Week 13 - Web Development with Flask (2/2)

Welcome to Week 13! Last week we started learning about Flask, a web development framework in Python. We reviewed the basic concepts that underlie what a framework does, and built a small example of how it all fits together. This week, we will expand on that, digging a little deeper into Models and how we can integrate a relational database into our web applications. We will continue to use SQLite as our database backend. We will also learn some techniques to help debug your Flask applications, which could come in handy when working on your class project. First, we will take a quick detour and talk about sessions.

Sessions and HTTP Cookies

In Week 9, when we reviewed HTTP, it was stated in the reading that "[HTTP] assumes very little about a particular system, and does not keep state between different message exchanges". What does 'not keeping state' mean exactly? In computer science, "the state of computer program is a technical term for all the stored information, at a given instant in time, to which the program has access" [1] . This means that, when a browser makes a HTTP request to some server, the server will process the request as if it ***never seen*** a request from this specific browser before. The server process, in our case Flask, simply does not remember the client from request to request.

However, how could we support a web site that needs to know if a user is logged in? If for every request, the server acts like it has never seen this client before, how could we ever maintain this 'state' of the client? HTTP, like most of the technology in the web stack, has evolved over time to support this, and this is done using what is known as a ***cookie***. A Cookie is a special HTTP header sent in requests to identify the client. You may have heard of cookies before, mostly due to privacy concerns over how cookies can potentially track you as you browse the web. However, cookies are a critical part of the web now, so how do they work?

When a browser makes its first request to some website, lets say Google.com, it will make the request with no cookie. The server, seeing that there is no cookie value in the request, will tell the client in the response that from now on, when you make requests to this server, add a Cookie header (which is done via the Set-Cookie header in the response). The server is essentially sending back a unique identifier to the client so that from now on, the server will know who is making this request. This cookie value is part of the ***state information*** that the server needs to keep track of now.

Luckily for us, Flask takes care of most of the heavy lifting when dealing with cookies by using a feature called ***sessions***. Flask gives us access to a global object (a dictionary) called `session`, which allows us to save information specifically for the client who made the request.

---

[1] http://en.wikipedia.org/wiki/State_(computer_science)

Flask takes care of loading this information into the session object when the request comes in. To see an example of its usage, check the Session section in the Flask Quickstart guide. The code there will show you how to support basic logins for a web application. Also read this section on using sessions in last week's Flask Tutorial reading.

You may be wondering where this information ends up being stored by Flask. For now, the session information is stored in memory, which means this information is lost when Flask is restarted. Flask, however, does support saving session information into a more persistent data store, like a SQL database.

Logging

As you start to build larger applications with Flask, it becomes more difficult to track down errors in your logic. One method to help with debugging is to have a log file that you print messages to. If you remember back in Week 2 where we reviewed some useful Python modules, we learned about the logging module. We can easily use this module in Flask to log important errors or messages. To do so, we simply need to import the module and instantiate a logger object, and attach it to the main Flask app object:

```
import logging
from logging import FileHandler
file_handler = FileHandler("/path/to/logfile.log")
file_handler.setLevel(logging.INFO)
app.logger.addHandler(file_handler)
```

Now, whenever we want to print something to the log in any of our functions, we just need to access the application object and use the logger like we normally would:

```
app.logger.error("This is a custom error message!")
```

To read more about how to handle errors, including how to setup an email handler, check the Error Handling guide from the Flask documentation.

Debug Mode

To help facilitate debugging of errors and exceptions that may happen in your code, Flask can be run in 'debug mode'. In this mode, two things happen:

- Any code changes will be detected and will result in Flask auto restarting
- Any exception raised during processing will cause a debugger interface to be returned to the browser, will let you interactively inspect how your code was running when the error occurred.

This can be quite helpful when you are having difficulties figuring out what is causing some exception to occur. To see how to enable this mode, read the Debug Mode section in the QuickStart guide.

Interactive Shell

The last debugging tip to go over is setting up an interactive shell that can let you work with your Flask application. The Flask documentation explains how to set up a shell, as well as explaining how to utilize the shell to make 'requests' inside your application.

Expanding Our Model Into A Database

Now with that out of the way, lets jump into using an actual database within our application. Last week, our 'model' component was very simple: it consisted of a simple list of values, with no persistence across restarts of the application. This is a huge downside! We would like to be able to permanently save data for our users. In order to do so, we need to expand our application into using some kind of persistent data store. In future classes, you may get to use 'NoSQL' databases, like MongoDB or Neo4j, and certainly Flask can support using them. However, we will stick with using SQLite as our persistence layer.

When using a database in your application, you generally need to follow these steps:

1. Create a database schema, or structure, that describes how the data in your application will be stored
2. Initialize your database by loading any data that needs to be there upfront
3. Create a connection to the database in your Flask application
4. Use this connection to retrieve and store data as needed in your 'controller' functions

Let's walk through these steps one by one.

Step 1: Database Schema

For this step, we need to create a set of tables in a SQLite database that will describe our application's data. This is very specific to the application at hand. In order to do this, therefore, you need a good understanding of what your application does, what it needs to keep track of, and what the best way of modeling this is. You should spend a lot of time on this step, as it can be very difficult to fix mistakes in your database once you already have data stored there. When designing web applications, you should expect to spend a lot of time on this step, as it'll save you time in the long run.

Step 2: Initialize your Data

Once you have the database schema ready to go, your application may need to have some data already loaded into the database. For instance, if you have a 'users' table that

keeps track of registered users of your application, then you may need to load some rows that represent administrative users.

Step 3: Connect to the Database

Now that you have a way of storing and retrieving data, we need to connect to it and provide this connection to various parts of our code. We already know how to connect to a SQLite database, using the sqlite3 module (from week 10):

```
conn = sqlite3.connect('path/to/database.db')
```

Step 4: Using the Database Connection

How can we make this connection object available to our code? We could make it a global, but Flask gives us a way of doing this. Flask provides a global object called `g`, which gives us an object we can attach anything we want to. The purpose of this object is to simply make it easier to pass custom objects around, versus having to pass them to all your functions. Therefore, we can attach our database connection to the g object, and the rest of our code can use it.

Flaskr Tutorial

To see all of this in action, please review the Flaskr tutorial, which goes over the development of an example of building a blog application with Flask. Make sure to review the tutorial code that is provided to you on Github. It is very important to see how it all links together.

Object-Relational Mappers

One final note about models. The way we have gone about using a database is a little tedious. For example, in the Flaskr application, the controllers actually execute hand-written SQL statements in order to retrieve information. This is not wrong, per se, but can be very tedious and error prone. This is where ***Object Relational Mappers*** can come in handy.

Object Relational Mappers, or ORMs for short, allow a programmer to interface to a database using Objects. We can think of these ORMs as making an analogy between classes and databases:

- A Class is similar to a table
- An object or instance of a class is similar to a row in a table
- An object's properties are similar to the columns present in the table

While using an ORM can make your life simpler, we will not review them here for brevity. If you are curious about ORMs, check out SQLAlchemy, which is the standard ORM for Python

applications, and Flask-SQLAlchemy, a Flask extension that helps you utilize SQLAlchemy. Also, read chapter 5 in the “Flask Web Development” textbook which covers SQLAlchemy (again, optional). You will be seeing ORMs in some of your future courses, but for now we are going to stick with working with databases in a more manual way.