HTML Forms

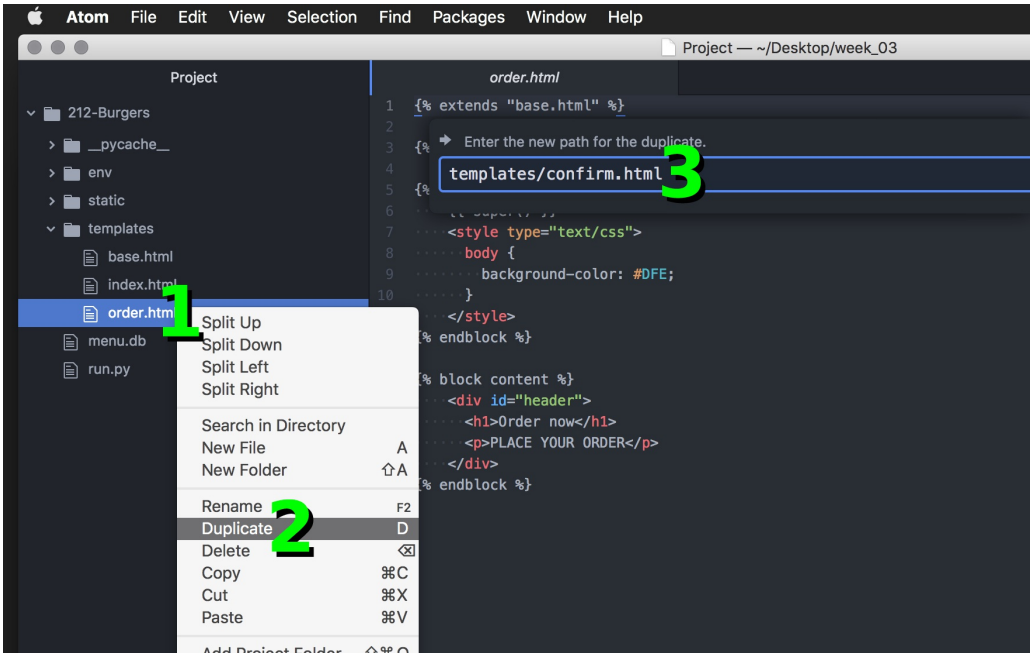
Forms allow one to capture information, which can then be entered into a database or used for other puroposes. For this website, users will order burgers, drinks, and sides -- as well provide their names and delivery addresses.

However, before creating an order form, begin by adding an order confirmation page.

Creating a Confirmation Page

Duplicate the order.html template file and name it "confirm.html". If you are using Atom, this can be accomplished by:

1. right-clicking on the order.html file in the Project panel;
2. selecting *Duplicate*;
3. and then entering the file name (confirm.html).



You will also need to edit the confirm.html file, changing the `title`, `h1`, and `p` content; and adding the `<div class="center-text"> ...` section to the code:

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

{% block title %}Confirm Order - 212 Burgers{% endblock %}
```

```
...

{% block content %}

    <div id="header">
      <h1>Confirm</h1>
      <p>ORDER CONFIRMATION</p>
    </div>

    <div class="center-text">
      <p>
        <a class="btn-yellow" href="/">home page</a>
      </p>
    </div>

{% endblock %}
```

As you can tell from the `href` attribute, the button links back to the landing page. To center it, add a corresponding CSS rule to your `static/screen.css` file:

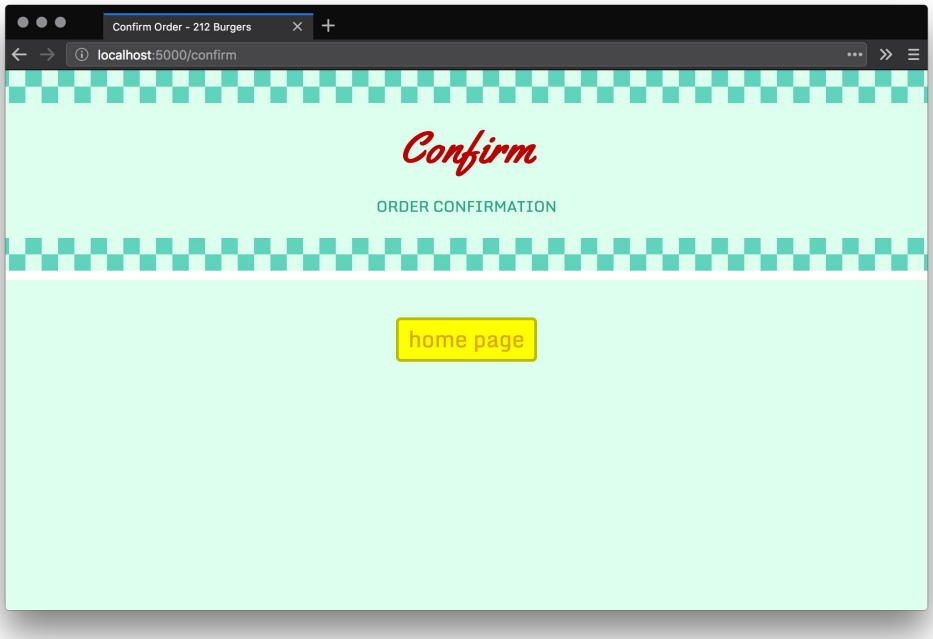
```
...

.center-text {
  text-align: center;
}

...
```

With the confirm.html template completed, you can now add the route for the confirm page:

run.py

```
...

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

Open your browser and load the confirm page:
http://localhost:5000/confirm



It's not much to look at now, but will soon list the customer's order details.

Creating an Order Form

Open the order.html template in your editor (Atom?). Add the menu beneath the PLACE YOUR ORDER paragraph. This is, essentially, the same menu from the landing page with some number `input` fields added:

order.html

```
...

{% block content %}

    <div id="header">
      <h1>Order now</h1>
      <p>PLACE YOUR ORDER</p>
    </div>

    <form action="/confirm" method="post">

      <div id="menu">

        <h1>Menu</h1>

        <div>
          <h2>Burgers</h2>
```

```
            <ul class="menu-items">
              {% for burger in burgers %}
              <li>
                <input type="number" min="0" name="{{ burger[0] }}" autocomplete="off" />
                {{ burger[0] }}
                <span class="price">{{ burger[1] }}</span>
              </li>
              {% endfor %}
            </ul>
          </div>

          <div>
            <h2>Drinks</h2>
            <ul class="menu-items">
              {% for drink in drinks %}
              <li><input type="number" min="0" name="{{ drink[0] }}" autocomplete="off" />
              {{ drink[0] }}
              <span class="price">{{ drink[1] }}</span></li>
              {% endfor %}
            </ul>
          </div>

          <div>
            <h2>Sides</h2>
            <ul class="menu-items">
              {% for side in sides %}
              <li><input type="number" min="0" name="{{ side[0] }}" autocomplete="off" />
              {{ side[0] }}
              <span class="price">{{ side[1] }}</span></li>
              {% endfor %}
            </ul>
          </div>

        </div>

      </form>

  {% endblock %}
```

What is important to note is that each input has a unique `name` attribute. This will accompany the value submitted, i.e.

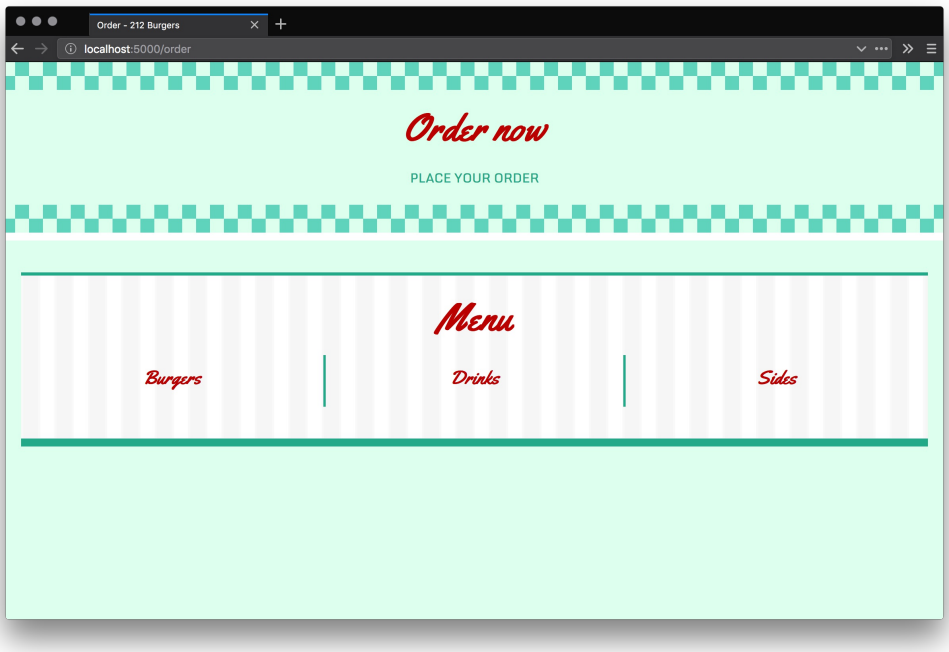
`Classic Burger: 1`

`Beer: 1`

```
Fries: 2
```

This will make more sense further along when you are dealing with the form data.

Save, then open your browser and load the order page:
http://localhost:5000/order



The lists are empty because no menu data has been provided for the template variables (`burgers`, `drinks`, `sides`). To fix this, edit the `order()` function in the *run.py*:

```
...

@app.route('/order')
def order():
  con = sqlite3.connect(MENUDB)

  burgers = []
  free = '0'
  cur = con.execute('SELECT burger,price FROM burgers WHERE price>=?', (free,))
  for row in cur:
    burgers.append(list(row))

  drinks = []
  cur = con.execute('SELECT drink,price FROM drinks')
  for row in cur:
    drinks.append(list(row))
```

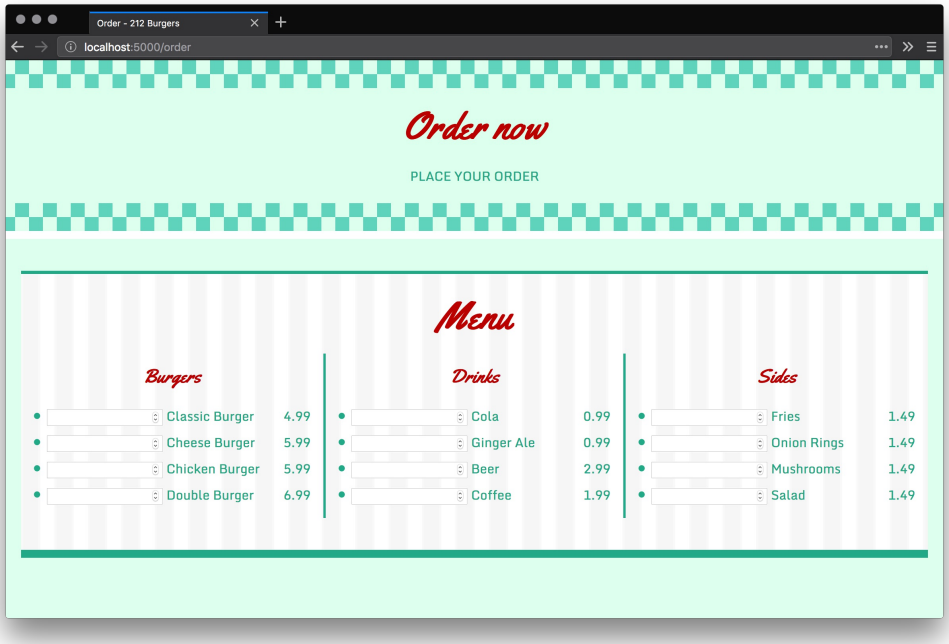
```
  sides = []
  cur = con.execute('SELECT side,price FROM sides')
  for row in cur:
    sides.append(list(row))

  con.close()

  return render_template('order.html', burgers=burgers, drinks=drinks, sides=sides)

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

You will notice, however, that this is an exact duplicate of of the `index` route's query. To avoid having the same code appear twice in the run.py file, define a function (`fetchMenu`) instead; then edit the `index()` and `order()` functions accordingly:

```
...

MENUDB = 'menu.db'

def fetchMenu(con):
  burgers = []
  free = '0'
  cur = con.execute('SELECT burger,price FROM burgers WHERE price>=?', (free,))
  for row in cur:
    burgers.append(list(row))

  drinks = []
  cur = con.execute('SELECT drink,price FROM drinks')
  for row in cur:
    drinks.append(list(row))

  sides = []
  cur = con.execute('SELECT side,price FROM sides')
  for row in cur:
    sides.append(list(row))

  return {'burgers':burgers, 'drinks':drinks, 'sides':sides}

@app.route('/')
def index():
  con = sqlite3.connect(MENUDB)
```

```
    menu = fetchMenu(con)
    con.close()
    return render_template(
      'index.html',
      disclaimer='may contain traces of nuts',
      burgers=menu['burgers'],
      drinks=menu['drinks'],
      sides=menu['sides']
    )

@app.route('/order')
def order():
    con = sqlite3.connect(MENUDB)
    menu = fetchMenu(con)
    con.close()
    return render_template('order.html', burgers=menu['burgers'], drinks=menu['drinks'], sides=menu['sides'])

...
```

Save, then refresh the order page:



The database's menu data is now coming through. The user can specify how many of each item they wish to order. However, you will also need their name and address, so add the following order-details code to your form:

order.html

```
            ...
            <span class="price">{{ side[1] }}</span></li>
            {% endfor %}
          </ul>
        </div>

      </div>

      <div id="order-details">
        <p>
          <label for="name">Name:</label>
          <input type="text" id="name" name="name" required />
        </p>
        <p>
          <label for="address">Address and delivery instructions:</label>
          <textarea id="address" name="address" rows="6" required></textarea>
        </p>
        <input type="submit" value="order" class="btn-yellow" />
      </div>

    </form>

{% endblock %}
```

Save, then refresh the order page:

Note that I have zoomed-out a bit to fit everything within the browser window.

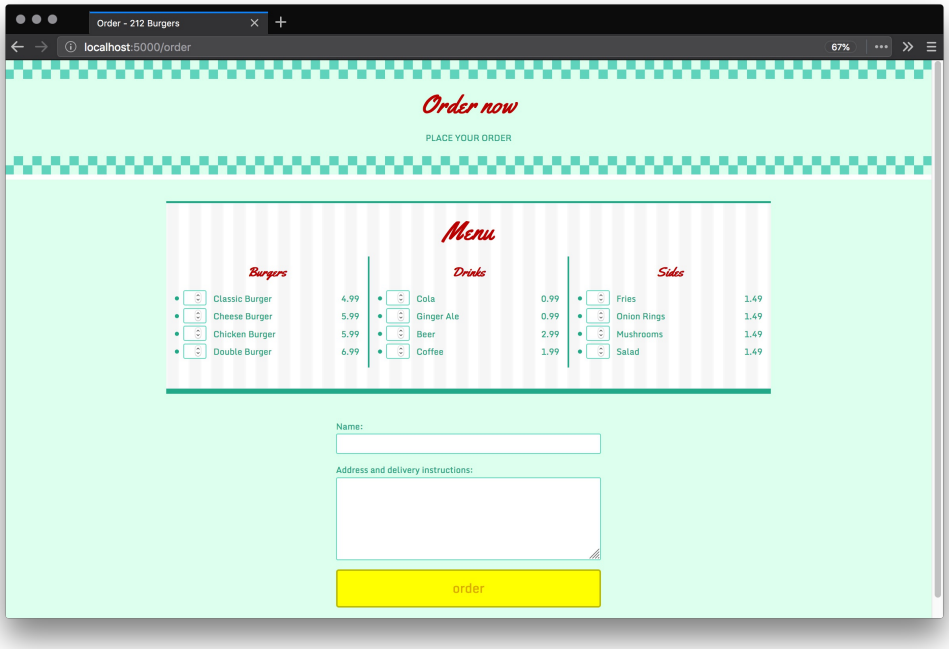
You now have all of the input fields you require, but these could do with some styling. Open your stylesheet and add the following rules:

screen.css

```
...

/* forms */

input, textarea {
  border: solid 2px #5DB;
  border-radius: 3px;
  color: #2A8;
  font-family: 'Monda', sans-serif;
  font-size: 0.8em;
  height: 2em;
  padding: 0 0.5em;
}

textarea {
  height: auto;
  width: 100%;
}

#menu input, #menu textarea {
  float: left;
  margin-right: 1em;
  margin-top: -0.15em;
  width: 2.2em;
}

#order-details {
  margin: auto;
  max-width: 500px;
  padding: 2em 2.5em;
}

#order-details input,
#order-details textarea {
  box-sizing: border-box;
  min-height: 3em;
  width: 100%;
```

```
}


...
```

Save, then refresh the order page:



For now, the browser validates any input. You could add some JavaScript validation -- then further validate the input using Python -- but to keep things simple, this tutorial will not cover such techniques.

Submitting Form Data to Flask

In the *order.html* file, take note of the opening form tag's `action` and `method` attributes:

```
<form action="/confirm" method="post">
```

The `method` can either be `get` or `post`. Get passes all of the form data across the address bar; for example, one can search Google for "dog" using the following url:

https://www.google.com/search?q=dog

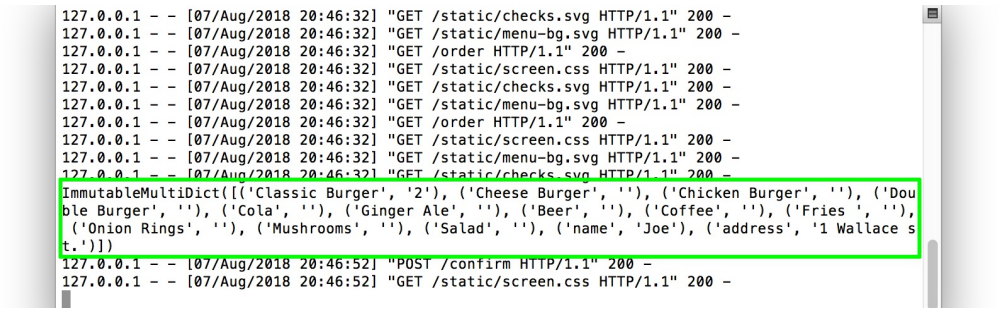
This is because Google search form uses `get`.

The `post` method conceals this information, which makes it better suited for submitting sensitive form data (although, further steps should be taken to secure things properly). We will use post in this instance.

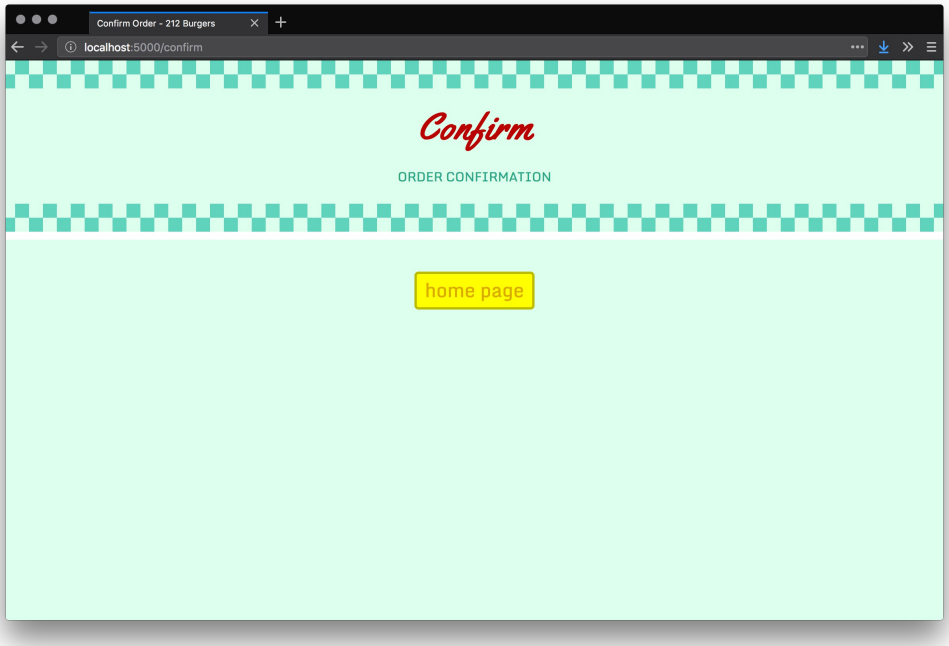
Because you will be submitting form data to the `confirm` route, it must be configured to accept `post` requests -- this entails importing the Flask `request` functionality and adding the `methods=['POST']` argument to the route line:

```
from flask import Flask, render_template, request

...

@app.route('/confirm', methods=['POST'])
def confirm():
  print(request.form)
  return render_template('confirm.html')
```

The `print(request.form)` line will print whatever is captured to the Terminal. Give it a spin by placing an order using the order form; then clicking "order" button. In this example, Joe of 1 Wallace street has ordered 2 Classic Burgers:



The problem is that the confirm page is blank:



Fix this by amending your confirm function:

run.py

```
@app.route('/confirm', methods=['POST'])
def confirm():
  details = {}
  items = {}

  for input in request.form:
    if input == 'name' or input == 'address':
      details[input] = request.form[input]
    elif request.form[input] and request.form[input] != '0':
      items[input] = request.form[input]

  return render_template('confirm.html', details=details, items=items)
```

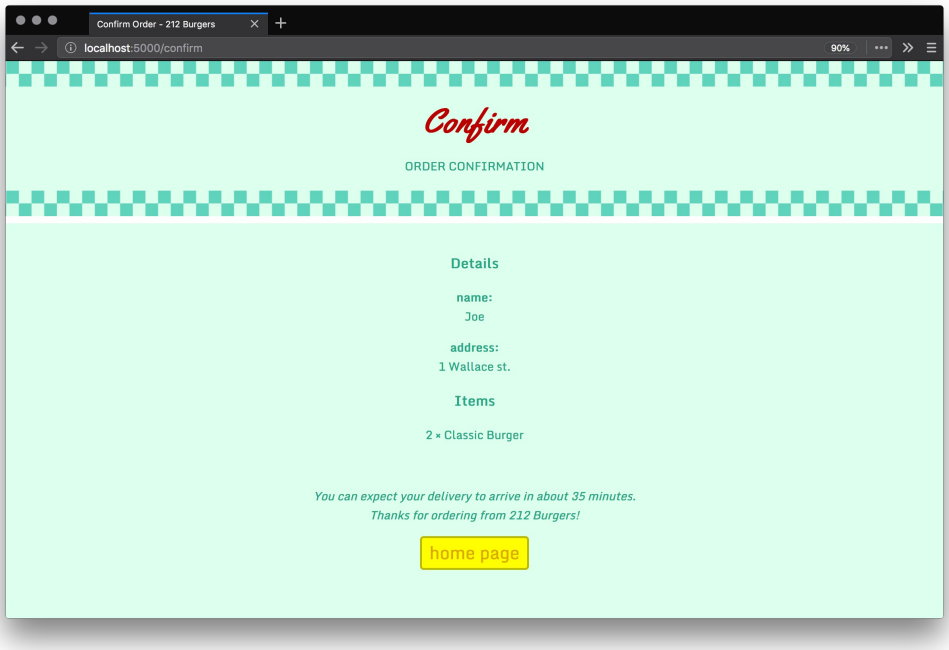
The `for` loop above prunes the data, omitting any zero or empty values. The refined `details` and `items` dictionaries (associate arrays) are then passed to the template. To render them, add some Details and Items code to your confirm template:

confirm.html

```
...

<div class="center-text">

  <div>
    <h3>Details</h3>
    {% for key in details %}
    <p><strong>{{ key }}:</strong><br />{{ details[key] }}</p>
    {% endfor %}
  </div>

  <div>
    <h3>Items</h3>
    {% for item in items %}
    {{ items[item] }} &times; {{ item }}<br />
    {% endfor %}
  </div>

  <p> </p>
```

```
  <p>
    <em>You can expect your delivery to arrive in about 35 minutes. <br />Thanks for ordering from 212
Burgers!</em>
  </p>
  <p>
    <a class="btn-yellow" href="/">home page</a>
  </p>

</div>

{% endblock %}
```

Save, then refresh the confirm page:



Entering Data into a Database

Open your database (in a new Terminal window):

`sqlite3 menu.db`

and create a new table:

```
CREATE TABLE orders(
  id INTEGER PRIMARY KEY,
  name TEXT,
  address TEXT,
```

```
  items TEXT
)
```

```
(env) [tabunn]week_03$ sqlite3 menu.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE TABLE orders(
   ...>   id INTEGER PRIMARY KEY,
   ...>   name TEXT,
   ...>   address TEXT,
   ...>   items TEXT
   ...> );
sqlite>
```

Now amend your confirm code, adding the four lines for connecting to the database and inserting the data:

```
@app.route('/confirm', methods=['POST'])
def confirm():
  details = {}
  items = {}

  for input in request.form:
    if input == 'name' or input == 'address':
      details[input] = request.form[input]
    elif request.form[input] and request.form[input] != '0':
      items[input] = request.form[input]

  con = sqlite3.connect(MENUDB)
  cur = con.execute(
    'INSERT INTO orders(name, address, items) VALUES(?, ?, ?)',
    (details['name'], details['address'], str(items))
  )
  con.commit()
  con.close()

  return render_template('confirm.html', details=details, items=items)
```

The `(?, ?, ?)` part -- split across two lines to make it more legible -- is substituted by the variables that follow it. This will prevent SQL injection attacks. The `con.commit()` is necessary because you are performing a change to the database. The `str()` function converts the dictionary/associative-array to a string.

Refresh the confirm page. You can verify that your data has been entered in the SQLite database by running:

```
SELECT * FROM orders;
```

Sessions

The lesson files (289.212.04.files on Stream) include a 'control panel' feature for reviewing orders. Take a look at the source code to see how things work. You can also run it (`cd` to the directory, setup a virtual environment, etc.) using:

- login page: http://localhost:5000/login
- username: `admin`

For further information on using Flask, refer to the official documentation:
http://flask.pocoo.org/docs/1.0/

end