        return super(ModelView, self).index_view()

class AuthAdminIndexView(AuthMixinView, AdminIndexView):
    @expose()
    @login_required
    def index_view(self):
        return super(AdminIndexView, self).index_view()

admin = Admin(name='Admin', index_view=AuthAdminIndexView())
```

Now, only users with `admin` role can access the admin pages.

How It Works

1. We extend both `ModelView` and `AdminIndexView` in a design pattern called `mixin`. We override the `is_accessible()` method, so that users without permission will receive a 403; and overwrite the `index_ view()` for `AdminIndexView` and `ModelView`, adding the `login_required` decorator.
2. [TODO] customization on CRUD.

[TODO] MORE

10.5 Flask-Mail Extension

Reference: Flask-Mail @ https://pythonhosted.org/Flask-Mail/.

The Flask-Mail extension provides a simple interface to set up SMTP inside Flask app, and to send emails from your views and scripts.

Installing Flask-Mail

```
# Activate your virtual environment
(venv)$ pip install Flask-Mail
Successfully installed Flask-Mail-0.9.1

(venv)$ pip show flask-mail
Name: Flask-Mail
Version: 0.9.1
Location: .../venv/lib/python3.5/site-packages
Requires: Flask, blinker
```

Example: Using Flask-Mail Extension to Send Mail in Flask App

```
from flask import Flask
from flask_mail import Mail, Message
```

```
app = Flask(__name__)

mail = Mail()
mail.init_app(app)
    # Or: mail = Mail(app)

@app.route('/')
def main():
    msg = Message("Hello", sender="from@example.com", recipients=["to@example.com"])
        # Create a Message instance with a 'subject'
        # You can set the MAIL_DEFAULT_SENDER and omit 'sender'
        # The 'recipients' is a list
    msg.body = "testing"
        # Add body
    mail.send(msg)
        # Send
    return "Email Sent"

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

These configuration options (to be set in `app.config`) are available:

- `MAIL_SERVER`: default `'localhost'`
- `MAIL_PORT`: default `25`
- `MAIL_DEFAULT_SENDER`: default `None`
- more

11. Structuring Large Application with Blueprints

As you app grows, Flask Blueprints allow you to modularize, structure and organize you large project.

Python Modules, Packages and Import - Recap

Recall that a Python module is any `'.py'` file. You can import a module, or an attribute inside a module.

A Python package is a directory containing an `__init__.py` file. This `__init__.py` is analogous to the object-constructor, i.e., whenever a package is imported, its `__init__.py` will be executed. Similarly, you can import a package, a module inside a package, an attribute inside a module of a package.

[TODO] Relative import

Example 1: Using Blueprints

A blueprint defines a collection of views, templates, static files and other resources that can be applied to an application. It allows us to organize our application into components.

`bp_component1.py`

```
"""
bp_component1: An application component
"""
from flask import Blueprint, render_template

# Construct a blueprint for component1, given its name and import-name
# We also specify its template and static directories
component1 = Blueprint('c1', __name__,
      template_folder='templates', static_folder='static')

@component1.route('/')
@component1.route('/<name>')
def index(name=None):
    if not name:
        name = 'world'

    return render_template('bp_index.html', name=name)
```

1. We construct a `Blueprint` object, and set its `name` to `c1`. The functions such as `main()` will be referenced as `c1.main()`.
2. A `Blueprint` can have its own `templates` and `static` directories. In the above example, we set them to their default.

`bp_controller.py`

```
"""
bp_controller: main entry point
"""
from flask import Flask
from bp_component1 import component1

app = Flask(__name__)
app.register_blueprint(component1)  # Register a Blueprint

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

`templates/bp_index.html`

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

You can now access the app via `http://localhost:5000` or `http://localhost:5000/<name>`

You can also specify a `url_prefix` to all the routes in a blueprint, using keyword argument `url_prefix` in `register_blueprint()`, e.g.,

```
app.register_blueprint(component1, url_prefix='/c1path')
```

Now, you access the app via `http://localhost:5000/c1path` or `http://localhost:5000/c1path/<name>`

Organizing Large App using Blueprints

```
myapp
    |-- run.py            # or start.py to run the flask app
    |-- config.ini        # or config.py or /config folder
    |__ /env              # Virtual Environment
    |__ /test             # unit tests and acceptance tests
    |__ /app_main         # Our App and its Components
         |-- __init__.py
         |-- models.py    # Data Models for all components
         |-- /component1  # or application module
             |-- __init__.py
             |-- controllers.py
             |-- forms.py
         |-- /component2
             |-- __init__.py
             |-- controllers.py
             |-- forms.py
         |......
         |......
         |__ /templates
             |-- 404.html   # Also 401, 403, 500, 503
             |-- base.html  # Base templates to be extended
             |__ /component1
                 |-- extended.html
         |__ /static
             |-- css/
             |-- js/
             |-- images/
```

1. The `run.py` (or `start.py`) starts the Flask app.
2. The `config.ini` keeps all the configuration parameters organized in sections such as `[app]`, `[db]`, `[email]`, etc., to be parsed by `ConfigParser`. [After Note] It is a lot more convenience to use a Python module for configuration!
3. The common HTTP error codes are:
    - 400 Bad Request
    - 401 Unauthorized (wrong username/password, inactive)
    - 403 Forbidden or No Permission (wrong role, no access permission)
    - 404 Not Found
    - 405 Method Not Allowed
    - 500 Internal Server Error
    - 503 Service Unavailable
4. Some designers prefer to keep the `templates` and `static` folders under each component.

Example

Follow the above structure, with a component called `mod_auth` for User authentication (the same example from Flask-Login).

`run.py`

```
# -*- coding: UTF-8 -*-
"""(run.py) Start Flask built-in development web server"""
from app_main import app  # from package app_main's __init__.py import app

if __name__ == '__main__':
   # for local loopback only, on dev machine
   app.run(host='127.0.0.1', port=5000, debug=True)

   # listening on all IPs, on server
   #app.run(host='0.0.0.0', port=8080, debug=True)
```

`config.ini`

```
# -*- coding: UTF-8 -*-
# (config.ini) Configuration Parameters for the app

[app]
DEBUG: True
SECRET_KEY: YOUR-SECRET

[db]
SQLALCHEMY_DATABASE_URI: mysql://testuser:xxxx@localhost:3306/testdb

[DEFAULT]
```

1. We shall use `ConfigParser` to parse this configuration file.
2. The format is "`key: value`".
3. After Note: More convenience to use a Python module for config, and load via `from_object()`, instead of `.ini` file.

`app_main/models.py`

```
# -*- coding: UTF-8 -*-
"""(models.py) Data Models for the app, using Flask-SQLAlchemy"""
from flask_sqlalchemy import SQLAlchemy
from flask_login import UserMixin
from passlib.hash import bcrypt  # For hashing password

# Flask-SQLAlchemy: Initialize
db = SQLAlchemy()

# Define a 'Base' model for other database tables to inherit
class Base(db.Model):
   __abstract__  = True
   date_created  = db.Column(db.DateTime,
         default=db.func.current_timestamp())
   date_modified = db.Column(db.DateTime,
         default=db.func.current_timestamp(),
         onupdate=db.func.current_timestamp())

# Define a 'User' model mapped to table 'user' (default to lowercase).
# Flask-Login: Use default implementations in UserMixin
class User(Base, UserMixin):
   username = db.Column(db.String(30), primary_key=True)
   pwhash   = db.Column(db.String(300), nullable=False)  # Store password hash
   active   = db.Column(db.Boolean, nullable=False, default=False)
   def __init__(self, username, password, active=False):
      """Constructor: to hash password"""
      self.pwhash = bcrypt.encrypt(password)  # hash submitted password
      self.username = username
      self.active = active

   @property   # Convert from method to property (instance variable)
   def is_active(self):
      """Flask-Login: return True if the user is active."""
      return self.active

   def get_id(self):  # This is method. No @property
      """Flask-Login: return a unique ID in unicode."""
      return unicode(self.username)

   def verify_password(self, in_password):
      """Verify the given password with the stored password hash"""
      return bcrypt.verify(in_password, self.pwhash)

def load_db(db):
   """Create database tables and records"""
   db.drop_all()
   db.create_all()
   db.session.add_all(
         [User('user1', 'xxxx', True),
          User('user2', 'yyyy')])  # inactive
   db.session.commit()
```

`app_main/mod_auth/__init__.py`: an empty file

`app_main/mod_auth/forms.py`

```
# -*- coding: UTF-8 -*-
"""(forms.py) Web Forms for this module"""
from flask_wtf import FlaskForm  # import from flask_wtf, NOT wtforms
from wtforms import StringField, PasswordField
from wtforms.validators import InputRequired, Length

# Define the LoginRorm class by sub-classing Form
class LoginForm(FlaskForm):
   # This form contains two fields with validators
   username = StringField(u'User Name:', validators=[InputRequired(), Length(max=20)])
   passwd = PasswordField(u'Password:', validators=[Length(min=4, max=16)])
```

app_main/mod_auth/controllers.py

```
# -*- coding: UTF-8 -*-
"""(controllers.py) Using Flask-Login for User Session Management"""
from flask import Blueprint, render_template, redirect, flash, abort, url_for
from flask_login import LoginManager, login_user, logout_user, current_user, login_required
from ..models import User     # Our models
from .forms import LoginForm  # Our web forms

# Define a blueprint for this module ----------------------------
# All function endpoints are prefixed with "auth", i.e., auth.fn_name
mod_auth = Blueprint('auth', __name__,
      template_folder='../templates/mod_auth',
      static_folder='../static')

# Flask-Login ----------------------------------------------------
login_manager = LoginManager()

@login_manager.user_loader
def user_loader(username):
   """Flask-Login: Given an ID, return the associated User object, or None"""
   return User.query.get(username)

# Define routes for this module -----------------------------------
@mod_auth.route("/login", methods=['GET', 'POST'])
def login():
   form = LoginForm()  # Construct a LoginForm

   if form.validate_on_submit():  # POST request with valid data?
      # Retrieve POST parameters (alternately via request.form)
      username = form.username.data
      passwd = form.passwd.data
      # Query 'user' table with 'username'. Get first row only
      user = User.query.filter_by(username=username).first()

      # Check if username presents?
      if user is None:
         flash(u'Username or Password is incorrect')  # to log incorrect username
         return redirect(url_for('.login'))  # Prefix endpoint with this blueprint name

      # Check if user is active? (We are not using Flask-Login inactive check!)
      if not user.active:
         flash(u'Your account is inactive')  # to log inactive user
         return redirect(url_for('.login'))
      # Verify password
      if not user.verify_password(passwd):
         flash(u'Username or Password is incorrect')  # to log incorrect password
         return redirect(url_for('.login'))

      # Flask-Login: establish session for this user
      login_user(user)
      return redirect(url_for('.startapp'))

   # Initial GET request, and POST request with invalid data
   return render_template('login.html', form=form)

@mod_auth.route("/startapp")
@login_required  # Flask-Login: Check if user session exists
def startapp():
   return render_template('startapp.html')

@mod_auth.route("/logout")
@login_required
def logout():
   # Flask-Login: clear this user session
   logout_user()
   return redirect(url_for('.login'))
```

1. We construct a `Blueprint` for this webapp component called `mod_auth`. The first argument specifies the prefix of the endpoint of the functions in the blueprint. For example, function `login()` is now `auth.login()`. We also specify the templates and static folder, relative to this module.
2. In `url_for()`, you can refer to function as `auth.login` or simply `.login` for functions in this module.

app_main/__init__.py

```
# -*- coding: UTF-8 -*-
"""(__init__.py) app_main package initialization file"""
from flask import Flask, render_template

# Configurations --------------------------------------
import ConfigParser
config = ConfigParser.SafeConfigParser()
config.read('myapp.ini')  # Relative to run.py

# Define the WSGI application object ------------------
app = Flask(__name__)
app.config['SECRET_KEY'] = config.get('app', 'SECRET_KEY')
```

```
# Flask-SQLAlchemy --------------------------------------
from flask_sqlalchemy import SQLAlchemy
from models import db, load_db

app.config['SQLALCHEMY_DATABASE_URI'] = config.get('db', 'SQLALCHEMY_DATABASE_URI')
# Bind SQLAlchemy to this Flask app
db.init_app(app)

# Build the database
with app.test_request_context():
   load_db(db)

# Error Handlers ---------------------------------------
@app.errorhandler(404)
def not_found(error):
    return render_template('404.html'), 404

# Blueprints -------------------------------------------
# Import mod_auth and register blueprint
from app.mod_auth.controllers import mod_auth, login_manager
app.register_blueprint(mod_auth, url_prefix='/auth')
      # URL is /auth/login

# Initialize Flask-Login for this module
login_manager.init_app(app)
```

app_main/templates/base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Application</title>
  <meta charset="utf-8">
</head>
<body>
{# The body-section contents here #}
{% block body_section %}{% endblock %}
</body>
</html>
```

app_main/templates/base_flash.html

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

```
{# Display flash messages, if any, for all pages #}
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class='flashes'>
    {% for message in messages %}<li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}

{# The body contents here #}
{% block body %}{% endblock %}
{% endblock %}
</body>
</html>
```

app_main/templates/404.html

```
{% extends "base.html" %}
{% block body_section %}
<h1>Sorry, I can't find the item</h1>
{% endblock %}
```

app_main/templates/mod_auth/login.html

```
{% extends "base_flash.html" %}{# in app_main's templates folder #}
{% block body %}
<h1>Login</h1>
<form method="POST">
  {{ form.hidden_tag() }} {# Renders any hidden fields, including the CSRF #}
  {% for field in form if field.widget.input_type != 'hidden' %}
    <div class="field">
      {{ field.label }} {{ field }}
    </div>
    {% if field.errors %}
      {% for error in field.errors %}
        <div class="field_error">{{ error }}</div>
      {% endfor %}
    {% endif %}
  {% endfor %}
  <input type="submit" value="Go">
</form>
{% endblock %}
```

1. Flask finds the base template from app_main's templates folder and then in blueprint's templates folder.

`app_main/templates/mod_auth/startapp.html`

```
{% extends "base_flash.html" %}{# in app_main's templates folder #}
{% block body %}

{% if current_user.is_authenticated %}
  <h1>Hello, {{ current_user.username }}!</h1>
{% endif %}

<a href="{{ url_for('.logout') }}">LOGOUT</a>
{% endblock %}
```

12. Deploying Your Flask Webapp

12.1 On Apache Web Server with '`mod_wsgi`'

Reference: Deployment with mod_wsgi (Apache) @ http://flask.pocoo.org/docs/0.10/deploying/mod_wsgi/.

Step 1: Install and Enable Apache Module `mod_wsgi`

Firstly, check if Apache module `mod_wsgi` is installed. Goto `/etc/apache2/mods-available` and look for `wsgi.conf` to check if it is installed. If `mod_wsgi` is not installed:

```
# For Python 2
$ sudo apt-get install libapache2-mod-wsgi
......
Setting up libapache2-mod-wsgi (3.4-4ubuntu2.1.14.04.2) ...
apache2_invoke: Enable module wsgi

# For Python 3
$ sudo apt-get install libapache2-mod-wsgi-py3
......
Setting up libapache2-mod-wsgi-py3 (4.3.0-1.1build1) ...
apache2_invoke: Enable module wsgi

# Verify installation
$ ll /etc/apache2/mods-available/wsgi*
-rw-r--r-- 1 root root 5055 Nov 19  2014 /etc/apache2/mods-available/wsgi.conf
-rw-r--r-- 1 root root   60 Nov 19  2014 /etc/apache2/mods-available/wsgi.load
$ ll /etc/apache2/mods-enabled/wsgi*
```

```
lrwxrwxrwx 1 root root 27 Nov 17 16:55 /etc/apache2/mods-enabled/wsgi.conf -> ../mods-available/wsgi.conf
lrwxrwxrwx 1 root root 27 Nov 17 16:55 /etc/apache2/mods-enabled/wsgi.load -> ../mods-available/wsgi.load
$ dpkg --list libapache2-mod-wsgi
||/ Name           Version      Architecture Description
+++-==============-============-============-=================================
ii  libapache2-mod 3.4-4ubuntu2 amd64        Python WSGI adapter module for Ap
```

If mod_wsgi is installed but not enabled (wsgi.conf is present in /etc/apache2/mods-available/ but not in /etc/apache2/mods-enabled/), you can enable the module via a2enmod utility:

```
$ sudo a2enmod wsgi
$ sudo service apache2 reload
```

Step 2: Write your Python-Flask Webapp

As an example, let me write a hello-world Python-Flask webapp. Suppose that our document base directory is /var/www/myflaskapp.

As an illustration,

1. We shall create a package called mypack, by creating a sub-directory mypack under the project directory.
2. Write the package init module __init__.py (at /var/www/myflaskapp/mypack) as follows:

```
# -*- coding: UTF-8 -*-
from flask import Flask, render_template
app = Flask(__name__)
app.debug = True   # Not for production

@app.route('/')
def hello():
    return render_template('index.html', username='Peter')
```

3. Write the template index.html (at /var/www/myflaskapp/mypack/templates) as follows:

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

Notes: The sub-directory `template` are to be placed inside the package `mypack`.

Step 3: No `app.run()`

Take note that the `app.run()` must be removed from `if __name__ == '__main__':` block, so as not to launch the Flask built-in developmental web server.

Step 4: Creating a `.wsgi` file

Create a `.wsgi` file called `start.wsgi`, under the project directory, as follows. This file contains the codes `mod_wsgi` is executing on startup to get the Flask application instance.

```
import sys, os

# To include the application's path in the Python search path
sys.path.insert(1, os.path.dirname(__file__))
# print(sys.path)

# Construct a Flask instance "app" via package mypack's __init__.py
from mypack import app as application
```

Step 5: Configuring Apache

Create an Apache configuration file for our webapp `/etc/apache2/sites-available/myflaskapp.conf`, as follows:

```
# This directive must be place outside VirtualHost
WSGISocketPrefix /var/run/wsgi

# If you use virtual environment
# Path for 'python' executable
#WSGIPythonHome /home/username/virtualenv/bin
# Path for 'python' additional libraries
#WSGIPythonPath /home/username/virtualenv/lib/python3.5/site-packages

<VirtualHost *:8000>
    WSGIDaemonProcess myflaskapp user=flaskuser group=www-data threads=5
    WSGIScriptAlias / /var/www/myflaskapp/start.wsgi

    <Directory /var/www/myflaskapp>
        WSGIProcessGroup myflaskapp
        WSGIApplicationGroup %{GLOBAL}
        # For Apache 2.4
        Require all granted
        # For Apache 2.2
        # Order deny,allow
        # Allow from all
    </Directory>
</VirtualHost>
```

Notes:

1. An alias is defined such that the root URL / is mapped to `start.wsgi` to start the Python-Flask app.
2. The `WSGISocketPrefix` directive is added to resolve this error:

   ```
   (13)Permission denied: [client 127.0.0.1:39863] mod_wsgi (pid=29035): Unable to
   connect to WSGI daemon process 'myflaskapp' on '/var/run/apache2/wsgi.18612.1.1.sock'
   after multiple attempts.
   ```

   The WSGI process (flaskuser:www-data) does not have write permission on `/var/run/apache2/` for writing sockets. We change it to `/var/run`.

   This directive must be placed outside `<VirtualHost>`.
3. [TODO]

Next, edit `/etc/apache2/ports.conf` to add "`Listen 8000`".

In the above configuration, we run the application under a special non-login system user (called `flaskuser`) for security reasons. You can create this non-login user as follows:

```
$ sudo adduser --system --group --disabled-login flaskuser
Adding system user `flaskuser' (UID 119) ...
Adding new group `flaskuser' (GID 127) ...
Adding new user `flaskuser' (UID 119) with group `flaskuser' ...
Creating home directory `/home/flaskuser' ...
```

[TO CHECK] Home directory is needed for this user to run wsgi application?! A login user needs a home directory to set as current working directory.

This user `flaskuser` and Apache's user `www-data` shall have permissions to all the resources. We make `flaskuser` as the user-owner and `www-data` as the group-owner with at least `'r'` permission for files and `'r-x'` for directories. `'w'` is needed for `flaskuser` too [TO CHECK].

```
$ cd /var/www/myflaskapp
$ sudo chown -R flaskuser:www-data .
$ sudo chmod -R 750 .
```

Enable our web site and reload the Apache:

```
$ sudo a2ensite myflaskapp
$ sudo service apache2 reload
```

Step 6: Run

You can now access the webapp via `http://localhost:8000`.

Check the `/var/log/Apache2/error.log`, in case of error such as `50x`.

Debugging - IMPORTANT

Read "Debugging Python-Flask Webapp Remotely" under Eclipse PyDev Section.

12.2 On Cloud

[TODO]

13. Flask with AngularJS

[TODO]

REFERENCES & RESOURCES

1. [TODO]

Latest version tested: Python (Ubuntu) 3.5.2, Flask 0.12
Last modified: March, 2017