Tutorial Week 13

HTTP Requests

Whenever we transfer data between the frontend (i.e. the HTML) and the backend (i.e. Flask), we do so using something called an **HTTP request**. There are four main **HTTP request methods**; one of which we have been using a lot of (i.e. the `GET` method):

Method name	Description
`GET`	Used to retrieve data from the server
`POST`	Used to send data to the server
`PUT`	Used to update pre-existing data on the server
`DELETE`	Used to delete pre-existing data on the server

Each of the HTTP request methods is denoted by the `method=` attribute in the `<form>` tag in your HTML. If you do not denote a method, it defaults to a `GET` request. So far, we have only been reading data from the server so we have only needed to use the `GET` request. At times, we want to look for specific parameters from the backend. For instance, in the last lab, we wanted to look for only employees with a specific name. To do this with the `GET` request, we added these values into the URL itself using **query parameters** and we accessed them in Flask using `request.args.get()`. This is because by the standard, we are not allowed to pass data alongside a `GET` request; we can only add "data" that can be read by the server through the URL itself

If we want to send data to the server (e.g. such as sending anonymous feedback to an instructor), we want to do so using a `POST` request instead of a `GET` request. To do this, we slightly modify our `<form>` in the HTML code by adding the attribute `method="POST"` to account for this:

```
<form action="/new-feedback" method="POST">
 <input type="text" name="author-name" placeholder="Enter your name">
 <input type="text" name="professor-name" placeholder="Enter the professor's name">
```

```
  <textarea name="anon-comment" cols="30" rows="10"></textarea>
  <input type="submit">
</form>
```

This attribute will tell Flask that we are intending to send it a POST request to the route denoted at action=. In order to catch the data, we also need to modify our Flask route accordingly:

```
@app.route('/new-feedback', methods=['POST'])
def new_employee():
  # ... database initialization ...

  # a POST request's data is put into something called a "body"
  new_feedback = request.form

  # ... do some stuff ...

  return redirect(url_for('root'))
```

Inside the app.route(), we need to specify to Flask that we intend to catch a request of type POST (not putting anything defaults to GET). Unlike the GET request, the POST request does not send the data from the form in the URL but instead puts it in something called the **POST body** which is a section of the request meant to hold data. There are two main reasons why POST requests don't put data in the URL:

1. Adding a large amount of data will clutter the URL
    - URLs actually have a limit of around 2000 characters
2. GET requests are not allowed to carry data whereas POST requests are
    - They do not have the same limitations as GET requests

In order to access the data, we access it through a variable called request.form which represents the POST body. This variable is in a form called **JavaScript Object Notation** or **JSON** for short. Flask sees this data type as a regular Python dictionary where the keys are equal to the name= attribute in your HTML form and the values are whatever were entered into them upon submission. An example of what the above POST body might look like is as follows:

```
{
  "author-name": "Kevin",
  "professor-name": "Prof",
  "anon-comment": "keviniscool"
}
```

Cursors

In the previous lab (and in today's starter code) there are two functions called `get_db()` and `query_db()` which simplify some of the database interaction for querying the database. In order to perform INSERT queries, we need to understand (at a high level) what is happening in these two functions

`get_db()` is a pretty straightforward function. It attempts to make a connection to the database at the path `DATABASE` and otherwise creates one at that path if it does not exist

`query_db()` is a simplification of something called a **cursor action** in the database.
A **cursor** in a database is sort of like a cursor on a computer (i.e. the mouse pointer). It sort of "hovers" over the database and performs the queries in the database. When you run queries in SQLite browser or on the command-line, these programs create a cursor for you that you don't see and perform what is happening in this function. To go over this function in a little more detail:

```
def query_db(query, args=(), one=False):
    # create a cursor called "cur" and execute the query "query" with arguments "args"
    cur = get_db().execute(query, args)
    # get all the results (this function only works for SELECT statements)
    rv = cur.fetchall()
    # close the connection (do not leave an open connection)
    cur.close()
    # return the results depending on if there are many or just one
    return (rv[0] if rv else None) if one else rv
```

The general process of executing any sort of SQL query is a 3-step process:

1. Create a cursor
2. Run the query using `.execute()`

    - Also commit the changes if modifying data (we will go into this in a second)
3. Close the cursor

For a SELECT query, the cursor does not modify any data in the database so you will immediately get back the result of the query. For a query that modifies data in the database (such as an INSERT or an UPDATE), the database does not automatically apply the query when it is executed. Instead, it "stage" the query and waits until you "commit" it to apply the changes to the database. An example of this in action is as follows:

```
# suppose we are trying to add a new student to a database at this route
@app.route('/new-student')
def new_student():s
  db = get_db()
  db.row_factory = make_dicts
  # make a new cursor from the database connection
  cur = db.cursor()

  # get the post body
  new_student = request.form

  # insert the new student into the database
  cur.execute('insert into students (firstname, lastname, grade) values (?, ?, ?)', [
    new_student['firstname'],
    new_student['lastname'],
    new_student['grade'],
  ])
  # commit the change to the database
  db.commit()
  # close the cursor
  cur.close()

  return redirect(url_for('root'))
```

Exercise

NOTE: you must do this exercise on your own computer; you will not be able to finish this lab on the lab computers

Today's lab will be an extension of last week's lab. For help with writing SQL queries, refer to last week's lab handout

Starter code is given here

Make the following improvements to the application:

1. For a given employee, add the functionality to leave them feedback for their job performance
    - **HINT**: consider creating a new table to hold all the comments that references the employee table
    - The employee feedback page should link to another page
        - You will access this page by doing a search for that employee (similar to last week's lab)
    - A feedback comment should be composed of the following attributes
        - Feedback author name
        - Comment itself
2. Add the functionality to add new employees from the database
    - **BONUS**: consider adding functionality to remove users from the database