Flask/Python Tips

Flask is a lightweight but powerful Web development framework based on Python. This document describes how to set up and run a simple Flask website from your course container. A similar setup should suffice for your course project and deployment to a limited user population. (Running Flask in a real production setting would be very different; follow these instructions if you want to explore although it is not needed for this course.)

We provide an example Flask application for our beloved `beers` database that you used in your homework. We assume that you already have this database set up; if not, run the following in your container shell:

```
$DBCOURSE/examples/db-beers/setup.sh
```

Setting Up a Flask Web App

First, refresh your course materials by running "`$DBCOURSE/sync.sh`".

Pick a folder for your Flask application, e.g., `~/shared/flask-beers` (feel free to use a different destination as you see fit; for example, if you have a Duke OIT container, putting the folder under `~/shared/` wouldn't be helpful). Hereafter we will refer to this folder as your "Flask app directory." Copy the code for the app into your Flask app directory:

```
cp -r $DBCOURSE/examples/flask-beers ~/shared/
```

After copying the code, you need to prepare its environment. First, as with any non-trivial Python project, you will need a virtual environment management tool. We use `poetry` (already installed in your container). This tool installs and tracks your library dependencies on a per-project basis. We will save the `poetry` tutorial for another day, but for now, you just need to run the following commands:

```
cd ~/shared/flask-beers/ # replace with your directory as appropriate
poetry config virtualenvs.in-project true
poetry install
```

You are now ready to go!

Running the Flask Web App

To run the website, go to your Flask app directory, and issue the following commands:

```
poetry shell
python app.py
```

The first command ensures that you are in the correct Python virtual environment (you can tell that your command-line prompt looks differently — it would start with the name of the

environment). The second command runs the Flask/web server. You can now use your laptop's browser to explore the website. Depending on your setup, the URL will be different:

- If you use containers on your own laptop, point your browser to http://localhost:8080/
- If you use the Duke OIT container, use the URL for "Flask app" shown in the CONTAINER CONTROLS info pane.

You can keep your Flask server running and edit your app code on the side. In most cases, your edits will be automatically reflected. If your changes introduce some error (such as a syntax error in your code), it may cause the server to exit with an error; you can note the error reported in the container shell, fix it, and restart by running "`python app.py`" again. Sometimes you will see an error reported by the server in your browser (while the server continues to run); in that case you can do some limited form of debugging directly from your browser (you can find the debugger PIN from the output in the container shell).

To stop your app, type `CTRL-c` in the container shell; that will take you back to the command-line prompt, still within the `poetry` environment. If you are all done with this app for now, you should type `exit` to get out of the `poetry` environment and get back to the normal container shell.

A Flask Primer

Flask has following features that simplify Web development:

- A MVC (model-view-controller) architecture, which separates data, presentation, and control flow.
    - The model "wraps" data stored in the database as Python objects, which can be manipulated (created, queried, and updated) by the controller and presented through the views.
    - The controller defines a set of views. Each view handles a Web request. It can interact with the database through the model, and feed relevant data to a Web page to be shown.
    - The Web pages are rendered in a Python-based template language called Jinja2. Templates can do some simple post-processing with the content to facilitate presentation (such as sorting and looping through a list of items). There is typically one template for each type of Web page. Flask templates can be thought of as functions, and you can define a "base" template to factor out common structures/elements on your pages so it can be reused by other templates.
- Mapping between URLs and views. Flask allows you to map URLs to views, which will take parameters passed in through the URL request as their input arguments. Flask also allows you to map views back to URLs. With the latter, your template code can generate links with parameters to other pages by referring to the appropriate views, without worrying about how to write the actual URLs.
- Automatic mapping of object models to database tables using SQLAlchemy. You can specify models using Python classes; SQLAlchemy automatically handles their storage,

retrieval, and update in a relational database. You can do (almost) all data manipulation on your Python objects without being concerned about SQL. However, we will not use the full capability of SQLAlchemy in our example app.

We now explain the relevant parts of a Flask app directory:

- `config.py`: This configuration file stores (among other things) your database connection information. For this example, we connect to the PostgreSQL database named beers, which you used extensively in your assignments. You can change the connection information to work with other databases instead.
- `app.py`: This is the main file that sets up the app and defines all the views and what URLs map to them (using `@app.route` decorators). Most views return with a call to `render_template`, which passes a bunch of data objects to a template file. `app.py` can also define additional Jinja2 "filters" that you can use in your templates (using `@app.template_filter` decorators).
- `models.py`: This file defines how data stored in the database can be mapped to Python objects. With these definitions, SQLAlchemy can automatically retrieve data from the database as Python objects. For example, the view `drinker` in `app.py` retrieves a given drinker and associated information using just Python, without any SQL. For more complex queries and updates, however, SQLAlchemy can be quite convoluted and error-prone; you might be better off writing SQL instead. For example, we code `Drinker.edit` using SQL. Note the use of `:param` as placeholders for parameters to be passed into SQL; this method of constructing SQL queries is safer than trying to concatenate strings to make SQL queries yourself.
- `forms.py`: This file defines forms objects on your website. Writing HTML forms manually can be painful. Luckily, Flask offers `Flask-WTF` to simplify this process: you can specify fields and how to validate them declaratively. Things become trickier if you want to generate your form structure dynamically (e.g., from a database); an example can be found in `DrinkerEditFormFactory` in this file (it is used in the `edit_drinker` view in `app.py` and the `edit_drinker.html` template).
- `templates/`: This directory contains all your templates. Here, all templates "extends" `layout.html`, which controls the overall look of the website. Note the use of `url_for` to link to another page instead of hardcoding the URL.
- `static/`: This directory contains your static files, i.e., files such as images, CSS, and JavaScript that don't need to be generated dynamically at runtime.

Rolling Your Own App

You can just modify the example Flask app directory (`~/shared/flask-beers`). You can change the directory name to, for example, `flask-mycoolapp`.

Remember to modify the database configuration in `config.py` as needed. You might also want to update the `[tool.poetry]` section of the `poetry` configuration file `pyproject.toml`.

If your app requires a new Python library (say `new-library`), don't use the traditional `pip install`; instead, run "`poetry add new-library`", and `poetry` will install it into your project-specific virtual environment and automatically update `pyproject.toml`.